Enter the Hydra: Towards Principled Bug Bounties and Exploit-Resistant Smart Contracts*

Lorenz Breidenbach[1, 2, 5], Phil Daian[2, 5], Florian Tramèr[3], and Ari Juels[4, 5]

[1]ETH Zürich
[2]Cornell Tech
[3]Stanford University
[4]Cornell Tech, Jacobs Institute
[5]The Initiative for CryptoCurrencies and Contracts (IC3)

***Abstract*—Vulnerability reward programs, a.k.a. *bug bounties*, are a popular tool that could help prevent software exploits. Today, however, they lack rigorous principles for setting bounty amounts and require high payments to attract economically rational hackers. Rather than claim bounties for serious bugs, hackers often sell or exploit them.**

We present the *Hydra Framework*, the first general, principled approach to modeling and administering bug bounties and boosting incentives for hackers to report bugs. The key idea is what we call an *exploit gap*, a program transformation that enables runtime detection of security-critical bugs. The Hydra Framework transforms programs via *N-of-N-version programming* (NNVP), a variant of classical N-version programming that executes multiple independent program instances.

We apply the Hydra Framework to *smart contracts*, small programs that execute on blockchains. We show how Hydra contracts greatly amplify the power of bounties to incentivize bug disclosure by economically rational adversaries, establishing the first framework for economic evaluation of smart contract security. We also model powerful adversaries capable of *bug withholding*, exploiting race conditions in blockchains to claim bounties before honest users can. We present *Submarine Commitments*, a countermeasure of independent interest that conceals transactions on blockchains.

We present a simple core Hydra Framework for Ethereum. We report the implementation of two Hydra contracts—an ERC20 token contract and a Monty-Hall-like game.

I. Introduction

Despite theoretical and practical advances in code development, software vulnerabilities remain an ineradicable security problem. Vulnerability reward programs—a.k.a. *bug bounties*—have thus become instrumental in many organizations' security assurance strategies. These programs offer rewards as incentives for hackers to disclose software bugs. Unfortunately, hackers often prefer to exploit critical vulnerabilities or sell them in underground markets.

The chief reason for this choice is that the bugs eligible for large bounties are generally weaponizable vulnerabilities. The financial value of critical bugs (0-days) in gray markets may exceed bounty amounts by a factor of as much as ten to one hundred [1]. For example, while Apple offers a maximum 200k USD bounty, a broker intermediary such as Zerodium purportedly offers 1.5 million USD for certain iPhone jailbreaks. In some cases hackers can monetize vulnerabilities themselves for large payouts [2], [3]. Modest bounties may thus fail to successfully incentivize disclosure [4].

Pricing bounties appropriately can also be hard because of a lack of research giving principled guidance. Payments are often scheduled arbitrarily based on bug categories and may not reflect bugs' market value or impact. For example, Apple offers up to 100k USD for "Extraction of confidential material protected by the Secure Enclave Processor" [4].

Finally, bounty payments present a problem of fair exchange. A bounty payer does not wish to pay before reviewing an exploit, while hackers are wary of revealing exploits and risking non-payment or mis-payment of bounties (e.g., [5], [6], [7]). This uncertainty creates a market inefficiency that limits incentives for hackers to uncover vulnerabilities.

In this paper, we introduce the *Hydra Framework*, the first principled approach to bug bounty administration that addresses these challenges. The Hydra Framework deters economically rational actors, including black-hat hackers, from exploiting bugs or selling them in underground markets. We focus on smart contracts as a use case to demonstrate the framework's power analytically and empirically.

The Hydra Framework. The key to the Hydra Framework is to build support for bug detection and bounties into software at development time using a concept that we call an *exploit gap*. This is a program transformation that makes critical bugs *detectable* at runtime, but *hard to exploit*.

We propose an exploit gap technique that we refer to as *N-of-N-version programming* (NNVP). A variant of classical N-version programming, NNVP leverages multiple versions of a program that are independently developed, or otherwise made heterogeneous. In the Hydra Framework, these program versions, or *heads*, are executed in parallel within a meta-program that we call a *Hydra program*.

In stark contrast to N-version programming's goal of *fault tolerance* (i.e., where the program attempts to produce a correct output even in the face of partial failures), NNVP focuses on *error detection and safe termination*. If heads' outputs are identical, a Hydra program runs normally. If the outputs diverge for some input, a dangerous state is indicated

*The first three authors contributed equally to this work.

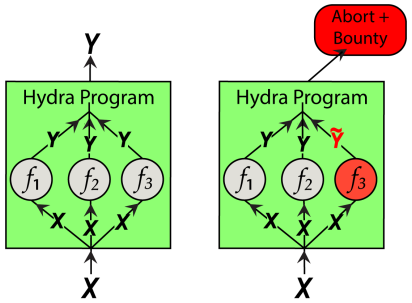

Fig. 1: Hydra program with heads f_1, f_2, and f_3. Example on right shows effect of bug induced by input X in f_3.

and the program aborts. The Hydra program may then trigger payment of a bounty for this bogus input or take other actions. The basic idea is depicted in Figure 1.

A bug is only exploitable if it affects all Hydra heads identically. If failures are somewhat uncorrelated across heads, a bug in one head is thus unlikely to affect the Hydra program as a whole. Similarly, an adversary that breaks one head and, instead of claiming a bounty, tries to generalize the exploit, risks preemption by honest bounty hunters. Modest bounties can thus incentivize economically rational hackers to disclose bugs rather than attempt an exploit. We show that even when an exploit's market value exceeds the bounty by multiple orders of magnitude, disclosing bugs in Hydra heads yields higher expected payoff than attempting a full exploit.

A Hydra Framework for smart contracts. We focus on smart contracts, programs that execute in blockchain systems such as Ethereum [8]. They are especially well suited as a use case given several distinctive properties:

- *Heightened vulnerability:* Smart contracts are often financial instruments. Program bugs usually directly affect funds, enabling hackers to extract (pseudonymous) cryptocurrency, as shown by tens of millions of dollars worth of Ethereum stolen from [2] and [3]. Smart contract binaries are publicly visible and executable, and often open-source. Given their high value and exposure to adversarial study and attack, smart contracts urgently require new bug-mitigation techniques.
- *Unique economic properties:* Smart contracts often carry cryptocurrency balances that represent a direct measure of their value at risk and exploit value. This fact facilitates principled bounty price setting in our framework. Moreover, the protocols that underpin these systems are often secured by a combination of cryptography and economic guarantees. Creating similar quantifiable economic guarantees of correctness at the smart contract level is an open problem we address.
- *Bounty automation:* Application of our framework to and by smart contracts can award bounties automatically. The result is a *fair exchange* of bugs for bounties and *guaranteed payment* for the first valid submitted bug. Bounties are *transparent* to bounty hunters and can be adjusted *dynamically* to reflect contracts' changing exploit value, creating a stable bounty marketplace.
- *Graceful termination conditions:* Smart contracts are not (yet) mission critical software and can often be aborted with minimal adverse effects for users, as required for NNVP. Remediation of the DAO and Parity multsig attacks involved refunding money to users, a mechanism we consider in this paper.

We implement a simple Hydra Framework library for Ethereum and evaluate it on two applications, an ERC20 token [9] and a Monty Hall game [10]. In both cases, our Hydra contract automatically rewards bugs in one of three deployed heads, implemented in three different languages in the Ethereum ecosystem. Our Hydra ERC20 token is deployed on the Ethereum main network (with an initial 1,000 USD bounty), the first example of a principled, automated and trust-free bug bounty. We will release our framework and smart contracts prior to publication of this paper.

Major challenges. Several well-cited papers [11], [12] criticize traditional N-version programming, observing that multiple versions of a program often exhibit correlated faults—an ostensible hitch in the Hydra Framework.

We revisit these papers and show that NNVP achieves an appealing cost-benefit trade-off, by abandoning *fault-tolerance* in favor of *error detection*. Compared to N-version programming majority voting scheme, partial independence is greatly amplified by NNVP, which requires agreement by *all* heads. Previous experimental results in fact show that NNVP can achieve a large exploit gap in Hydra programs. In particular, we review high-profile smart contract failures, showing that NNVP would have addressed many of them.

A second challenge arises in automating bug bounties for smart contracts. Decentralized blockchain protocols allow adversaries to perform *front-running*—ordering their transactions ahead of those of honest users [13]. As a result, a naïvely implemented bounty smart contract is vulnerable to a *bug-withholding* attack. Upon discovering an exploit for one head, a hacker can withhold it and try to compromise the remaining heads to exploit the full contract. If an honest user discovers a bug, the hacker front-runs her and claims the bounty first. Thus, withholding carries no cost for the hacker, removing incentives for early disclosure.

We propose *Submarine Commitments*, a countermeasure of independent interest. Our technique temporarily conceals a bounty claim among ordinary transactions, preventing a hacker from observing and front-running a claim. We formally define security for Submarine Commitments and prove that they effectively prevent bug withholding.

Contributions. In summary, our main contributions are:

- *The Hydra Framework:* We propose, analyze, and demonstrate the first general approach to principled bug bounties. We introduce the idea of an *exploit gap* and explore *N-of-N-version programming* (NNVP) as a specific instantiation. We demonstrate the power of NNVP Hydra programs in revisiting the N-version programming literature and reviewing high-profile smart contract failures that it could have prevented. We provide the first quantifiable notion of economic security for smart contracts and analyze the resulting.
- *Bug withholding and Submarine Commitments:* We identify the subtle *bug withholding* attack. To analyze its

security, we present a strong, formal adversarial model that encompasses front-running and other attacks. We introduce a countermeasure of independent interest called *Submarine Commitments* and prove that it effectively prevents bug withholding.
- *Implementation:* We describe and implement a simple Hydra Framework for Ethereum smart contracts. We present Hydra implementations of an ERC20 token and Monty Hall game. We empirically measure the costs in scaling smart contracts to multiple heads on-chain, and explore cost-reduction strategies. We launched our bounty-backed Hydra ERC20 token on Ethereum.

Organization. We present the concept and formalism for *exploit gaps* in Section II. We discuss and justify our use of N-of-N-version programming (NNVP), and present Hydra contracts in Section III. We formally analyze the interplay between bug bounties and exploit gaps to incentivize bug disclosure in Section IV. In Sections V and VI, we introduce the bug-withholding attack and *Submarine Commitments* as a countermeasure, along with a formal model in which to prove their effectiveness. We present an implementation and evaluation of ERC20 and Monty Hall Hydra contracts in Section VII. Finally, we review related work in Section VIII, concluding in Section IX. Details on Submarine Commitment implementation and security proofs are in the appendix.

II. Preliminaries and Notation

We now define some notation useful to our Hydra model.

A. Programs

Let f denote a stateful program. From a state s, running f on input x produces output y and updates s. For an input sequence $X = [x_1, x_2, \dots]$, we denote by $\texttt{run}(f, X) \coloneqq [y_1, y_2, \dots]$ a serial *execution trace* of f starting from the initial state and producing output y_i on input x_i.

B. Exploits

For a program f, we let $\mathcal{I}$ be an abstract *ideal program* that defines the intended behavior of f. That is, for any input X, the output of $\texttt{run}(\mathcal{I}, X)$ is correct. We assume that the input space is bounded and that input sequences are finite.

We further assume that a program may produce a *fallback* output $\perp$ if it detects that the execution is diverging from the intended behavior (e.g., throwing a runtime exception if a stack canary detects a stack overflow). The ideal program $\mathcal{I}$ never outputs $\perp$. If a program f outputs $\perp$ on some input x_i, then all subsequent outputs in that execution trace will also be fallbacks. A program's execution trace is a *fallback trace* if it agrees with the ideal program up to some input x_i, and then outputs $\perp$. The set of fallback traces is

$$\mathcal{Y}_\perp \coloneqq \left\{ Y \mid \exists i. [y_1, \dots, y_i] \sqsubset \texttt{run}(\mathcal{I}, X) \wedge \textstyle\bigwedge_{j=i+1}^{n} (y_j = \perp) \right\} ,$$

where $A \sqsubset B$ means that sequence A is a strict prefix of sequence B.

We naturally define an *exploit* against f as any sequence of inputs X for which f's output is neither that of the ideal program nor a fallback trace. If $E(f, \mathcal{I})$ denotes the *exploit set* of f with respect to $\mathcal{I}$, then $X \in E(f, \mathcal{I})$ if and only if $\texttt{run}(f, X) \notin \mathcal{Y}_\perp \cup \{\texttt{run}(\mathcal{I}, X)\}$. Note that the notions of ideal program, fallback output, and exploit are oblivious to the actual representation of the program's internal state.

C. Exploit Gaps and Bug Bounties

A program transformation $\mathcal{T}$ aggregates a set of $N \geq 1$ programs into a new program $f^* \coloneqq \mathcal{T}(f_1, f_2, \dots, f_N)$. Our definition of exploit gap aims to capture the natural notion that f^* has fewer exploits than the original f_i. However, directly relating the sizes of $|E(f^*, \mathcal{I})|$ and $|E(f_i, \mathcal{I})|$ is problematic as we cannot measure these quantities in practice. Instead, we define a *probabilistic* notion of exploit gap, for input sequences X sampled from a distribution $\mathcal{D}$ (e.g., the distribution of user inputs to a program).

Definition 1 (Exploit Gap)**.** *A program transformation $\mathcal{T}(f_1, f_2, \dots, f_N) \coloneqq f^*$ introduces an* affirmative exploit gap *for a distribution $\mathcal{D}$ over input sequences X if we have*

$$\mathit{gap} \coloneqq \frac{\Pr_{X \in \mathcal{D}} \left[X \in \bigcup_{i=1}^{N} E(f_i, \mathcal{I}) \right]}{\Pr_{X \in \mathcal{D}} \left[X \in E(f^*, \mathcal{I}) \right]} > 1 . \qquad (1)$$

The exploit gap is empirically measurable and its magnitude reflects the likelihood that an input sequence that is an exploit for some f_i does not affect f^*.

A transformed program f^* that always returns $\perp$ trivially induces a large exploit gap, while not having any utility. We therefore also require the following notion of availability.

Definition 2 (Availability Preservation)**.** *Let $F(f)$ be the set of input sequences that lead to a fallback output, i.e. $X \in F(f)$ iff $\texttt{run}(f, X) \in \mathcal{Y}_\perp$. Then a program transformation $\mathcal{T}$ is* availability-preserving *iff*

$$F(f^*) \subseteq \textstyle\bigcup_{i=1}^{N} \left(E(f_i, \mathcal{I}) \cup F(f_i) \right)$$

To be availability-preserving and yield an exploit gap, a program transformation may trade availability for correctness. That is, a transformed program may fallback on inputs that are exploits for one or more of the original programs.

Given a transformation $\mathcal{T}$ that induces an affirmative exploit gap, a natural bug bounty for a deployed program f^* rewards exploits in the original programs f_i. Some of these exploits may be harmless against f^*, so attackers cannot sell or exploit them. We further want the disclosure of such bugs to be "useful" towards ultimately improving the security of f^*, a notion captured with the following natural definition:

Definition 3 (Monotonicity)**.** *For a program f_i let f_i' be such that $E(f_i', \mathcal{I}) \subset E(f_i, \mathcal{I})$. A program transformation $\mathcal{T}$ is* monotone *if for any such f_i, f_i',*

$$E(\mathcal{T}(f_1, \dots, f_i', \dots, f_N), \mathcal{I}) \subseteq E(\mathcal{T}(f_1, \dots, f_i, \dots, f_N), \mathcal{I}) .$$

Monotonicity says that fixing bugs in the original programs can only reduce the exploit set of f^*.

Given such a transformation, the bug bounty scheme described previously satisfies three important properties:

1) The bugs are efficiently verifiable, via *differential testing*: If $\texttt{run}(f_i, X) \neq \texttt{run}(f^*, X)$, then the input X is an exploit against f_i or f^* or both.
2) A claimable bug need not be an exploit on f^*. If the exploit gap is large (i.e., $\texttt{gap} \gg 1$), then it is likely that a submitted bug affects one of the programs f_i but not f^*. The output of f^* on such an input is either correct (according to $\mathcal{I}$), or a fallback output $\perp$.
3) The bugs are valuable. As $\mathcal{T}$ is monotone, finding and fixing bugs in the original programs must eventually reduce the exploit set of f^*.

D. Achieving an Affirmative Exploit Gap

We give examples of program transformations (some on $N = 1$ programs) that may induce exploit gaps suitable for bug bounties. In Section III, we present N-of-N-version Programming, the transformation we use in this work.

Runtime checks. The addition of runtime checks (e.g., stack canaries, assertions, under- or overflow detection) is an availability-preserving transformation on a single program: the checks may result in a fallback output (e.g., a runtime exception), where the original program had an exploit.

N-version programming. A more broadly applicable program transformation that satisfies our requirements is the use of *redundancy* in *fault-tolerant* systems. Prominent examples include *Recovery Blocks* [14] and *N-version programming* [15]. These transformations operate on $N > 1$ programs and aim at full availability (i.e., no fallback outputs), a natural requirement in mission-critical systems.

We focus on N-version (or *multiversion*) programming, a paradigm we build upon in Section III. This software development process consists of three core steps [15], [16]:

1) A specification (not necessarily a formal one) is written, that describes the program's functionality, API, and error handling. It further specifies how the outputs of different versions are combined (Step 3 below).
2) N versions of the program specification are independently developed. Independence among versions is promoted via *isolation* (i.e., minimal interactions between developers) and *diversity* (i.e., different programming languages, or technical backgrounds of developers).
3) A meta-program runs the N versions in isolation and combines their outputs according to some voting scheme.

N-version programming traditionally uses majority voting to aggregate the programs' outputs [15], [16], [17]. It has been shown that majority voting may induce only a small exploit gap, if program failures are somewhat correlated [12].

III. N-of-N-version Programming: An Exploit Gap Tailored for Smart Contracts

N-version programming builds upon the assumption that heterogeneous implementations have weakly correlated failures [15]. However, this assumption has been challenged by various experiments [11], [12] questioning the cost-benefit trade-off of the paradigm. Our thesis is that *smart-contract ecosystems present a number of key properties that render multiversion programming and derived bug-bounty schemes attractive*. These properties are absent in settings considered in prior experiments with N-version programming.

The main differentiator between the traditional setting of N-version programming, and ours, is the role of *availability*. Prior work focuses on mission-critical software and thus favors *availability* over *safety* in the face of partial failures. For instance, Eckhardt et al. [12] explicitly ignore the "*error-detection capabilities*" of multiversion programming. This setup is not suitable for a smart-contract environment. Indeed, as in centralized financial institutions (including stock-markets [25]), the cost of a fault is typically much higher than that of a temporary loss of availability of resources.

The Ethereum community's preference for safety in this trade-off was recently exemplified when attackers exploited a bug in the *Parity Multisig Wallet* [3] to steal users' funds. A consortium of "white-hat hackers" used the same bug to move user's funds into a safe account. Despite funds being locked away for several weeks, and reimbursement being contingent on the consortium's good will, the action was acclaimed by the community and in particular by the victims of the white-hat "attack". Note that the "escape hatch" in this scenario (i.e., send all funds to a safe account), albeit crude, was deemed a successful alternative to an actual exploit.

We thus propose trading availability for safety in multiversion programming, by means of a wider spectrum of voting schemes. In effect, we transition from a goal of *fault-tolerance* to one of *error detection and safe termination*. Suppose that programs $f_1, \ldots, f_N$ have no fallback outputs (i.e., $F(f_i) = \emptyset$). Then majority voting yields a program f^* that also satisfies $F(f^*) = \emptyset$, but may not induce a large exploit gap. At the other end of the spectrum, we propose *N-of-N-version programming* (NNVP), wherein f^* aborts (i.e., outputs $\perp$) unless *all* of the N versions agree. NNVP is an *availability-preserving* transformation that induces a much larger *exploit gap* (f^* only fails if all the f_i fail simultaneously). The balance between availability and correctness offered by intermediate points in this design space (e.g., abort if more than k versions disagree for $1 \leq k \leq N/2$) is an interesting question to explore.

Table I lists prominent losses to Ethereum smart contract failures. We discuss these in more detail in Appendix A, and argue that a majority could have been abated with NNVP.

A. Revisiting N-version Programming

We revisit experiments challenging the cost-effectiveness of N-version programming, in light of our NNVP alternative.

In a famous experiment, Knight and Leveson [26] found that the null-hypothesis of *statistical independence* between program failures should be rejected. Yet partial dependence between program failures need not invalidate the multiversion paradigm, as long as the reduction in failure rates warrants the increased development costs.

However, in an experiment at NASA, Eckhardt et al. [12] found that the correlation between individual versions' faults could be too high to be considered cost-effective, with a majority vote between three programs reducing the probability of

Contract name	Exploit value (USD)	Root cause	Independence source	Exploit gap
Parity Multisig [3]	180M	Delegate call+unspecified modifier	programmer/language?	✔/✗
The DAO* [18]	150M	Re-entrancy	language	✔
SmartBillions [19]	500K	Bug in caching mechanism	programmer	✔
HackerGold (HKG)* [20]	400K	Typo in code	programmer+language	✔
MakerDAO* [21]	85K	Re-entrancy	language	✔
Rubixi [22]	<20K	Wrong constructor name	programmer+language	✔
Governmental [22]	10K	Exceeds gas limit	None?	✗

TABLE I: Selected smart contract failures and potential exploit gaps. The list is extended from [23]. For each incident, we report the value of affected funds (data from [24]) and identify the high-level cause of the exploited vulnerability, as well as the (hypothetical) potential for fault independence between multiple contract versions. Green lines indicate settings in which a Hydra contract is likely to have induced a large exploit gap and prevented the loss of funds. Yellow and red lines indicate incidents that our solution can address only partially or not at all. Asterisks indicate ERC20 compatible contracts, like our bounty described in Section VII. Details about each exploit and the potential for exploit gaps are in Appendix A.

some fault classes by a factor of only 4. For input sequences X sampled from a broad test suite, majority-voting over the programs they analyzed thus achieves $\texttt{gap} \approx 4$.

Under NNVP, the cost-benefit analysis is much more appealing. From the experimental results of Eckhard et al. [12] we find that three of their programs failed *simultaneously* with probability at least 30-5,000 times lower than a single program. Similarly, for four versions, the failure rate is reduced by a factor of at least 190-24,200. More details are in Appendix B. The actual gain is probably much larger, as Eckhardt et al. [12] do not distinguish whether program failures are *identical* or not. In NNVP, a failure only occurs if all N versions produce *exactly the same incorrect output*. Otherwise, our scheme would abort whenever a divergence in the N versions occurs. Thus, if loss of availability can be tolerated in rare situations, NNVP can significantly boost the error detection capabilities of multiversion programming.

B. NNVP-Friendly Properties of Smart Contracts

In addition to favoring safety over availability, other properties of smart contract ecosystems (and Ethereum in particular) render NNVP bug bounties attractive.

High risk for small applications. Smart contracts store large financial values in small applications (e.g., token transfer contracts), thus achieving a "price per line of code" that may be unparalleled in other software. Incentives for bug-mitigating strategies are high, as smart contract code is stored on a public blockchain and can often be executed by any party. Consequently, exploits can usually directly extract or destroy stored funds. The cost of developing multiple versions, however, is typically small in absolute terms.

Principled bounty pricing. A contract's balance can often provide a direct measure of an exploit's market value. This facilitates our analysis of principled bounty pricing that incentivizes early disclosure of bugs (see Section IV).

Bounty automation. The smart contract ecosystem enables automation of the full bounty program, from bug detection (with on-chain differential testing) to rollback to bounty payments. Bounties administered by smart contracts can satisfy many desirable properties such as *fair exchange* of bounties for bugs and *guaranteed payment* for successful bug discovery and disclosure [27]. Moreover, bounties are *transparent* to users (i.e., the bounty is publicly visible on the blockchain) and may be *dynamically* adjusted to reflect a contract's changing balance (and thus exploit value). The result is a stable, decentralized bounty marketplace.

Programming language diversity. Many exploits in Ethereum arose due to traits of specific programming languages. The simplicity of Ethereum's virtual machine has led to the development of multiple interoperable languages, thus enabling potentially diverse implementations.

A step towards formal verification. The development process underlying multiversion programming [16] can itself increase program correctness. In particular, developing multiple interoperable program versions requires a detailed specification of the program's behavior. Such a specification is only rarely available for current smart contracts, yet could pave the way to more systematic formal verification.

C. The Hydra Contract

The Hydra consists of two program transformations. The first transformation $\mathcal{T}_{NNVP}$ uses the NNVP paradigm to induce an exploit gap. $\mathcal{T}_{NNVP}$ combines N smart contracts (or heads) $f_1, \ldots, f_N$ into a single contract f^*, which delegates calls to each head on every input. If the N outputs match, f^* returns the output; otherwise, f^* rolls back any state changes and returns the fallback output $\perp$.

The second transformation $\mathcal{T}_{Bounty}$ is responsible for paying out a bounty and providing escape-hatch functionality. It transforms a program f^* into a program $\hat{f}$ which forwards any input to f^* and then returns f^*'s output, unless f^* returns $\perp$. In the latter case, $\hat{f}$ will pay out a bug bounty to its caller and enter an *escape hatch mode*.

Ideally, bugs could be patched *online*. Yet this is hard in systems such as Ethereum where a smart contract's code cannot be updated after deployment [28]. An advocated best practice [29] is thus to enhance smart contracts with an *escape hatch* mode, which enables the contract's funds to be retrieved, before it's eventual termination and redeployment.

The exact design of the escape hatch mode depends on the application, but there are some universal design criteria:

- *Security:* Since the escape hatch will not benefit from the protection afforded by NNVP, special care must be taken to ensure its correctness.
- *Availability:* If the escape hatch mechanism is unavailable, all assets held by the contract could end up stuck. The design of the escape hatch should ensure availability for the entire lifetime of the contract.
- *Distributed trust:* The contract's assets should be returned to their owners (if these can be safely established), or distributed among multiple parties.[1]

Simple designs are generally easier to analyze and less likely to be vulnerable or unavailable. In simple cases, where the last uncompromised state of the contract can be safely established, reverting to that state and allowing stakeholders to withdraw their assets is a satisfactory solution devoid of any trust assumptions. More generally, we suggest transferring the contract's funds to an external *multisig* contract. These standardized contracts hold many millions of dollars worth of cryptocurrency and have been heavily audited. By virtue of requiring multiple signatures to perform any action on the contract, trust is distributed among multiple parties. To further improve security, this escape-hatch multisig contract can itself be developed using the Hydra Framework. If a bug in the multisig contract is found (e.g., [3]) the Hydra multisig can resort to sending all funds to a trusted mediator.

IV. Economic Analysis of Hydra Bounties

We formally analyze the exploit gap introduced by the Hydra contract, and derive a pricing model for bounties that incentivize bug disclosure. We assume that when a bounty hunter discovers a bug in a head, she is awarded a bounty instantaneously. In Section V, we revisit and refine our analysis in the blockchain model, wherein the adversary may delay and reorder messages sent to smart contracts.

A. Bug Finding as a Stochastic Process

We consider a set of parties that try to find vulnerabilities in a Hydra contract f^* composed of N heads $f_1, \ldots, f_N$. Running an exploit on contract f^* may require multiple transactions (e.g., [2], [3]). For simplicity, we slightly overload notation and identify an exploit with the input that ultimately causes the contract's outputs to depart from the ideal behavior $\mathcal{I}$ (although the internal state of f^* may have been corrupted earlier). That is, an input x is an exploit if $\texttt{run}(f^*, X \sqcup [x]) \neq \texttt{run}(\mathcal{I}, X \sqcup [x])$, where X is the (implicit) sequence of all inputs previously submitted to f^*.

If an honest party finds an input x that yields an exploit for at least one of the heads ($\exists i \in [1, N] \ : \ x \in E(f_i, \mathcal{I})$), then the party is awarded a bounty of value $\$\text{bounty}$ and the contract's escape hatch is triggered. If a malicious party finds an exploit against the full Hydra (x is an exploit for each head), then we assume that the party can exploit this vulnerability to steal the full contract's balance, $\$\text{balance}$.

[1] An escape hatch that sends all funds to the contract's administrator (often the contract developer) opens up a perverse incentive for planting an obscure bug in one head that can later be triggered to deplete the contract. The same incentive also exists for non-Hydra contracts (see e.g., FirePonzi [22]).

We model bug finding as a Poisson process with rate λ_i, which captures a party's work rate towards finding program flaws. We assume that parties sample inputs x from a common distribution of potential exploits $\mathcal{D}$. In this context, we recover our exploit gap notion (Definition 1) by considering the difference in arrival times of two random events: (1) a party discovers a flaw in one of the heads; (2) a party finds a full exploit. The waiting times for both events are exponentially distributed with respective rates λ_i and

$$\begin{aligned}
&\lambda_i \cdot \Pr_{x \in \mathcal{D}}\left[x \in E(f^*, \mathcal{I}) \mid x \in \textstyle\bigcup_{i=1}^{N} E(f_i, \mathcal{I})\right] \\
=&\lambda_i \cdot \frac{\Pr_{x \in \mathcal{D}}\left[x \in E(f^*, \mathcal{I}) \ \wedge x \in \bigcup_{i=1}^{N} E(f_i, \mathcal{I})\right]}{\Pr_{x \in \mathcal{D}}\left[x \in \bigcup_{i=1}^{N} E(f_i, \mathcal{I})\right]} \\
=&\ \lambda_i \cdot \frac{\Pr_{x \in \mathcal{D}}\left[x \in E(f^*, \mathcal{I})\right]}{\Pr_{x \in \mathcal{D}}\left[x \in \bigcup_{i=1}^{N} E(f_i, \mathcal{I})\right]} = \lambda_i \cdot \texttt{gap}^{-1} \ , \qquad (2)
\end{aligned}$$

where we used the definition of $\texttt{gap}$ in Definition 1.

For simplicity, we first consider the strong assumption of independent program failures. For a head f_i, let p be the probability that an input x sampled from $\mathcal{D}$ is an exploit for f_i. Our analysis can easily include a distribution over a head's *vulnerability* p, as in [17]. Here, the gap is

$$\texttt{gap} = \frac{\Pr_{x \in \mathcal{D}}\left[x \in \bigcup_{i=1}^{N} E(f_i, \mathcal{I})\right]}{\Pr_{x \in \mathcal{D}}[x \in E(f^*, \mathcal{I})]} = \frac{1 - (1-p)^N}{p^N} \ , \qquad (3)$$

which grows exponentially in N, for $p \in (0, 1)$.

In general, the gap can be computed by plugging empirical estimates into Equation (1). For instance, from the results of Eckhardt et al. [12], we estimate a gap of $4{,}400$ for three heads and $34{,}500$ for four heads (details are in Appendix B). Note that these results are for inputs x sampled from the test suite used in [12]. A bug hunter may of course use a different distribution. In [12], the classes of inputs with highest failure rates (i.e., the inputs that a bug hunter would aim to sample) actually yield the largest exploit gaps.

B. Economic Incentives

We assume a set of *honest* parties with combined work rate λ_{H}. These bounty hunters only try to exchange bugs for bounties. Note that a bug that affects all heads (i.e., a full exploit) cannot be detected and rewarded by the meta-contract f^*. For simplicity, we thus let λ_{H} be the rate at which honest parties find bugs that affect $1 \leq k < N$ heads.

To analyze economic incentives of bounties, we consider malicious parties which, if given an exploit, would use it to deplete the contract's balance. W.l.o.g, we consider a single adversary $\mathcal{A}$ with work rate λ_{M}. Indeed, for m (non-colluding) malicious parties with work rates $\lambda_1, \lambda_2, \ldots, \lambda_m$, it suffices to analyze incentives for the party with rate $\lambda_{\text{M}} = \max_{1 \leq i \leq m} \lambda_i$. If the bounty incentivizes this party to act honestly, it is easy to see that less efficient parties will have the same incentive. The work rate of any party that decides to act honestly is incorporated into λ_{H}.

Let T_{H} be a random variable modeling the waiting time until some honest party finds a bug. T_{H} follows an exponential distribution with rate λ_{H}. Moreover, let T_{M} be the waiting

time until the malicious party finds an exploit against f^*. This variable is exponential with rate $\lambda_M \cdot \texttt{gap}^{-1}$. We analyze two cases: (1) $\mathcal{A}$ finds an exploit against f^*, and (2) $\mathcal{A}$ finds a bug that affects a proper subset of the heads.

In the first case, it is clear that $\mathcal{A}$ has no incentive to disclose, unless the bounty exceeds the contract's value. This is the situation of a "traditional" bounty scheme. However, the probability of this bad event occurring is

$$\Pr[T_M < T_H] = \frac{\lambda_M \cdot \texttt{gap}^{-1}}{\lambda_H + \lambda_M \cdot \texttt{gap}^{-1}} = \frac{\lambda_M}{\lambda_H \cdot \texttt{gap} + \lambda_M} ,$$

which naturally decays as the exploit gap increases.

The second case is the one where an appropriate bounty can incentivize honest behavior of $\mathcal{A}$. Suppose $\mathcal{A}$ found a non-exploitable bug. If $\mathcal{A}$ discloses the bug, she receives a payout of $\texttt{payout}_H := \$\text{bounty}$. If instead, she conceals the vulnerability and continues searching for exploits, she risks a payout of 0 if another party finds a bug and claims the bounty. Her expected payout, $\texttt{payout}_M$, is thus

$$\Pr[T_M < T_H] \cdot \$\text{balance} = \frac{\lambda_M}{\lambda_H \cdot \texttt{gap} + \lambda_M} \cdot \$\text{balance} .$$

Let $\alpha := \frac{\lambda_H}{\lambda_M}$. Then, honest behavior is incentivized if

$$\frac{\texttt{payout}_H}{\texttt{payout}_M} > 1 \iff \$\text{bounty} > \frac{1}{\alpha \cdot \texttt{gap} + 1} \cdot \$\text{balance} .$$

Under the (conservative) assumption that $\lambda_M = \lambda_H$ (the malicious party's work rate is equal to the *combined* work rate of all other parties), we get $\$\text{bounty} > \frac{1}{\texttt{gap}+1} \cdot \$\text{balance}$. Assuming independent program failures (see Equation (3)) the bounty decays exponentially in the number of heads N.

Thus, given estimates of α and of the exploit gap, our analysis provides a principled way of setting a bounty that incentivizes honest disclosure of discovered bugs. For instance, for the particular experiment conducted by Eckhardt et al. [12], a three headed Hydra could provide a bounty that is 3 to 4 orders of magnitude below an exploit's value.

V. Bounties on the Blockchain: The Bug Withholding Problem

Our economic analysis in the previous section assumes that a bounty is paid immediately upon a bug being claimed. However, when a bounty is run on a blockchain, adversaries can potentially exploit blockchain network protocols to cheat honest users. In this section, we refine our analysis by modeling bounty smart-contract execution with respect to a powerful blockchain adversary. We highlight in this model a problem called *bug withholding* and propose and analyze a solution called a *Submarine Commitment* in Section VI.

Front-running. The main challenge is that transactions need not be ordered in blocks according to the times they are sent to the network. When an honest user submits a bounty-claim transaction τ to the network, an adversary can potentially insert its own bounty-claim transaction τ' earlier into the block in which τ appears. It does this by ensuring faster network propagation of τ' or by causing a miner to order τ' before τ. (The adversary can pay a higher fee—more gas in Ethereum, for example—or corrupt the miner.) This problem is called *rushing* or *front-running* [13].

Front-running opens up a bug bounty system to *bug-withholding attacks*. Suppose an adversary has found a bug in one or more heads in a Hydra contract, and aims to find a stronger exploit compromising all heads. If another party in the meantime claims a bug bounty, the adversary's progress is wiped out: It loses all potential payoff on its already discovered bugs. By front-running, though, the adversary can *withhold the bug while trying to exploit the full contract*. If another player tries to claim a bounty, the adversary preemptively first claims its own bounty via front-running.

We propose a formal model for blockchain security. Our model, expressed as an ideal functionality $\mathcal{F}_{\mathsf{withhold}}$, encompasses front-running, but is far stronger and subsumes many previous models (e.g., Hawk [30]). We present a basic bug-bounty contract BountyContract in $\mathcal{F}_{\mathsf{withhold}}$. Refining our analysis of Section IV, we show how bug withholding in BountyContract breaks incentives for bug disclosure. We show that commit-reveal schemes cannot protect against bug withholding. Instead we introduce *Submarine Commitments*, a new technique for transaction concealment in Section VI. We prove within an $\mathcal{F}_{\mathsf{withhold}}$-hybrid world that *using Submarine Commitments for* BountyContract *drastically reduces the payoff of a bug-withholding adversary*.

A. Adversarial Model

We model an adversary $\mathcal{A}$ that can front-run a victim. In our model, $\mathcal{A}$ can mount strong *history-revision* attacks, overwriting blocks at the head of the blockchain, and can *delay* a victim's transactions by a bounded number of blocks.

These capabilities reflect an adversary's ability to monitor transactions in the network, mount network-level attacks against transaction propagation, control client accounts, and even corrupt or bribe miners to suppress or overwrite legitimately mined blocks. Previous models, e.g., [30], considered weaker attacks in which $\mathcal{A}$ can arbitrarily reorder transactions in any given epoch, i.e., within a pending block. They are equivalent to history-revision attacks with only a single block. Our model thus reflects a much stronger adversary.

In our model, $\mathcal{A}$ *itself constructs the blockchain*. $\mathcal{A}$ controls all but one honest player, denoted P_0. (P_0 models the collective behavior of all honest players.) $\mathcal{A}$ can affect the ordering of P_0's transactions by: (1) *Rewinding* the blockchain from its head, i.e., mounting a history-revision attack, for a sequence of up to ρ blocks; and (2) *Delaying* the posting on the blockchain of a transaction by P_0 by up to δ blocks. We call such an adversary $\mathcal{A}$ a (δ, ρ)-adversary.

Our adversarial model takes the form of an *ideal functionality* $\mathcal{F}_{\mathsf{withhold}}$ that characterizes an (δ, ρ)-adversary $\mathcal{A}$.

Preliminaries. Let $\mathcal{B} = \{B_1, B_2, \ldots, B_{\mathcal{B}.\mathsf{Height}}\}$ denote a *blockchain* consisting of a fully ordered sequence of *blocks*. Here, $\mathcal{B}.\mathsf{Height}$ denotes the number of blocks $\mathcal{B}$ contains. A block $B_i = \{\tau_{i,1}, \tau_{i,2}, \ldots, \tau_{i,s}\}$ is an ordered sequence of s transactions, i.e., B_i has blocksize s. (Smaller block sizes can be modeled via null transactions $\tau_{i,j} = \varnothing$.) For simplicity, we

assume no forks. In the case of a fork, $\mathcal{A}$ may operate on what it believes to be the authoritative chain.

Let $\mathcal{P} = \{P_0, P_1, \ldots, P_m\}$ denote a set of *clients* or *players* that execute transactions. We assume w.l.o.g. that P_0 is honest and the other m players are controlled by $\mathcal{A}$. We assume a system-wide transaction buffer Mempool, from which transactions are selected for mining.

The function $\mathsf{ValidTx}(\tau; \mathcal{B}, \mathsf{Mempool})$ verifies that a transaction is valid. It checks that τ respects the syntax and semantics of the blockchain. Additionally ValidTx verifies that the transaction carries a valid nonce. This nonce may be a counter value associated with the sender P_i that is incremented for every transaction posted by P_i. A transaction posted by P_i is valid if its nonce is larger than the nonces of any other transactions from P_i already in $\mathcal{B}$ or Mempool. Thus any transaction in a set of valid transactions is unique.

Ideal functionality $\mathcal{F}_{\mathsf{withhold}}$. The ideal functionality $\mathcal{F}_{\mathsf{withhold}}$, shown in Figure 2, supports three functions: "post", "add block", and "rewind". The function "post" permits any player (honest or adversarial) to send a transaction into the Mempool buffer. The function "add block" is called by $\mathcal{A}$ to extend the blockchain $\mathcal{B}$ by adding a new block that includes transactions from Mempool. The function "rewind" allows $\mathcal{A}$ to remove blocks from $\mathcal{B}$. This capability may seem redundant, as $\mathcal{A}$ controls the blocks added to $\mathcal{B}$. $\mathcal{A}$ may, however, wish to make retroactive modifications based on information in a fresh transaction submitted by P_0.

After adding a block, $\mathcal{A}$ must wait until P_0 has posted all its transactions, before $\mathcal{A}$ can add a new block. Execution of $\mathcal{F}_{\mathsf{withhold}}$ is bounded by a target height n, at which point $\mathcal{F}_{\mathsf{withhold}}$ halts and outputs $\mathcal{B}$. P_0 does not observe Mempool in our model, although variant models are possible of course.

B. BountyContract

Within $\mathcal{F}_{\mathsf{withhold}}$, we specify a contract BountyContract to administer a bounty for a single bug, using a simple *commit-reveal* scheme. BountyContract is parameterized by $\Delta > \delta + \rho$, as well as $deposit and $bounty. It takes as input a commitment to a bug in some block B_i (via transaction "commit"). The commitment must be revealed before block $B_{i+\Delta}$ (via transaction "reveal"). After a delay Δ, the player with the first validly revealed commitment may claim the bounty (via transaction "claim"). A "commit" incurs a cost of $deposit, to prevent $\mathcal{A}$ from committing in every block and revealing only if P_0 also reveals.

We assume a function isvalidbug (which may involve a call to another contract) that determines whether a submitted bug is valid. Within the $\mathcal{F}_{\mathsf{withhold}}$ model, BountyContract is fed a height-n blockchain $\mathcal{B}$, which is replayed, i.e., transactions are executed as ordered by $\mathcal{F}_{\mathsf{withhold}}$ in $\mathcal{B}$.

C. *Front-Running Attacks on* BountyContract

BountyContract uses a commit-reveal scheme, a simple and general folklore solution to certain rushing / front-running attacks [31]. This works if $\mathcal{A}$ *cannot post a valid commitment itself until it sees a victim's decommitment*. For instance,

$\mathcal{F}_{\mathsf{withhold}}$ with $\mathcal{P} = \{P_0, P_1, \ldots, P_m\}$, (δ, ρ)-adversary $\mathcal{A}$, blocksize s, target height n

Init: $\mathcal{B} \leftarrow \emptyset$, $\mathcal{B}$.Height $\leftarrow 0$, MaxHeight $\leftarrow 0$, Mempool $\leftarrow \emptyset$

On receive ("post", τ) from P_i: // P_i submits tx
- assert ValidTx(τ; $\mathcal{B}$, Mempool)
- tag(τ) $\leftarrow$ ($\mathcal{B}$.Height, P_i) // Label tx with current chain height and sender
- Mempool $\leftarrow$ Mempool $\cup$ τ
- send Mempool to $\mathcal{A}$

On receive ("add block", B) from $\mathcal{A}$: // $\mathcal{A}$ extends blockchain
- if $\mathcal{B}$.Height $= n$ then
  - output $\mathcal{B}$; halt // To complete chain, $\mathcal{A}$ adds arbitrary $n+1^{th}$ block
- assert $(|B| = s) \wedge (B \subseteq$ Mempool$)$
- assert $\nexists \tau \in$ Mempool $- B$ s.t. $(\mathsf{tag}(\tau) = (h, P_0)) \wedge (h \leq \mathcal{B}.\mathsf{Height} - \delta)$
  // Ensure delay at most δ for P_0's transactions
- $\mathcal{B}$.Height $\leftarrow \mathcal{B}$.Height $+ 1$
- $B_{\mathcal{B}.\mathsf{Height}} \leftarrow B$ // Add new block to chain
- Mempool $\leftarrow$ Mempool $- B$ // Remove processed txs from Mempool
- MaxHeight $\leftarrow \max(\mathcal{B}.\mathsf{Height}, \mathsf{MaxHeight})$
- send $\mathcal{B}$ to P_0

On receive ("rewind", r) from $\mathcal{A}$ // $\mathcal{A}$ rewinds by r blocks
- assert MaxHeight $- (\mathcal{B}.\mathsf{Height} - r) \leq \rho$
  // Ensure that $\mathcal{A}$ rewinds by no more than ρ
- Mempool $\leftarrow$ Mempool $\bigcup \{B_i\}_{i \in [\mathcal{B}.\mathsf{Height}-r+1, \mathcal{B}.\mathsf{Height}]}$
  // Return rewound transactions to Mempool
- $\mathcal{B}$.Height $\leftarrow \mathcal{B}$.Height $- r$

Fig. 2: Ideal functionality $\mathcal{F}_{\mathsf{withhold}}$ for (δ, ρ)-adversary $\mathcal{A}$

BountyContract with $\mathcal{B}$, $\mathcal{P} = \{P_0, P_1, \ldots, P_m\}$, Δ, $deposit, $bounty

Init: CommitList, RevealList $\leftarrow \emptyset$

On receive τ = ("commit", comm, $val) from P_i: // P_i commits to bug
- if $val $\geq$ $deposit then
  - CommitList.append(comm, $\mathcal{B}$.Height; P_i)

On receive τ = ("reveal", (comm, height), (witness, bug)) from P_i:
// P_i reveals commitment made in block height
- if (comm, height; P_i) $\in$ CommitList then
  - assert ($\mathcal{B}$.Height $-$ height) $\leq \Delta$
  - assert Decommit(comm; (witness, bug)) $=$ TRUE
  - assert IsValidBug(bug) $=$ TRUE
  - RevealList.append(height; P_i)

On receive τ = ("claim", height) from P_i: // P_i tries to claim bounty
- assert (height; P_i) $\in$ RevealList
- assert $\mathcal{B}$.Height $-$ height $> \Delta$
- assert $\nexists$(height$'$; $P_{i'}$) $\in$ RevealList s.t. height$'$ $<$ height
- send $bounty to P_i and halt // Pay bounty and ignore further messages

Fig. 3: Smart contract BountyContract

BountyContract prevents $\mathcal{A}$ from trying to learn and steal the committed bug from an honest player P_0.

Unfortunately, if Commit is a standard cryptographic commitment scheme, this approach does not protect against front-running in the $\mathcal{F}_{\mathsf{withhold}}$-hybrid model if $\mathcal{A}$ is withholding a bug it already knows. Here, $\mathcal{A}$ waits to see P_0 send a "commit". $\mathcal{A}$ then knows that someone is trying to claim a bounty, and can simply *front-run P_0's commitment* by posting her own "commit" ahead in the blockchain.

Players could in principle conceal true commitments by sending dummy commitments with random values $val $\geq$ *deposit*—so that they are indistinguishable from real commitments—but have a "dummy" flag that can be revealed to trigger a refund. This approach turns out to be complicated and unworkable, though. A community of users would not

in general have an incentive to generate dummy traffic and incur transaction fees. A would-be claimant could generate dummy traffic to conceal her true commitment, but then the very inception of dummy traffic would signal a pending claim and incentivize $\mathcal{A}$ to release its withheld bug.

This problem arises in many other scenarios, e.g., token sales or auctions, where a bidding user must send funds for her bid, thus exposing the bid amount on the blockchain.

In Section VI, we show how a new technique, called a *Submarine Commitment*, can address the bug-withholding problem. First, we show why such front-running is harmful.

D. Impact of Bug Withholding in BountyContract

In our analysis of Hydra bug bounties in Section IV-B, we assumed that if $\mathcal{A}$ conceals a bug, she might end up forfeiting a payout of $\$bounty$. However, a bug-withholding attack has the potential of removing any incentives for early disclosure, as $\mathcal{A}$ can *ensure* a payout of at least $\$bounty$ by front-running the honest bounty hunter.

If $\mathcal{A}$ conceals her bug, she finds an exploit before an honest party attempts to claim the bounty with probability $q := \Pr[T_{\mathrm{M}} < T_{\mathrm{H}}]$, and otherwise front-runs to claim the bounty. Her expected payout is

$$\texttt{payout}_{\mathrm{M}} = q \cdot \$balance + (1-q) \cdot \$bounty \ .$$

Conversely, if $\mathcal{A}$ discloses the bug, her payout is $\texttt{payout}_{\mathrm{H}} = \$bounty$. To incentivize honest behavior, we need $\texttt{payout}_{\mathrm{H}} > \texttt{payout}_{\mathrm{M}}$, i.e., $\$bounty > \$balance$, which again corresponds to a traditional bounty with no exploit gap. Fortunately, we now show an elegant solution that effectively thwarts bug-withholding attacks in Ethereum, thus re-instantiating positive incentives for bug disclosure.

VI. Thwarting Front-Running Attacks with Submarine Commitments

We present a bug-withholding defense called a *Submarine Commitment*. This is a powerful, general solution to the problem of front-running that may be of independent interest, as it can be applied to smart-contract-based auctions, exchange transactions, and other settings.

As the name suggests, a Submarine Commitment is a transaction whose existence is temporarily concealed, but can later be surfaced to a target smart contract. It may be viewed as a special, stronger form of a commit / reveal scheme. Achieving Submarine Commitments is challenging in systems like Ethereum, however, because message contents and currency in all transactions are *in the clear*.

Briefly, in Ethereum, to *commit* in a Submarine Commitment scheme, P posts a transaction τ that sends (nonrefundable) currency $\$val \geq \$deposit$ to an address $\widehat{\mathsf{addr}}$. This address is itself a commitment of the form

$$\widehat{\mathsf{addr}} = H(addr(\mathsf{Contract}), H(addr(P), \mathsf{key}), data) \ ,$$

for H a commitment scheme (e.g., hash function in the ROM), key a randomly selected witness (e.g., 256-bit string), and $data$ other ancillary information. P's address is included in the commitment to prevent replay by $\mathcal{A}$. To *reveal*, P sends key to Contract. A Submarine Commitment scheme includes an operation *DepositCollection* that permits Contract to recover $\$val$ using $addr(P)$ and key. This scheme has these key properties:

1) *Commit:* As key is randomly selected, $\widehat{\mathsf{addr}}$ is indistinguishable from random in the view of $\mathcal{A}$. Thus τ has no ascertainable connection to Contract, and looks to $\mathcal{A}$ like an *ordinary send to a fresh address*.
2) *Reveal:* After learning key, Contract can compute $\widehat{\mathsf{addr}}$ as above and verify that $\$val$ was sent correctly. Via *DepositCollection*, Contract recovers $\$val$ thus avoiding unnecessary burning of funds.

Thus if $\mathcal{A}$ does not know P's address (e.g., P can use a *mixer*), and $\$val$ is sampled from an appropriate distribution of values $\$val \geq \$deposit$, $\mathcal{A}$ cannot distinguish transaction τ from other sends to fresh addresses. As we show in Appendix C, such sends are common in Ethereum and, for a reasonable commit-reveal period (e.g., 25 minutes), form an *anonymity set* of hundreds of transactions with a diverse range of values among which $\$val$ is statistically hidden. Notably, the anonymity set represents 2-3% of *all* transaction traffic over the commit-reveal window.

Submarine Commitments via contract creation. A simple realization of Submarine Commitments in Ethereum leverages a new Ethereum Virtual Machine (EVM) opcode, `CREATE2`, introduced by EIP-86 (EIP stands for "Ethereum Improvement Proposal"). `CREATE2` creates new smart contracts, much like an already existing `CREATE` opcode. Unlike `CREATE`, which does not include a user-supplied value, `CREATE2` computes the address of the created contract C as $H(addrCreator, salt, codeC)$, where $addrCreator$ is the address of the contract's creator, $salt$ is a 256-bit salt value chosen by the creator, $codeC$ is the EVM byte code of C's initcode, and H is Ethereum-SHA3 (Keccak-256).

To realize a Submarine Commitment, we can use $salt$ to play the role of key in sending money $\$deposit$ to contract BountyContract. Let Forwarder be a contract that sends any money received at its address to BountyContract. A Submarine Commitment involves these functions:

- *Commit:* P selects a witness key $\leftarrow_{\$} \{0,1\}^{\ell}$ for suitable ℓ (e.g., $\ell = 256$). P sends $\$deposit$ to address

  $$\widehat{\mathsf{addr}} = H(addr(\mathsf{BountyContract}), H(addr(P), \mathsf{key}), code),$$

  where $addr(\mathsf{BountyContract})$ is BountyContract's address and $code$ is Forwarder's EVM initcode.
- *Reveal:* P sends key and commitBlk (the block number in which P committed) to BountyContract. BountyContract verifies that the commit indeed occurred in block commitBlk (see Appendix C-A for more details).
- *DepositCollection:* BountyContract creates an instance of Forwarder at address $\widehat{\mathsf{addr}}$ using `CREATE2`. A call to Forwarder sends $\$deposit$ to BountyContract.

EIP-86 will be included in Ethereum in the second stage of the Metropolis hard fork (tagged "Constantinople") [32]. More details are in Appendix C, including a less efficient alternative scheme that is realizable in Ethereum today.

Submarine Commitments in $\mathcal{F}_{\mathsf{withhold}}$. To model Submarine Commitments, we let players send a special message $(\text{“submarine-post”}, \tau)$ where $\tau = (\text{“commit”}, comm, \$val)$ to $\mathcal{F}_{\mathsf{withhold}}$. If τ is valid, $\mathcal{F}_{\mathsf{withhold}}$ adds τ to the end of block B, before B is added to the blockchain $\mathcal{B}$. Thus $\mathcal{B}$ can contain transactions that $\mathcal{A}$ does not see as it is constructing blocks. $\mathcal{A}$ cannot delay such messages but can still rewind $\mathcal{B}$ to evict them once the commitment is revealed.

A. *Submarine Commitments in* BountyContract

We prove that Submarine Commitments strongly mitigate bug withholding in BountyContract. Our analysis uses a game-based proof in the $\mathcal{F}_{\mathsf{withhold}}$-hybrid world.

Figure 4 shows our simple game, denoted by $\mathbf{Exp}^{\mathsf{bntyrace}}_{\mathcal{A}}$. The game is played between an honest user $P^* = P_0$, and a user P_1 controlled by $\mathcal{A}$. W.l.o.g., P^* models a collection of honest players, while P_1 models players controlled by $\mathcal{A}$. $\mathcal{A}$ interacts with P^* in the ideal functionality $\mathcal{F}_{\mathsf{withhold}}$. Let $\Delta > \delta + \rho$, where δ and ρ are the number of blocks by which $\mathcal{A}$ can delay or rewind in $\mathcal{F}_{\mathsf{withhold}}$. The experiment considers an interval of n blocks in a blockchain $\mathcal{B}$ of length $n' = n + \Delta$. The experiment's parameters are (n', δ, ρ, s) for $\mathcal{F}_{\mathsf{withhold}}$, and values Δ, \$deposit and \$bounty.

In this game, only two messages may validly be submitted by a player: (“commit”, \$deposit), and “reveal”. To model Submarine Commitments, we assume that P^*'s commitment message is opaque to $\mathcal{A}$, i.e., its presence in a block is not detectable by $\mathcal{A}$ and by implication does not count toward the size of the block in which it is included.

For clarity of exposition, we first analyze Submarine Commitments outside the Poisson framework from Section IV. Our results also hold in that setting, with a slightly tighter bound for our main Theorem 4, below (see Appendix D for a proof). Instead, we consider a blockchain interval of n blocks, wherein P^* commits in a block chosen uniformly at random. That is, P^* posts (“commit”, \$deposit), in the block at index $\mathsf{commblock}_{P^*} \leftarrow\!\!\$\, [1, n]$. P^* posts a “reveal” in block $\mathsf{revblock}_{P^*} = \mathsf{commblock}_{P^*} + \rho$.

$\mathcal{A}$ wins the game if it posts a valid “commit” message before P^* does, and also posts a valid corresponding “reveal” message. It then claims the bounty. We let

$$p_{\mathsf{wins}} = \mathsf{adv}^{\mathbf{Exp}^{\mathsf{bntyrace}}}_{\mathcal{A}} = \Pr\left[(\text{TRUE}, \cdot) \leftarrow \mathbf{Exp}^{\mathsf{bntyrace}}_{\mathcal{A}}\right].$$

As a first goal, an economically rational adversary $\mathcal{A}$'s aims to *maximize its expected payoff*, namely

$$\mathbb{E}[\$\text{payoff}] = p_{\mathsf{wins}} \cdot \$\text{bounty} - \mathbb{E}[\$\text{cost}]. \tag{4}$$

Of course, $\mathcal{A}$ can win with probability 1 by posting a “commit” message in B_1 followed by a valid “reveal” message within Δ blocks, in which case it achieves $p_{\mathsf{wins}} = 1$ with $\$\text{payoff} = \$\text{bounty} - \$\text{deposit}$, which is optimal.

But $\mathcal{A}$ has a second goal. Recall that $\mathcal{A}$ is a *bug-withholding adversary*. $\mathcal{A}$ may gain financial benefit outside the experiment $\mathbf{Exp}^{\mathsf{bntyrace}}_{\mathcal{A}}$ from delaying disclosure of its bug. So $\mathcal{A}$ would like to emit a “reveal” message as late as possible, so that it maximizes its withholding period.

Experiment $\mathbf{Exp}^{\mathsf{bntyrace}}_{\mathcal{A}}(n', \delta, \rho, s; \Delta, \$\text{deposit}, \$\text{bounty})$

Init: $n \leftarrow n' - \Delta, \$\text{cost} \leftarrow 0, \mathsf{commblock}_{P^*} \leftarrow\!\!\$\, [1, n]$

$\mathcal{A}^{\{\mathcal{B} \leftarrow \mathcal{F}_{\mathsf{withhold}}(\{P_0 = P^*, P_1\}, n, \delta, \rho, s)\}}$ //$\mathcal{A}$ interacts with $\mathcal{F}_{\mathsf{withhold}}$

for $i = 1$ **to** n

  if $(\text{“commit”}, \$\text{deposit}) \in B_i$ **then**

    $\$\text{cost} \leftarrow \$\text{cost} + \$\text{deposit}$ //Every commit costs \$deposit

if $\big(\exists (1 \le i \le \mathsf{commblock}_{P^*} \ \wedge\ i \le j \le \min(i + \Delta, n))$ s.t.

  $\exists\, (\tau = \text{“commit”}) \in B_i$ s.t. $\mathsf{tag}(\tau) = (i, P_1) \wedge$

  $\exists\, (\tau = \text{“reveal”}) \in B_j$ s.t. $\mathsf{tag}(\tau) = (j, P_1)\big)$ **then**

    output$(\text{TRUE}, \$\text{payoff} := \$\text{bounty} - \$\text{cost})$ //$\mathcal{A}$ wins

output$(\text{FALSE}, \$\text{payoff} := -\$\text{cost})$

Fig. 4: Adversarial game $\mathbf{Exp}^{\mathsf{bntyrace}}_{\mathcal{A}}$

One possible strategy for $\mathcal{A}$ is to reveal a bug only by *front-running* P^*. We call this a *pure front-running* strategy. Specifically, if P^* posts the message “reveal” in block B_j, then $\mathcal{A}$ learns that P^* posted a “commit” in block $B_{j-\rho}$. $\mathcal{A}$ can rewind and post its own “reveal” message earlier than P^*. But $\mathcal{A}$ can rewind at most ρ blocks (i.e., block $B_{j-\rho}$ cannot be erased), so $\mathcal{A}$ only succeeds if it has *previously* posted a “commit” in the interval $[B_{j-\rho-\Delta}, B_{j-\rho}]$.

We show that for natural parameter choices, a pure front-running strategy greatly reduces the payoff of $\mathcal{A}$. Intuitively, this is because front-running is quite expensive: Since $\mathcal{A}$ observes a “commit” message from P^* too late to remove it by rewinding, $\mathcal{A}$ must keep posting “commit” messages continuously to ensure that it can front-run P^*. We prove the following theorem in Appendix D.

Theorem 4. *Suppose that* $\Delta \ge 4$ *and* $\$\text{deposit} > \frac{10(\Delta+1)}{9n} \cdot \$\text{bounty}$. *Then a pure front-running adversary cannot achieve* $\mathbb{E}[\$\text{payoff}] \ge 0$.

This result is fairly tight and enables practical parameterizations of BountyContract, as this example shows.

Example 1. *Consider a bounty on the Ethereum blockchain, with 15-second block intervals. Suppose that* $\$\text{bounty} = 100{,}000$ *USD, that the period over which $\mathcal{A}$ competes with honest bounty hunters is one week, and that a commitment must be revealed in* $\Delta = 100$ *blocks. Then given* $\$\text{deposit} \ge 278$ *USD, a pure front-running adversary cannot achieve a positive expected payoff (i.e.,* $\mathbb{E}[\$\text{payoff}] > 0$*).*

Of course, $\mathcal{A}$ could adopt other strategies. Specifically, $\mathcal{A}$ could reveal its bug *preemptively*—i.e., before observing a “reveal” message from P^*. We therefore also consider a second, natural strategy that we call α-*revealing*.

An α-revealing adversary uses front-running in the interval $[B_1, B_{\alpha n}]$. If it has not yet revealed its bug, it does so in block $B_{\alpha n+1}$, for αn an integer. Thus, $\mathcal{A}$ ensures that it wins the bounty with probability at least $1 - \alpha$, while also potentially withholding for αn blocks. With $\alpha = 1$, the strategy is equivalent to pure front-running.

The proof of Theorem 4 immediately yields the result:

Corollary 5. *Suppose that* $\Delta \ge 4$ *and* $\$\text{deposit} \ge \frac{10(\Delta+1)}{9n} \cdot \$\text{bounty}$. *Then an α-revealing adversary $\mathcal{A}$ achieves* $\mathbb{E}[\$\text{payoff}] \le \big((1 - \alpha) \cdot \$\text{bounty}\big) - \$\text{deposit}$.

Of course, the space of strategies for an economically rational adversary $\mathcal{A}$ is a superset of α-withholding. $\mathcal{A}$ might use a probabilistic strategy, reveal preemptively at a time that depends on the set of blocks in which it has made commitments, etc. We conjecture that such an approach is no better than α-withholding. We leave proof of this claim, and thus general results about economically rational adversaries $\mathcal{A}$, as an open problem. Additionally, our analysis can be extended to model imperfectly concealed Submarine Commitments and non-uniform commitment times by P^*.

For instance, returning to the Poisson model in Section IV, we can identify $\mathsf{commblock}_{P^*}$ with the waiting time T_{H} until P^* finds a bug to commit, which has an exponential distribution of rate λ_{H}. We show in Appendix D that if $\mathcal{A}$ competes with P^* over a period of approximately $n = \lambda_{\mathrm{H}}^{-1}$ blocks (in which case we expect honest parties to find exactly one bug in this n-block interval) Theorem 4 still holds.

VII. Implementation and Evaluation

We now present our implementation of a decentralized automated bug bounty for Ethereum smart contracts. We describe the main technical challenges in deploying a Hydra contract on-chain, and explain our design choices. We applied the Hydra Framework to two applications: (1) a generic ERC20 contract [9] for token transfers, and (2) a generalized *Monty Hall Lottery*, wherein two participants play a multi-round betting game [10]. Details on the implementation of Submarine Commitments in Ethereum are in Appendix D.

A. EVM Preliminaries

The Ethereum Virtual Machine (EVM) is a simple stack-based architecture [33]. Smart contracts executing in the EVM get access to three data structures: (1) the stack; (2) volatile memory; and (3) permanent on-chain storage.

Execution of a contract begins with a *transaction* sent to the blockchain, specifying the called contract, the call arguments, and an amount of ether, Ethereum's default currency. The EVM executes the contract's code in a sequential single-threaded fashion. Operations can update stack items, read and write to memory or to storage, and spawn a new call frame (with a new empty memory region) by calling other contracts. Each instruction costs a fixed amount of *gas*, a special resource used to price transactions.

Contracts can exceptionally halt—reverting all changes made in the current call frame (e.g., storage updates, transfers of ether)—and report an exception to the callee.

B. An Execution Environment for the EVM

The main technical challenge in deploying our Hydra Framework on the Ethereum blockchain is the implementation of the N-version "Execution Environment" [15], [16], the agent that coordinates the N versions and combines their outputs. The Execution Environment's complexity should be minimal, as it constitutes a Trusted Computing Base (TCB) for our application: exploiting the coordinating software is likely to lead to an exploit against the Hydra contract.

To achieve the full power of our Hydra bounty program, N versions of a smart contract as well as the Execution Environment (the *meta-contract*) are run on the blockchain. Indeed, while we could run a traditional bounty program off-chain (to reward bugs for a single smart contract deployed on chain), this would not provide an affirmative exploit gap, a central property in our analysis of attacker incentives.

A simple proxy contract. Suppose we have N smart contract versions or *heads* $f_1, \ldots, f_N$. In principle, a Hydra meta-contract could take on a simple and generic design: On a call with input x, call each head on input x sequentially and record each head's output. If all outputs match, return that output. Otherwise, pay the bounty and invoke an escape hatch (e.g., reimburse all users and destroy the contract).

However, this design suffers from a major shortcoming: smart contracts can, in addition to changing their own long-term storage, interact with each other. For instance, a contract can send ether to another contract. A naïve sequential execution of the N heads would result in N duplicate sends.

Ordering reads and writes. To handle interactions between contracts, we could run each head until it reads or writes the state of another contract (e.g., reading a contract's balance, or sending ether). The meta-contract then checks that all heads agree, issues the read/write operation, and resumes the heads. The contract's specification determines these *cross-check points* [15], [16], by defining a *total-ordering* of the reads and writes issued on each call.

We have implemented a generic Hydra *instrumenter* that converts each head's global reads and writes into callbacks to the meta-contract, which issues the calls on the heads' behalf. As handling arbitrary read/write sequences introduces some technicalities unrelated to the main contributions of this paper, we leave a detailed description of this generic solution to a forthcoming manuscript.

Hereafter, we describe a simplified design, sufficient for the applications we target, that slightly limits the type of reads and writes a contract can issue:

- Heads can issue arbitrary sequences of *reads* of the blockchain state (e.g., read contract balances).
- The only type of write operation permitted is *sending* ether to other contracts.
- All sends must occur as the last actions in a call (this is an example of the "Checks-Effects-Interactions" pattern described in Solidity's Security Guidelines [29]).

Given these assumptions, we ask that each head returns, in addition to its output, a list of sends. If the outputs and sends returned by all heads match, the meta-contract executes the sends and returns the output. Otherwise, it pays a bounty and invokes an escape hatch.

Sending ether is by far the most common interaction between Ethereum contracts. Others, omitted here for simplicity, include creating child contracts, self-destructing a contract, or calling arbitrary functions in other contracts.

Handling exceptions. Finally, we consider exception-handling in our meta-contract. Recall that the EVM halts when contracts perform illegal operations, such as explicitly

throwing exceptions, underflowing the stack, or running out of gas. Ideally, we would consider any difference in the heads' throw behavior as a bug and pay a bounty. However, it is easy to set gas amounts so that one head runs out of gas, yet others succeed. This issue is fixed in the recent *Byzantium* hard-fork [34]. Thrown exceptions can now return data, allowing the meta-contract to distinguish explicit exceptions thrown by heads from stack or gas exceptions.

C. Applications

To demonstrate our approach empirically, we developed three independent copies of two applications, a simple token transfer contract implementing the ERC20 token interface [9], and a generalized version of a Monty Hall game.

For each application, three authors developed one head in each of *Solidity*, *Serpent*, and *Viper*, the main programming languages in Ethereum. Solidity is the most popular language, focusing on ease-of-use and general applicability. Serpent is a lower-level language allowing direct access to EVM opcodes. Viper is an experimental language with a focus on security and decidability. Although we do not measure this quantitatively, the languages themselves seem to be a source of strong diversity between our Hydra heads.

The Hydra ERC20 token. We deployed a Hydra token transfer contract on the main Ethereum network. The standard ERC20 API has been thoroughly peer reviewed [9], and is supported by most of the highest-dollar contracts in Ethereum (as of October 2017, the combined market cap of the top ten Ethereum tokens is over 2.5 billion USD [24]). Notably, the exploit in the DAO [2] was partially present in the code managing DAO tokens, or shares.

Our token is implemented by a meta-contract that dispatches calls to the three heads and rewards a bounty for inputs that trigger a discrepancy in the heads' output behavior. This contract can be used as a drop-in replacement for any ERC20 token, including the tokens used in the DAO [2] and ether.camp [20] contracts. Note that the heads themselves do not hold any ether, and simply implement the token's bookkeeping logic. When a user wishes to deposit, transfer or withdraw tokens, the meta-contract examines the heads' outputs and executes the order if agreement is reached. Our current bounty on divergence in the heads is 1,000 USD, which we plan to increase as the contract undergoes the security audit, review, and testing process.

A Hydra Monty Hall game. We further evaluate a more complex Hydra contract, for a Monty Hall game. One party, the *house*, first hides a reward behind one of n doors. The *player* bets on which door holds the reward, and the house opens k other non-winning doors. Then the player may change his guess. If the guess is correct, the player obtains the reward, otherwise the house collects the bet.

A fourth author independently wrote a specification describing the contract's API and expected behavior. The house's initial door choice takes the form of a cryptographic commitment that is later opened to reveal the winner. If either party aborts, the other party can claim both the reward and bet after a fixed timeout. The specification leaves the internal representation of the game open to developers. We will release our specification, along with the code for the heads and meta-contract prior to publication of this paper.

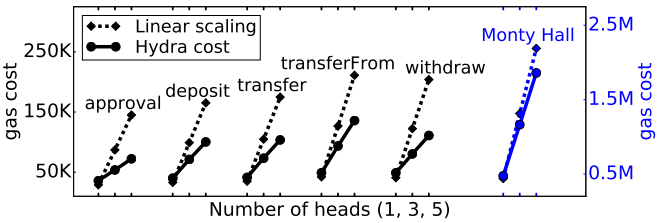

Fig. 5: Gas cost of Hydra contracts with N heads. We compare the Hydra contract to a linear scaling of a single contract for the ERC20 API (left) and a Monty Hall game (right).

D. Evaluation

This paper's goal is not to rigorously measure correlations between faults of developed smart contracts, but rather to propose a novel principled bug bounty framework built upon an assumed *exploit gap*. We leave a thorough analysis of failure patterns of multiversion smart contracts to future work. We evaluate our framework under standard software metrics, such as TCB size and performance overhead. We conclude with a discussion of our development process, and of some of the bugs we encountered (and fixed) when writing and testing multiple heads.

Size and complexity of the TCB. The meta-contract design described in Section VII-B is generic, and easily handles both of our target applications. The generic meta-contract is implemented in 130 lines of Solidity code, to which we add an application specific API (20-40 lines). As the main code is application-agnostic, and of a very simple nature, we believe this is a reasonable TCB. Given the simplicity of the functionality implemented in the meta-contract, it should also be relatively straightforward to write a formal specification for it, although we have not attempted this.

Gas costs. A concern when running N copies of a smart contract is a transaction's gas overhead. We note that some projects in Ethereum, notably the Viper language, already trade gas efficiency for security (e.g., by adding runtime checks to contracts). Moreover, Ethereum provides simple mechanisms to offload a transaction's gas cost onto the contract *owner*, thus dispensing users from the gas overhead incurred by Hydra. In any event, for small yet common workloads, most of the gas cost of a transaction is taken up by a fixed "base fee". As the meta-contract calls all the heads in a single transaction, this fee is amortized, leading to *sub-linear* scaling of the gas-cost for N-headed Hydras.

Figure 5 compares gas-costs for Hydra contracts with 2-4 heads, compared to a linear scaling of a single non-Hydra contract. We show results for the five non-static calls in the ERC20 API, and for a full Monty Hall game (five transactions). For the ERC20 contract, a transaction's main cost is the Ethereum transaction base fee of 21,000 gas. A call to the meta-contract incurs an overhead of about 8,000 gas (mostly independent of the number of heads) which

corresponds to 0.0024 USD[2]. Completing a game of Monty Hall requires long-term storage of many game parameters which overshadows the base fee costs (each stored word costs $20{,}000$ gas). As each head stores the data independently, the scaling is close to (but still below) linear in this case.

Evaluation of the gas costs for two variants of Submarine Commitments are in Appendix C. Note that these costs are only incurred upon a successful bounty claim and do not affect "normal" transactions.

Observations from the development process. As the goal of this project was not to rigorously measure independence between faults of developed smart contracts, we took some liberties with the N-version programming process [15], [16]. Each developer implemented a first version of their head along with a unit-test suite. After that, we iteratively refined our heads, the test suites and the meta-contract to correct for inconsistencies and discovered bugs.

This multi-phase development uncovered various bugs in each developer's heads, *none of which* impacted all heads simultaneously! Examples include a misunderstanding of an ERC20 API call, incorrect treatment of integer overflows, an "off-by-one" error in validating the inputs to a Monty Hall game, and a vulnerability to an only recently discovered EVM anti-pattern that allows contracts to silently send ether via the `SELFDESTRUCT` opcode.[3] Notably, all these bugs could have been exploited for some gain individually, yet none of them appear useful against all three heads simultaneously.

In addition to the exploit gap induced by the Hydra contract, the development process itself contributed to increasing the quality of our contracts. For the Monty Hall, ensuring compatibility between heads required writing a detailed specification, which revealed several blind spots in our original design. Moreover, we found that writing a *differential testing suite* [35] (generating inputs at random and verifying agreement between the heads) was remarkably simpler for exercising many different code paths in the Monty Hall contract than with a traditional test suite.

VIII. Related Work

Software assurance and fault-tolerance are well-studied topics supported by an extensive literature on programming languages, formal verification, static and dynamic analysis, and other techniques. N-version programming [15], [16], [17] in particular was first explored decades ago and challenged in early influential studies [12], [11] discussed in Section II.

Smart contracts [36] and script-enhanced cryptocurrency [37], first proposed in the 1990s, have recently gained popularity thanks largely to the inclusion of a limited scripting language in Bitcoin and, more importantly, to the advent of Ethereum [8]. Research on smart contract security is in its infancy but topically extremely varied and includes: Descriptions of common contract bugs [38], [39], [40], static analysis and language enhancements for Solidity [39], formal verification tools [41], [42], design of "escape hatches" [28], resistance to DoS-like attacks against miners [43], techniques for data integrity [44], and a formal semantics for the EVM [23]. While promising, none of these tools and techniques have yet seen mainstream adoption, and they do not relate directly to our explorations in this paper.

Perhaps most closely related to our work is that of Tramèr et al. [27], who explore the use of smart contracts for bug bounties (using trusted hardware), but not the converse, i.e., bug bounties for smart contracts.

Bug withholding is reminiscent of "selfish-mining" [45], wherein a miner withholds and selectively releases blocks to nullify other miners' work, thereby amplifying her own mining power. As selfish mining operates at the block level and bug-withholding at the application level, the two attacks differ in their mechanisms, analysis, and implications.

Submarine commitments conceal bounty-related transactions using ordinary ones. This form of *cover traffic* reveals a conceptual connection to a variety of technologies, including anonymity networks such as Tor [46], network-based covert channels [47], and steganography and watermarking [48]. Submarine commitments differ from these techniques in that they assume ultimate decommitment of a hidden value and in their reliance on Ethereum-specific techniques.

Several works [30], [31], [27] model blockchain-level adversaries and their impact on smart contracts. They consider an adversary that can mount rushing or front-running attacks within a given block, however, and not the much stronger model of block rewriting we explore here.

IX. Conclusion

We have presented the Hydra Framework, the first principled approach to modeling and administering bug bounties that incentivize honest disclosure. The framework relies on a novel notion of an exploit gap, a program transformation that enables runtime detection of critical bugs. We have described one such strategy, N-of-N-version programming (NNVP), a variant of N-version programming that detects behavioral divergences between multiple program instances.

We have applied the Hydra Framework to smart contracts, highly valuable and vulnerable programs that are particularly well suited for fair and automated bug bounties. We have analyzed high profile smart contract compromises in Ethereum, totaling over $500M in losses, and argued that Hydra contracts could have prevented a majority of them.

We have formally shown that Hydra contracts incentivize bug disclosure by rational hackers, for bounties orders of magnitude lower than an exploit's value. We have modeled strong bug-withholding adversaries that could threaten the viability of on-chain bounties, and analyzed Submarine Commitments, a countermeasure of independent interest that conceals transactions among a pool of benign traffic.

[2] As of October 2017, 1 ether is worth roughly 300 USD and a gas price of 1 gwei ($= 10^9$ wei) is standard according to https://ethgasstation.info. A value of 1 ether corresponds to 10^{18} wei.

[3] This bug is particularly interesting: Two of our ERC20 heads explicitly stored the token balance in long-term storage, while the third head used the contract *balance* to reflect the number of tokens in circulation. The developers noted this difference in logic, but concluded that the two approaches should be equivalent. Yet, it was later demonstrated that the EVM actually allows for a contract to *silently* send ether to another when it is destroyed. The invariant that a token contract's balance reflects the tokens in circulation is thus false, yielding a bug in the third head.

Finally, we have described and evaluated an implementation of our Hydra Framework for two Ethereum applications, an ERC20 token contract and a Monty Hall betting game. We have launched our bounty-backed ERC20 Hydra token on the Ethereum main network, the first example of a principled and trust-free bug bounty offering. We hope that similarly rigorous bounties can bolster smart contract security, and be applied more broadly to high-risk applications.

ACKNOWLEDGEMENTS

We thank Paul Grubbs and Rahul Chatterjee for comments and feedback. This research was supported by NSF CNS-1330599, CNS-1514163, CNS-1564102, and CNS-1704615, ARL W911NF-16-1-0145, and IC3 Industry Partners. This material is also based upon work supported by the National Science Foundation Graduate Research Fellowship under Grant No. (NSF grant number). Lorenz Breidenbach was supported by the *ETH Studio New York* scholarship.

REFERENCES

[1] L. Ablon, M. C. Libicki, and A. A. Golay, *Markets for cybercrime tools and stolen data: Hackers' bazaar*. Rand Corporation, 2014.

[2] V. Buterin. (2016, Jul.) Hard fork completed. [Online]. Available: https://blog.ethereum.org/2016/07/20/hard-fork-completed/

[3] L. Breidenbach, P. Daian, A. Juels, and E. G. Sirer. (2017, Jul.) An in-depth look at the Parity multisig bug. [Online]. Available: http://hackingdistributed.com/2017/07/22/deep-dive-parity-bug/

[4] C. Miller. (2017, Jul.) Apple's bug bounty program faltering due to low payouts to researchers, new report claims. [Online]. Available: https://9to5mac.com/2017/07/06/apple-bug-bounty-program-payouts

[5] High-Tech Bridge SA. (2013, Sep.) What's your email security worth? 12 dollars and 50 cents according to Yahoo. [Online]. Available: https://www.htbridge.com/news/what_s_your_email_security_worth_12_dollars_and_50_cents_according_to_yahoo.html

[6] I. Arghire. (2016, Oct.) Researchers claim Wickr patched flaws but didn't pay rewards. [Online]. Available: http://www.securityweek.com/researchers-claim-wickr-patched-flaws-didnt-pay-rewards

[7] L. Vaas. (2013, May) Paypal refuses to pay bug-finding teen. [Online]. Available: https://nakedsecurity.sophos.com/2013/05/29/paypal-refuses-to-pay-bug-finding-teen/

[8] V. Buterin, "Ethereum: A next-generation smart contract and decentralized application platform," 2014, https://github.com/ethereum/wiki/wiki/White-Paper.

[9] F. Vogelsteller and V. Buterin. (2015, Nov.) ERC-20 token standard. Ethereum Improvement Proposal. Revision 405c369. [Online]. Available: https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20-token-standard.md

[10] Wikipedia. Monty Hall problem. Accessed Oct. 30, 2017. [Online]. Available: https://en.wikipedia.org/wiki/Monty_Hall_problem

[11] J. C. Knight and N. G. Leveson, "An experimental evaluation of the assumption of independence in multiversion programming," *IEEE Transactions on software engineering*, no. 1, pp. 96–109, 1986.

[12] D. E. Eckhardt, A. K. Caglayan, J. C. Knight, L. D. Lee, D. F. McAllister, M. A. Vouk, and J. P. J. Kelly, "An experimental evaluation of software redundancy as a strategy for improving reliability," *IEEE TSE*, vol. 17, no. 7, pp. 692–702, 1991.

[13] M. H. Swende. (2017, Jul.) Blockchain frontrunning. [Online]. Available: http://www.swende.se/blog/Frontrunning.html

[14] B. Randell, "System structure for software fault tolerance," *IEEE TSE*, no. 2, pp. 220–232, 1975.

[15] L. Chen and A. Avižienis, "N-version programming: A fault-tolerance approach to reliability of software operation," in *Fault-Tolerant Computing*. IEEE, 1995, p. 113.

[16] A. Avižienis, "The methodology of N-version programming," in *Software Fault Tolerance*, M. R. Lyu, Ed. John Wiley & Sons Ltd, 1995.

[17] D. E. Eckhardt and L. D. Lee, "A theoretical basis for the analysis of multiversion software subject to coincident errors," *IEEE TSE*, no. 12, pp. 1511–1517, 1985.

[18] P. Daian. (2016, Jun.) Analysis of the DAO exploit. [Online]. Available: http://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/

[19] J. Solana. (2017, Oct.) $500K hack challenge backfires on blockchain lottery SmartBillions. [Online]. Available: https://calvinayre.com/2017/10/13/bitcoin/500k-hack-challenge-backfires-blockchain-lottery-smartbillions/

[20] J. Manning. (2017, Jan.) Ether.Camp's HKG token has a bug and needs to be reissued. [Online]. Available: https://www.ethnews.com/ethercamps-hkg-token-has-a-bug-and-needs-to-be-reissued

[21] Reddit user "jupiter0". (2016, Jun.) From the MAKER DAO slack: "today we discovered a vulnerability in the ETH token wrapper which would let anyone drain it.". [Online]. Available: https://www.reddit.com/r/ethereum/comments/4nmohu/

[22] V. Buterin. (2016, Jun.) Thinking about smart contract security. [Online]. Available: https://blog.ethereum.org/2016/06/19/thinking-smart-contract-security/

[23] E. Hildenbrandt, M. Saxena, X. Zhu, N. Rodrigues, P. Daian, D. Guth, and G. Rosu, "KEVM: A complete semantics of the Ethereum Virtual Machine," 2017.

[24] Cryptocurrency market capitalizations. Accessed Oct. 30, 2017. [Online]. Available: https://coinmarketcap.com/tokens/

[25] S. Ro. (2014, Mar.) 29 instances of a major world stock market shutdown. [Online]. Available: http://www.businessinsider.com/history-of-world-stock-market-breaks-2014-3

[26] J. C. Knight and N. G. Leveson, "A reply to the criticisms of the Knight & Leveson experiment," *ACM SEN*, vol. 15, no. 1, pp. 24–35, 1990.

[27] F. Tramèr, F. Zhang, H. Lin, J.-P. Hubaux, A. Juels, and E. Shi, "Sealed-glass proofs: Using transparent enclaves to prove and sell knowledge," in *IEEE EuroS&P*, 2017, pp. 19–34.

[28] B. Marino and A. Juels, "Setting standards for altering and undoing smart contracts," in *RuleML*. Springer, 2016, pp. 151–166.

[29] Ethereum. Security considerations. Solidity documentation. Revision dc154b4e. [Online]. Available: http://solidity.readthedocs.io/en/develop/security-considerations.html

[30] A. Kosba, A. Miller, E. Shi, Z. Wen, and C. Papamanthou, "Hawk: The blockchain model of cryptography and privacy-preserving smart contracts," in *IEEE S&P*. IEEE, 2016, pp. 839–858.

[31] A. Juels, A. Kosba, and E. Shi, "The Ring of Gyges: Investigating the future of criminal smart contracts," in *ACM CCS*. ACM, 2016, pp. 283–295.

[32] R. R. O'Leary. (2017, Sep.) Metropolis today: The shifting plans for Ethereum's next big upgrade. [Online]. Available: https://www.coindesk.com/metropolis-today-shifting-plans-ethereums-next-big-upgrade/

[33] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," 2014. [Online]. Available: http://yellowpaper.io/

[34] Ethereum Team. (2017, Oct.) Byzantium HF announcement. [Online]. Available: https://blog.ethereum.org/2017/10/12/byzantium-hf-announcement/

[35] W. M. McKeeman, "Differential testing for software," *Digital Technical Journal*, vol. 10, no. 1, pp. 100–107, 1998.

[36] N. Szabo, "Formalizing and securing relationships on public networks," *First Monday*, vol. 2, no. 9, 1997.

[37] M. Jakobsson and A. Juels, "X-cash: Executable digital cash," in *Financial Cryptography*. Springer, 1998, pp. 16–27.

[38] K. Delmolino, M. Arnett, A. Kosba, A. Miller, and E. Shi, "Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab," in *Financial Cryptography*. Springer, 2016, pp. 79–94.

[39] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *ACM CCS*. ACM, 2016, pp. 254–269.

[40] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on Ethereum smart contracts (SoK)," in *International Conference on Principles of Security and Trust*. Springer, 2017, pp. 164–186.

[41] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, A. Gollamudi, G. Gonthier, N. Kobeissi, N. Kulatova, A. Rastogi, T. Sibut-Pinote, N. Swamy *et al.*, "Formal verification of smart contracts: Short paper," in *ACM PLAS*. ACM, 2016, pp. 91–96.

[42] Y. Hirai, "Formal verification of Deed contract in Ethereum name service," 2016. [Online]. Available: https://yoichihirai.com/deed.pdf

[43] L. Luu, J. Teutsch, R. Kulkarni, and P. Saxena, "Demystifying incentives in the consensus computer," in *ACM CCS*. ACM, 2015, pp. 706–719.

[44] F. Zhang, E. Cecchetti, K. Croman, A. Juels, and E. Shi, "Town Crier: An authenticated data feed for smart contracts," in *ACM CCS*. ACM, 2016, pp. 270–282.

[45] I. Eyal and E. G. Sirer, "Majority is not enough: Bitcoin mining is vulnerable," in *Financial Cryptography*. Springer, 2014, pp. 436–454.
[46] R. Dingledine, N. Mathewson, and P. Syverson, "Tor: The second-generation onion router," Naval Research Lab Washington DC, Tech. Rep., 2004.
[47] S. J. Murdoch and S. Lewis, "Embedding covert channels into TCP/IP," in *Information hiding*, vol. 3727. Springer, 2005, pp. 247–261.
[48] S. Katzenbeisser and F. Petitcolas, *Information hiding techniques for steganography and digital watermarking*. Artech house, 2000.
[49] J. Krug. (2016, Jun.) A Serpent send exploit. [Online]. Available: http://www.joeykrug.com/home/a-serpent-send-exploit
[50] L. Luu. (2017, Jul.) PeaceRelay: Connecting the many Ethereum blockchains. [Online]. Available: https://medium.com/@loiluu/22605c300ad3

APPENDIX A
BRIEF ANALYSIS OF PREVIOUS EXPLOITS

We briefly justify why various smart contracts exploits in Table I may have benefited from an exploit gap introduced by NNVP. Obviously, we cannot make definite claims that NNVP would have averted a loss. Instead, we give some informal arguments on why a NNVP setup would have likely introduced independence in many cases.

Typos and trivial errors. Some exploits in our analysis are due to trivial programmer errors. For instance in HKG a developer mistakenly used an `=+` expression rather than the correct `+=`, resulting in bad variable initialization. It is unlikely that this exact mistake would be repeated across the contracts of several developers. Moreover, `x =+ y` is not valid code in Viper or Serpent. It is thus impossible that this mistake would have persisted in a multi-language contract.

The mistake in Rubixi [22] resulted from a code refactoring that renamed a class but not the corresponding constructor. It is similarly unlikely that independent developers would have misnamed the constructor to the **exact same wrong name** (as any other inconsistent and incorrect naming would have triggered a recovery and bounty). Also, some languages like Viper have a fixed constructor name (i.e., `__init__`), so this bug has no analogue in that language.

Note that for such trivial errors, Hydra contracts are likely not *required*, and thorough testing should have exposed the flaws. Nevertheless, Hydra's principled development process would certainly have prevented these losses.

Re-entrancy. Re-entrancy is a flaw described in [18], whereby a victim contract calls an external untrusted contract, allowing the called contract to call back into the victim and effect state changes in the middle of the original call.

There are several aggravating factors that lead to a string of re-entrancy vulnerabilities, including the The DAO [18] and MakerDAO [21]. For one, Solidity encouraged the use of the `call.value` construct in sending funds to accounts, to prevent "out of gas" errors. By forwarding all the gas by default, Solidity contracts essentially gave these untrusted contracts infinite fuel to execute their attack.

Unlike in Solidity, in Serpent, the use of the `send` function was recommended, which did not provide enough gas to re-enter into the original contract. The difference between Solidity and Serpent (these analyses pre-dated Viper) with regards to re-entrancy is detailed in [49]. As pointed out in this analysis, not all recursive send issues are mitigated by Serpent, but recursive sends with a non-zero valued call (as in Maker and the DAO) are impossible.

Complex programmer errors. Some errors are more complex; for example, the Parity hack [3] involved a bad unforeseen interaction between a "delegate call"—allowing library code to be run in the trusted context of a victim contract—and missing modifiers and guards on the library contract to prevent misuse. Although different contracts may have similarly missed this vulnerability, the complex nature of the bug suggests that the failure patterns would not have been identical. It is possible of course that these broken design decisions would have been formalized in the specification, annulling the likelihood of an exploit gap.

The analysis of [19] is similar, with a complex mistake in a blockhash caching system's code by the developer. This caching system was required by a limitation of the Ethereum platform on retrieving old blockhashes. Again, it is not clear that this complex cache would have been implemented correctly by a second developer, yet it is unlikely that both versions would fail in exactly the same fashion.

Other out-of-scope exploits. Not all exploits can be covered by a Hydra contract. One example in Table I is Governmental [22], which had a function that required more gas than was allowed to be used on the network, resulting in a denial-of-service vulnerability. Gas errors are explicitly ignored in our framework (see Section VII-B), as they can be triggered at any time by users simply refusing to provide enough gas.

Another example is FirePonzi [22], in which a variable was intentionally misnamed by the developers to serve as a backdoor. Such subtle backdoors-by-construction are still possible in the Hydra framework, and may actually be made even more subtle: for example, minor disagreements between heads can be made to trigger recovery with plausible deniability, potentially stealing funds if the recovery process is trusted or vulnerable to its own hidden exploits.

Lastly, errors in the contract specification—e.g., the flawed rock paper scissors game in [22]—are not covered by our framework as the specification is common to all heads.

APPENDIX B
ANALYSIS OF NNVP IN THE NASA EXPERIMENT

We briefly justify the results we obtained when applying our NNVP paradigm for the experimental results in [12]. The experiment consisted of 20 different program versions evaluated on six work-loads (corresponding to different initial system states). For $y \in [0, 20]$, Eckhardt et al. report $g(y)$, the empirical proportion of inputs in each of their test suites that induce a failure in exactly y out of 20 programs. They do not distinguish whether the failures are identical or not.

Following the notation and analysis for majority-voting in [12], we compute the empirical probability $\tilde{P}_N$ that N programs (randomly chosen from the 20) fail simultaneously:

$$\tilde{P}_N = \binom{20}{N}^{-1} \sum_{y=0}^{20} \binom{y}{N} g(y) \ . \tag{5}$$

When comparing $\tilde{P}_N$ to $\tilde{P}_1$ (i.e., the expected failure rate for NNVP with N heads compared to the use of a single

program), we find that $30 \cdot \tilde{P}_3 \leq \tilde{P}_1 \leq 5{,}087 \cdot \tilde{P}_3$ and $190 \cdot \tilde{P}_4 \leq \tilde{P}_1 \leq 24{,}216 \cdot \tilde{P}_4$. In both cases, the lowest exploit gap is obtained for the third work-load (denoted $S_{1,0}$), for which the failure rate $\tilde{P}_1$ of a single program is the lowest.

If we combine all work-loads into one, and assume that hackers sample uniformly from the test inputs used in the experiment, then the exploit gap, `gap`, defined in Section IV is estimated as the ratio of $\tilde{P}_1$ and $\tilde{P}_N$ (averaged over the six workloads). We obtain `gap` $= 4{,}409$ for $N = 3$ and `gap` $= 34{,}546$ for $N = 4$. Note that we can also apply NNVP with $N = 2$ (whereas majority-voting obviously does not work in this case), and find a gap of `gap` $= 79$.

APPENDIX C
SUBMARINE COMMITMENT CONSTRUCTIONS

A. Merkle-Patricia Proof Verification

In order for Submarine Commitments to be secure against front-running attacks, we need to verify that the commit transaction indeed occurred in block commitBlk. Otherwise, an adversary can wait until she observes the "reveal" transaction τ. Upon observing τ, she can front-run it by including a backdated "commit" transaction and a corresponding "reveal" message in front of τ. We can prevent this attack by having Contract verify that "commit" was indeed sent in block commitBlk and that at least ρ blocks have elapsed since commitBlk upon receiving a "reveal". (Recall that the adversary can roll back the blockchain by at most ρ blocks.)

Unfortunately, Ethereum provides no native capability for smart contracts to verify that a transaction occurred in a specific block. However, Ethereum's block structure enables efficient verification of Merkle-Patricia proofs of (non-)inclusion of a given transaction in a block [50]: all transactions in a block are organized in a Merkle-Patricia Tree [33] mapping transaction indices to transaction data. The root hash of this tree is included in the block header and the block header is hashed into the *block hash*, which can be queried from inside a smart contract by means of the `BLOCKHASH` opcode.

We implemented this verification procedure in a smart contract that takes a block number, the transaction data, and a Merkle-Patricia proof of transaction inclusion as inputs, and outputs *accept* or *reject*. We benchmarked the gas cost of this contract by verifying the inclusion of 25 transactions from the Ethereum blockchain. The proof verification has a mean cost of 207,800 gas (approximately 0.06 USD[2]). Note that this cost is only incurred when a bounty is being claimed, and has no impact on "normal" transactions.

Proof of Cheat. We can reduce the gas cost of our Submarine Commitment scheme by not performing a Merkle-Patricia proof verification on every "reveal": instead of requiring parties to prove that their "commit" occurred in commitBlk, we only require them to provide commitBlk and the transaction data, *but no Merkle-Patricia proof*. A party P can then submit a *Proof of Cheat*, a Merkle-Patricia proof demonstrating that an adversary $\mathcal{A}$ backdated their transaction: to backdate their transaction $\mathcal{A}$ had to claim the existence of a non-existing transaction; therefore, there will either be a different transaction or no transaction at the purported transaction index in block commitBlk. If the proof of cheat is accepted, $\mathcal{A}$'s \$deposit is given to P and $\mathcal{A}$'s "commit" and "reveal" are voided.

Checking whether another party cheated is simple to do off-chain, so we expect competing parties to check each other's commits and provide a Proof of Cheat if they witness a cheat. In this setting, P benefits from catching a malicious competitor $\mathcal{A}$ in two ways: $\mathcal{A}$'s claim is voided (potentially netting P's \$bounty) and $\mathcal{A}$'s \$deposit is given to P.

Algorithm CreateForwarder(P, key)

nonces $\leftarrow \mathcal{E}(H'(addr(P), \text{key}))$
address $\leftarrow addr(\text{Contract})$
for $i = 1$ **to** k
  while no contract at address $H(\text{address}, \text{nonces}_i + 1)$
    call *Clone* on contract at address
  address $\leftarrow H(\text{address}, \text{nonces}_i + 1)$
//address now equals $\widehat{\text{addr}}$

Fig. 6: Algorithm for creating a Forwarder at address $\widehat{\text{addr}}$.

B. CREATE-based Construction

In Section VI, we gave a construction of Submarine Commitments that requires the `CREATE2` opcode. Hereafter, we show a different construction relying on the `CREATE` opcode, available in Ethereum today. However, the `CREATE2`-based construction is simpler and has 98.5% (75,000 gas vs 5,000,000 gas, or 0.023 USD vs 1.50 USD respectively[2]) lower gas costs than the `CREATE`-based construction.

When a contract C creates a new contract C_{new} using the `CREATE` opcode, C_{new}'s address is computed as $H(addr(C), nonce(C))$, where $nonce(C)$ a monotonic counter of the number of contracts created by C. (Ethereum's state records this nonce for each contract.)

By chaining a series of contract creations and encoding information in the associated nonce values, we can compute an address for Submarine Commitments. Let Contract be the contract that will receive Submarine Commitments. Let Forwarder be a simple contract that has two functions both of which abort if they aren't being called by Contract:

- *Clone* uses `CREATE` to spawn another Forwarder instance at address $H(addr(\text{Forwarder}), nonce(\text{Forwarder}))$.
- *Forward* sends all funds held by the contract to Contract.

We now describe the three functions that make up a Submarine Commitment:

- *Commit:* P selects a witness key $\leftarrow_\$ \{0,1\}^\ell$ and computes $x := H'(addr(\text{Contract}), \text{key})$ for a suitable ℓ and hash function H' with codomain $\{0,1\}^\ell$. Let $A := addr(\text{Contract})$ and let $\mathcal{E} : \{0,1\}^\ell \to \{0, \ldots, b-1\}^k$ be the function that takes an integer (encoded as a binary string) and reencodes it as a string of length k in base b. P sends \$deposit to address

$$\widehat{\text{addr}} = H(H(\ldots H(A, \mathcal{E}(x)_1 + 1) \ldots, \mathcal{E}(x)_{k-1} + 1), \mathcal{E}(x)_k + 1) \,.$$

- *Reveal:* P sends key to BountyContract.

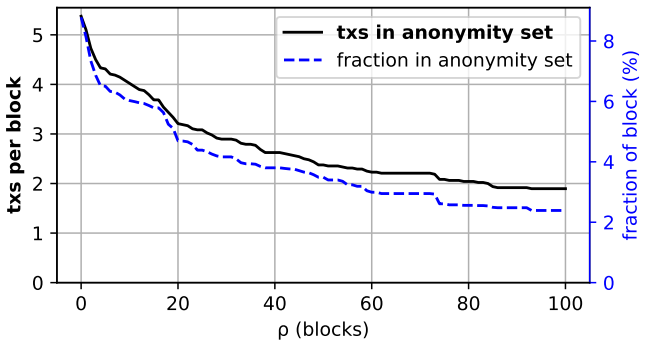

Fig. 7: Size of anonymity set for Submarine Commitments. We show the number of transactions (left) and the fraction of transactions (right) per block that are a part of the anonymity set, as a function of ρ, the size of the commit window. Statistics are computed by averaging 48 block sequences of length ρ, starting at (hourly-spaced) blocks $4430000 + i \cdot 240$ for $i \in [0, 47]$.

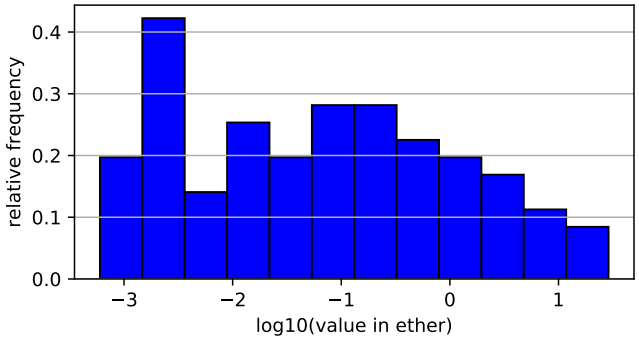

Fig. 8: Histogram of transaction values in anonymity set for Submarine Commitments. We set $\rho = 100$ and take all transactions in the anonymity sets of 48 sequences of 100 blocks, starting at blocks $4430000 + i \cdot 240$ for $i \in [0, 47]$.

- *DepositCollection:* BountyContract repeatedly calls the *Clone* function of appropriate Forwarder instances until a Forwarder is created at address $\widehat{\text{addr}}$. (See Figure 6 for details.) BountyContract then calls *Forward* to make this instance send the the deposit to BountyContract.

Choosing n and b. Since we aren't concerned with collision attacks on H', $n = 80$ provides sufficient security. For $n = 80$, in the ROM, a choice of $b = 4$ minimizes the expected number of contract creations $\log_b(2^n)\left(1 + \frac{b-1}{2}\right)$. In practice, we instantiate H' as a truncated version of Ethereum SHA-3 (Keccak-256) as this is the cheapest cryptographically secure hash function available in the EVM.

C. Empirical Analysis of Anonymity Set Size

The Submarine Commitment constructions from Section VI and Appendix C-B both rely on concealing "commit" transactions in an anonymity set of unrelated transactions: to prevent bug-withholding attacks, the "commit" transactions of the Submarine Commitment scheme must remain concealed until the "reveal" transaction is broadcast. Since the "commit" transactions are indistinguishable from benign transactions sending ether to a fresh address, a transaction to an address A is a part of the anonymity set if:

- The (external) transaction is a regular send of a non-zero amount of ether with an empty data field.
- A has never received any ether or sent any transactions.
- A has no associated code (i.e. A is not a contract).
- A is not involved in any other transactions (internal or external) during the commit window.

In the experiment $\mathbf{Exp}_{\mathcal{A}}^{\text{bntyrace}}$ analyzed in Section VI-A, a commitment is revealed after ρ blocks, where it is assumed that the adversary can rewind up to ρ blocks in the blockchain. Figure 7 shows the size of the anonymity set as a function of this commitment window ρ. Even for $\rho = 100$ (i.e. a 25 minute rewind window at 15 secs/block), an average block still contains two transactions that are part of the anonymity set. Furthermore, 34 of the 48 blocks we studied (70%) contained at least one transaction that is part of the anonymity set. For a full commit window of size $\rho = 100$, we get an anonymity set of approximately 200 transactions, which represents over 2% of *all* transaction traffic in that period.

As Figure 8 shows, the transaction values in the anonymity set span a wide range. Commitments with an associated value between 0.0001 ether and 10 ether (approximately 3,000 USD[2]) are easily concealed.

Appendix D
Security Proofs for BountyContract

In this appendix, we prove Thm. 4 of Section VI-A.

Recall that $\mathsf{commblock}_{P^*} \leftarrow\!\!\$\, [1, n]$ in $\mathbf{Exp}_{\mathcal{A}}^{\text{bntyrace}}$, i.e., P^* commits in a uniformly random block B_i. Thus, $\Pr[\mathsf{commblock}_{P^*} \in [a, b]] = (b - a + 1)/n$ for $1 \le a \le b \le n$. Note that $\mathcal{A}$ is oblivious to the value of $\mathsf{commblock}_{P^*}$ until P^* reveals in block $\mathsf{revblock}_{P^*} = \mathsf{commblock}_{P^*} + \rho$.

We assume an economically rational adversary $\mathcal{A}$. It will be useful to consider another rational adversary $\mathcal{A}_1$ which does not observe $\mathsf{revblock}_{P^*}$. $\mathcal{A}$ follows the same strategy as $\mathcal{A}_1$ until P^* reveals his commitment.

Let X_i be the event in an execution of $\mathcal{F}_{\mathsf{withhold}}$ that $\mathcal{A}_1$ places message "commit" in block B_i. Note that the $\{X_i\}$ may not be independent. Let $p_i = \Pr[X_i = 1]$. As $\mathcal{A}_1$ is oblivious to $\mathsf{revblock}_{P^*}$, the events X_i are independent of $\mathsf{commblock}_{P^*}$. Moreover, let Z_i be the event in an execution of $\mathcal{F}_{\mathsf{withhold}}$ that $\mathcal{A}$ places message "commit" in B_i.

We state some simple claims.

Claim 6. *If* $\mathsf{commblock}_{P^*} \ge i$, *then* $Z_i = X_i$.

Proof. If P^* commits in block i or later, $\mathcal{A}$ does not learn of this commit until at least block $i + \rho$, when P^* reveals. At this point, $\mathcal{A}$ cannot rewind to block B_i and change Z_i. □

Claim 7. *If* $\mathsf{commblock}_{P^*} < i$, *then* $Z_i = 0$.

Proof. Committing in block B_i will not enable $\mathcal{A}$ to frontrun P^*, as P^* committed earlier. If $\mathcal{A}$ did commit in block B_i before P^* reveals (as late as $B_{i+\rho-1}$), then $\mathcal{A}$ rewinds and erases its own "commit" in B_i, to save cost \$deposit. □

Claim 8. $\Pr[Z_i = 1] \ge \Pr[\mathsf{commblock}_{P^*} \ge i \ \wedge \ X_i = 1]$.

Proof. This is an immediate corollary of Claim 6: If $\mathsf{commblock}_{P^*} \geq i$ and $X_i = 1$, then we have $Z_i = 1$. For events A, B with $A \implies B$ we have $\Pr[B] \geq \Pr[A]$. □

Note that $\mathcal{A}$ never benefits from delaying P^*'s messages as, by assumption, $\mathcal{A}$ cannot delay the Submarine Commitment in block $\mathsf{commblock}_{P^*}$.

For adversary $\mathcal{A}$, let $\mathbb{E}_{\mathcal{A}}[\$\mathrm{cost}]$ denote the expected value of $\$\mathrm{cost}$ in an execution of $\mathbf{Exp}^{\mathsf{bntyrace}}_{\mathcal{A}}$ and $p_{\mathsf{wins}}(\mathcal{A})$ be the probability of winning. We have the following lemma:

Lemma 9. *Suppose for a given $\mathcal{A}$ that $q = \Pr[X_{i+k} = 1 \wedge X_i = 1] > 0$ for some $1 < k < \Delta$ and $i + k \leq n$. Then there exists an adversary $\mathcal{A}'$ such that $\mathbb{E}_{\mathcal{A}'}[\$\mathrm{cost}] < \mathbb{E}_{\mathcal{A}}[\$\mathrm{cost}]$ and $p_{\mathsf{wins}}(\mathcal{A}') \geq p_{\mathsf{wins}}(\mathcal{A})$.*

Proof. We construct $\mathcal{A}'$ that emulates $\mathcal{A}$ exactly, except that if $\mathcal{A}$ commits in B_i and in B_{i+k}, then $\mathcal{A}'$ does not commit in B_{i+k}, but commits in block $B_{i+\Delta}$. (If $i + \Delta > n$, then $\mathcal{A}'$ does not make the second commitment.) For any value of $\mathsf{commblock}_{P^*}$, it is easy to see that if $\mathcal{A}$ can frontrun, then $\mathcal{A}'$ can also frontrun P^*. Thus $p_{\mathsf{wins}}(\mathcal{A}') \geq p_{\mathsf{wins}}(\mathcal{A})$.

With probability $1/n$, $\mathsf{commblock}_{P^*} = i + k$. In this case, $\mathcal{A}'$ does not make its second commitment in block $B_{i+\Delta}$, and thus incurs $\$\mathrm{cost}$ at least $\$\mathrm{deposit}$ less than $\mathcal{A}$. Thus, $\mathbb{E}_{\mathcal{A}}[\$\mathrm{cost}] \geq \mathbb{E}_{\mathcal{A}'}[\$\mathrm{cost}] + \frac{q \cdot \$\mathrm{deposit}}{n}$. □

We now prove an upper bound on the probability that $\mathcal{A}$ wins based on the values $\{p_i\}$, i.e., the strategy of $\mathcal{A}_1$.

Lemma 10. $p_{\mathsf{wins}} \leq \frac{\Delta+1}{n}\left(\sum_{i=1}^{n} p_i\right)$.

Proof. Recall that $\mathcal{A}$ is a pure frontrunning adversary. Suppose $\mathsf{revblock}_{P^*} = j$ and thus $\mathsf{commblock}_{P^*} = j - \rho$. As $\mathcal{A}$ cannot rewind more than ρ blocks, for $\mathcal{A}$ to win $\mathbf{Exp}^{\mathsf{bntyrace}}_{\mathcal{A}}$ it must be the case that $\mathcal{A}$ has a message "commit" in a block B_i for $i \in [j - \rho - \Delta, j - \rho]$ (equivalently, $\mathsf{commblock}_{P^*} \in [i, i + \Delta]$). Thus, under a union bound,

$$\begin{aligned} p_{\mathsf{wins}} &\leq \sum_{i=1}^{n} \Pr[Z_i = 1 \wedge \mathsf{commblock}_{P^*} \in [i, i+\Delta]] \\ &= \sum_{i=1}^{n} \Pr[X_i = 1 \wedge \mathsf{commblock}_{P^*} \in [i, i+\Delta]] \\ &= \sum_{i=1}^{n} \Big(p_i \cdot \Pr[\mathsf{commblock}_{P^*} \in [i, i+\Delta]] \Big) \\ &= \sum_{i=1}^{n} \Big(p_i \cdot \frac{\Delta+1}{n} \Big) = \frac{\Delta+1}{n} \Big(\sum_{i=1}^{n} p_i \Big) , \end{aligned}$$

using Claim 6 and independence of X_i and $\mathsf{commblock}_{P^*}$. □

We now prove a lower bound on the expected cost incurred by a frontrunning $\mathcal{A}$. Let $\$\mathrm{cost}_i$ be a random variable denoting commitment costs by $\mathcal{A}$ in B_i.

Lemma 11. $\mathbb{E}[\$\mathrm{cost}] \geq \$\mathrm{deposit} \cdot \sum_{i=1}^{n} \left(\frac{n-i+1}{n}\right) p_i$.

Proof. Let C_i denote the event $(\mathsf{commblock}_{P^*} \geq i)$. Then,

$$\begin{aligned} \mathbb{E}[\$\mathrm{cost}] &= \sum_{i=1}^{n} \mathbb{E}[\$\mathrm{cost}_i] = \sum_{i=1}^{n} \$\mathrm{deposit} \cdot \Pr[Z_i = 1] \\ &\geq \$\mathrm{deposit} \cdot \sum_{i=1}^{n} \Pr[X_i = 1 \wedge C_i] \\ &= \$\mathrm{deposit} \cdot \sum_{i=1}^{n} \Pr[X_i] \cdot \Pr[C_i] \\ &= \$\mathrm{deposit} \cdot \sum_{i=1}^{n} \Big(\frac{n-i+1}{n} \Big) p_i , \end{aligned}$$

using Claim 8 and independence of X_i and $\mathsf{commblock}_{P^*}$. □

Let us restate Theorem 4:

Theorem. *Suppose that $\Delta \geq 4$ and $\$\mathrm{deposit} > \frac{10(\Delta+1)}{9n} \cdot \$\mathrm{bounty}$. Then a pure-frontrunning adversary cannot achieve $\mathbb{E}[\$\mathrm{payoff}] \geq 0$.*

Proof. By Claims 6, 7 and 8, $\mathcal{A}$'s strategy (i.e., the values of Z_i) are fully determined by $\mathcal{A}_1$'s strategy and the value of $\mathsf{commblock}_{P^*}$. We consider the optimal assignment of probabilities p_i, to maximize p_{wins} while minimizing $\mathbb{E}[\$\mathrm{cost}]$.

Let $p = \sum_{i=1}^{n} p_i$. By Lemma 10, $p_{\mathsf{wins}} \leq \frac{(\Delta+1) \cdot p}{n}$.

To achieve p_{wins}, then, we require $p \geq (np_{\mathsf{wins}})/(\Delta+1)$. Let $k = (np_{\mathsf{wins}})/(\Delta+1)$, and assume for simplicity of computation that k is an integer. Now, Lemma 11 states that

$$\mathbb{E}[\$\mathrm{cost}] = \$\mathrm{deposit} \cdot \sum_{i=1}^{n} \Big(\frac{n-i+1}{n} \Big) p_i. \tag{6}$$

For a given value of p, the sum in Eqn. 6 is minimized by concentrating probability mass among $\{p_i\}$ for the largest values of i. Additionally, by Lemma 9, if $p_i = 1$, then $p_{i+1} = p_{i+2} = \ldots = p_{\Delta-1} = 0$, i.e., non-zero p_i values are spaced by Δ. Therefore, as $p_i \in [0, 1]$, Eqn. 6 is minimized when $p_n = p_{n-\Delta} = \ldots = p_{n-(k-1)\Delta} = 1$, and thus:

$$\begin{aligned} \mathbb{E}[\$\mathrm{cost}] &\geq \$\mathrm{deposit} \cdot \sum_{i=1}^{k} \Big(\frac{n - \Delta(i-1) + 1}{n} \Big) \\ &= \$\mathrm{deposit} \cdot \Big(k - k/n - \frac{\Delta(k-1)(k-2)}{2n} \Big) \\ &> \$\mathrm{deposit} \cdot \Big(k - \frac{k^2}{2n} \Big) \\ &= \$\mathrm{deposit} \cdot \frac{np_{\mathsf{wins}}}{\Delta+1} \Big(1 - \frac{p_{\mathsf{wins}}}{2(\Delta+1)} \Big) \\ &\geq \$\mathrm{deposit} \cdot \frac{9np_{\mathsf{wins}}}{10(\Delta+1)}, \end{aligned}$$

as $\frac{p_{\mathsf{wins}}}{2(\Delta+1)} \leq 1/10$ ($\Delta \geq 4$ and $p_{\mathsf{wins}} \leq 1$). Thus,

$$\mathbb{E}[\$\mathrm{payoff}] \leq \$\mathrm{bounty} \cdot p_{\mathsf{wins}} - \$\mathrm{deposit} \cdot \frac{9np_{\mathsf{wins}}}{10(\Delta+1)}.$$

The theorem follows. □

Recall that an α-withholding adversary is one that may frontrun up to block $B_{\alpha n}$ and then posts messages "commit" and "reveal" in block $B_{\alpha n+1}$. It is straightforward, based on the proof of Thm. 4 to show Corollary 5 restated here:

Corollary. *Suppose that $\Delta \geq 4$ and $\$\mathrm{deposit} \geq \frac{10(\Delta+1)}{9n} \cdot \$\mathrm{bounty}$. Then an α-revealing adversary $\mathcal{A}$ achieves $\mathbb{E}[\$\mathrm{payoff}] \leq \big((1-\alpha) \cdot \$\mathrm{bounty}\big) - \$\mathrm{deposit}$.*

From uniform to exponential distributions. Finally, we sketch a proof of an analog Theorem 4 where $\mathsf{commblock}_{P^*}$ follows an exponential distribution rather than a uniform one. From our Poisson model of bug finding in Section IV, we obtained that P^* finds a bug after an exponentially distributed waiting time T_{H} of rate λ_{H}. We assume that P^* commits as soon as it finds a bug.

The only difference in the proof above are the probabilities $\Pr[\mathsf{commblock}_{P^*} \in [i, i+\Delta]]$ in the proof of Lemma 10 and $\Pr[\mathsf{commblock}_{P^*} \geq i]$ in the proof of Lemma 11. For the exponential distribution, we get

$$\Pr[\mathsf{commblock}_{P^*} \in [i, i+\Delta]] = e^{-i\cdot\lambda_{\mathrm{H}}} \cdot (1 - e^{-\Delta\cdot\lambda_{\mathrm{H}}})$$
$$\Pr[\mathsf{commblock}_{P^*} \geq i] = e^{-i\cdot\lambda_{\mathrm{H}}} .$$

Therefore, the analogs of Lemma 10 and Lemma 11 are:

$$p_{\mathsf{wins}} \leq \sum_{i=1}^{n} p_i \cdot e^{-i\cdot\lambda_{\mathrm{H}}} \cdot (1 - e^{-\Delta\cdot\lambda_{\mathrm{H}}}) \tag{7}$$

$$\mathbb{E}[\$\mathrm{cost}] \geq \$\mathrm{deposit} \cdot \sum_{i=1}^{n} p_i \cdot e^{-i\cdot\lambda_{\mathrm{H}}}. \tag{8}$$

Rearranging terms, we get:

$$\mathbb{E}[\$\mathrm{cost}] \geq \$\mathrm{deposit} \cdot p_{\mathsf{wins}} \cdot \frac{1}{1 - e^{-\Delta\cdot\lambda_{\mathrm{H}}}}$$
$$\mathbb{E}[\$\mathrm{payoff}] \leq \$\mathrm{bounty} \cdot p_{\mathsf{wins}} - \$\mathrm{deposit} \cdot \frac{p_{\mathsf{wins}}}{1 - e^{-\Delta\cdot\lambda_{\mathrm{H}}}} .$$

Suppose that the bug-finding period extends over approximately $n = \lambda_{\mathrm{H}}^{-1}$ blocks. That is, we expect honest parties to find *one* bug on average over the full bounty period. Then, (as $\frac{\Delta}{n} \ll 1$), we need

$$\$\mathrm{deposit} \geq \$\mathrm{bounty} \cdot (1 - e^{-\frac{\Delta}{n}}) \approx \$\mathrm{bounty} \cdot \frac{\Delta}{n} , \tag{9}$$

to ensure $\mathbb{E}[\$\mathrm{payoff}] \geq 0$.