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

Lorenz Breidenbach
lorenzb@ethz.ch
Cornell Tech, IC3[†]*,*
ETH Zürich

Philip Daian
phil@cs.cornell.edu
Cornell Tech, IC3[†]

Florian Tramèr
tramer@cs.stanford.edu
Stanford

Ari Juels
juels@cornell.edu
Cornell Tech, IC3[†]*,*
Jacobs Institute

Abstract

Bug bounties are a popular tool to help prevent software exploits. Yet, 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 that incentivize bug disclosure. Our key idea is an *exploit gap*, a program transformation that enables runtime detection, and rewarding, of critical bugs. Our framework transforms programs via *N-of-N-version programming*, a variant of classical N-version programming that runs 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 rigorous 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 design a simple, automated version of the Hydra Framework for Ethereum (`ethereum.org`) and implement two Hydra contracts, an ERC20 standard token and a Monty-Hall game. We evaluate our implementation for completeness and soundness with the official Ethereum virtual machine test suite and live blockchain data.

1 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 become instrumental in 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 gray 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 [2]. 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 [11, 9]. Modest bounties may thus fail to successfully incentivize disclosure by rational actors [38].

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 generic bugs defined as "Extraction of confidential material protected by the Secure Enclave Processor" [38].

Finally, bounties 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 (e.g., [21, 4, 50]). This uncertainty creates a market inefficiency that limits incentives for rational hackers to uncover vulnerabilities.

We introduce the *Hydra Framework*, the first principled approach to bug bounty administration that addresses these challenges. Our 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 our framework's power analytically and empirically.

[*] The first three authors contributed equally to this work.

[†] Initiative for Cryptocurrencies and Contracts, `initc3.org`

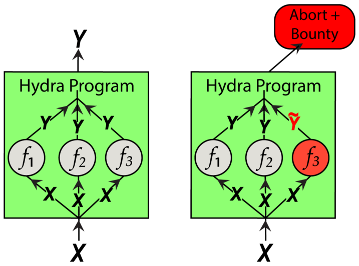

Figure 1: Hydra program with heads f_1, f_2, and f_3. Example on right shows effect of bug induced by input X in f_3.

The Hydra Framework. Our key idea 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 called 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 and the program aborts and pays out a bounty. 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. Moreover, an adversary that breaks one head and, instead of claiming a bounty, tries to generalize the exploit, risks preemption by honest bounty hunters. We show that even when an exploit's market value exceeds the bounty by multiple orders of magnitude, economically rational hackers are incentivized to disclose bugs rather than attempt an exploit.

A Hydra Framework for smart contracts. We focus on smart contracts, programs that execute on blockchains such as Ethereum [10]. They are especially well suited as a use case given several distinctive properties:

- *Heightened vulnerability:* Smart contracts are often financial instruments. Bugs usually directly affect funds, enabling hackers to extract (pseudonymous) cryptocurrency, as shown by tens of millions of dollars worth of Ethereum stolen from [11] and [9]. 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:* A smart contract's cryptocurrency balance is often a direct measure of an exploit value. This facilitates principled bounty price setting in our framework. Moreover, blockchain protocols are often secured through both cryptography and economic guarantees. For the first time, we lift similar economic safety guarantees to the smart-contract level, creating programs with measurable economic security.
- *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 value, creating a stable bounty marketplace.
- *Graceful termination:* Smart contracts are not (yet) mission critical software and can often be aborted with minimal adverse effects, as required for NNVP. Remediation of the DAO and Parity multisig attacks involved refunding users, a mechanism considered in this paper.

We design a Hydra Framework for Ethereum and evaluate it on two applications, an ERC20 token [51] and a Monty Hall game [52]. In both cases, our Hydra contract automatically rewards bugs in any of three deployed heads, implemented in three different programming languages. Our Hydra ERC20 token is deployed on the Ethereum main network (with a 3000 USD bounty), the first principled, automated and trust-free bug bounty. Our framework is applicable to over 76% of Ethereum contracts in use. Our full framework code, tests, and experiments are available at `thehydra.io`.

Major challenges. Several papers [28, 17] criticize traditional N-version programming, observing that multiple versions of a program often exhibit correlated faults—an ostensible hitch in our 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 the majority voting scheme used in N-version programming, 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 [47]. As a result, a naïvely implemented bounty contract is vulnerable to *bug-withholding* attacks: upon finding a bug in one head, a hacker can withhold it and try to compromise all 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 that 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. 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 provide the first quantifiable notion of economic security for smart contracts.
- *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 implement a Hydra Framework for Ethereum and instantiate it for an ERC20 token and a Monty Hall game. We measure costs of running multi-headed contracts on-chain and showcase Hydra's soundness and applicability, concluding that our framework can automatically transform the majority (76%) of contracts used in Ethereum while passing all official virtual machine tests. Our bounty-backed, three-headed Hydra ERC20 token is live on Ethereum.

2 Preliminaries and Notation

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 $\texttt{run}(f,X) := [y_1, y_2, \dots]$ a serial *execution trace* of f starting at the initial state and outputting y_i on input x_i.

Exploits. For a program f, let $\mathbb{I}$ be an abstract *ideal program* that defines f's intended behavior. I.e., for any input X, $\texttt{run}(\mathbb{I},X)$ is the correct output. The input space is assumed to be bounded and input sequences are finite.

We assume that a program may produce a *fallback* output $\perp$ if it detects that the execution is diverging from intended behavior (e.g., throwing an exception on a stack overflow). The ideal program $\mathbb{I}$ 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 $\perp$. Let $A \sqsubset B$ denote that sequence A is a strict prefix of sequence B. The set of fallback traces is then

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

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

Exploit gaps and bug bounties. A program transformation $\mathbb{T}$ combines $N \geq 1$ programs into a program $f^* := \mathbb{T}(f_1, f_2, \dots, f_N)$. Our notion of exploit gap aims to capture the idea that f^* has fewer exploits than the original f_i. However, directly relating the sizes $|E(f^*,\mathbb{I})|$ and $|E(f_i,\mathbb{I})|$ is problematic as these quantities are hard to measure. Instead, we define a *probabilistic* notion of exploit gap, for input sequences sampled from a distribution $\mathcal{D}$ (e.g., the distribution of user inputs to a program).

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

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

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 $\perp$ induces a large exploit gap, yet has no utility. We therefore also require the following notion of availability.

Definition 2 (Availability Preservation). *Let* $F(f)$ *be the set of inputs with fallbacks, i.e.* $X \in F(f)$ *iff* $\texttt{run}(f,X) \in \mathcal{Y}_\perp$. *Then a program transformation* $\mathbb{T}$ *is* availability-preserving *iff* $F(f^*) \subseteq \bigcup_{i=1}^{N} \left(E(f_i,\mathbb{I}) \cup F(f_i)\right)$

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 some of the original programs.

Given a transformation $\mathbb{T}$ that induces an exploit gap, a natural bug bounty for a deployed program f^* rewards bugs in the original programs f_i. Such a bug bounty scheme 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 ($\texttt{gap} \gg 1$), then a discovered bug likely affects one of the programs f_i but not f^*.
3. The bugs are valuable. If $\texttt{gap} > 1$, fixing bugs in the f_i eventually reduces the probability of exploits in f^*.

Achieving an exploit gap. Generically, dynamic runtime checks (e.g., stack canaries, under- or overflow detection) can yield an availability-preserving exploit-gap: the checks result in a fallback output (e.g., a runtime exception), where the original program had an exploit.

A broadly applicable method for achieving an exploit-gap is via *redundancy and fault-tolerance*, e.g., *Recovery Blocks* [42] or *N-version programming* [13]. 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, which we build upon in Section 3. This software paradigm consists in three steps [13, 6]:

1. A specification is written for the program's functionality, API, and error handling. It further defines how to combine outputs of different versions (see Step 3).
2. N versions of the program specification are 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. The N versions are run in parallel and their outputs combined via some voting scheme. N-version programming traditionally uses majority voting between programs to induce an exploit gap [13, 6].

3 N-of-N-version Programming

N-version programming assumes that heterogeneous implementations have weakly correlated failures [13]. Many experiments have challenged this view [28, 17], 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*.

The main differentiator between the traditional setting of N-version programming, and ours, is the role of *availability*. Prior works consider mission-critical systems and thus favor *availability* over *safety* in the face of partial failures. For instance, Eckhardt et al. [17] explicitly ignore the "*error-detection capabilities*" of N-version programming. This setup is not suitable for smart-contracts: As in centralized financial institutions (e.g., stock-markets [44]), the cost of a fault typically trumps that of a temporary loss of resource availability.

Ethereum's community exemplified its preference for safety in this trade-off, when attackers found an exploit in the *Parity Multisig Wallet* [9] and stole user funds. A consortium of "white-hat hackers" used the same bug to move user's funds to a safe account. Despite funds being unavailable for weeks, and reimbursement depending on the consortium's good will, the action was acclaimed by the community and affected users. The simple escape hatch in this scenario (i.e., move funds to a safe account) was deemed a successful alternative to an actual exploit.

We propose trading availability for safety in N-version programming, by replacing the goal of *fault-tolerance* by one of *error detection and safe termination*. Suppose that programs $f_1, \dots, f_N$ have no fallback outputs (i.e., $F(f_i) = \emptyset$). Then majority voting yields a program f^* that also satisfies $F(f^*) = \emptyset$, but the exploit gap may be small. At the other end of the spectrum, we propose *N-of-N-version programming* (NNVP), wherein f^* aborts 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).

Table 1 lists prominent Ethereum smart contract failures. We discuss these in more detail in Appendix A, and argue that a majority could have been abated with NNVP.

3.1 Revisiting N-version Programming

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

Knight and Leveson [29] first showed that the null-hypothesis of *statistical independence* between program failures should be rejected. Yet, such correlated failures only invalidate the N-version paradigm if increased development costs outweigh failure rate improvements.

Unfortunately, in an experiment at NASA, Eckhardt et al. [17] 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 some fault classes by a factor of only 4 (i.e., an exploit gap of $\texttt{gap} \approx 4$).

Fortunately, NNVP provides a better cost-benefit trade-off. In the experiment of Eckhardt et al. [17], three programs failed *simultaneously* with probability 30-5000× lower than a single program (see Appendix B). The actual gain is probably much larger, as Eckhardt et al. did not consider whether program failures were *identical* or not. In NNVP, a failure only occurs if all N versions produce *the same incorrect output*. In any other failure scenario, NNVP aborts. Thus, if loss of availability can be tolerated, NNVP can significantly boost the error detection capabilities of N-version programming.

Contract name	Exploit value (USD)	Root cause	Independence source	Exploit gap
Parity Multisig 2 [46]	300M	Delegate call+exposed self-destruct	programmer/language?	✔/✗
Parity Multisig 1 [9]	180M	Delegate call+unspecified modifier	programmer/language?	✔/✗
The DAO* [14]	150M	Re-entrancy	language	✔
Proof of Weak Hands [7]	1M	Arithmetic overflow	programmer+language	✔
SmartBillions [45]	500K	Bug in caching mechanism	programmer	✔
HackerGold (HKG)* [35]	400K	Typo in code	programmer+language	✔
MakerDAO* [43]	85K	Re-entrancy	language	✔
Rubixi [12]	<20K	Wrong constructor name	programmer+language	✔
Governmental [12]	10K	Exceeds gas limit	None?	✗

Table 1: Selected smart contract failures and potential exploit gaps. The list is extended from [22]. For each incident, we report the value of affected funds (data from [1]), the 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 would have likely induced a large exploit gap and prevented the exploit. Yellow and red lines indicate incidents that Hydra addresses only partially or not at all. Asterisks indicate ERC20 compatible contracts, like our bounty described in Section 6. More details are in Appendix A.

3.2 Smart Contracts are NNVP-Friendly

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 with an exceptionally high "price per line of code" (some token contracts hold over 1M USD per line [1]). Contract code is stored on a public blockchain and exploits often directly extract or destroy stored funds. Yet developing multiple versions is typically cheap in absolute terms.
- *Principled bounty pricing.* A contract's balance is often 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 4).
- *Bounty automation.* Smart contracts enable automation of the full bounty program, from bug detection (with differential testing) to rollback to bounty payments. Bounties administered by smart contracts can satisfy *fair exchange* of bounties for bugs and *guaranteed payment* for disclosure of valid bugs [49]. Bounties are also *transparent* (i.e., the bounty is publicly visible on the blockchain) and may be *dynamically* adjusted to reflect a contract's changing exploit value. The result is a stable, decentralized bounty market.
- *Programming language diversity.* Many exploits in Ethereum arose due to specific language idiosyncrasies. The multiple interoperable languages for Ethereum enable potentially diverse implementations.

3.3 The Hydra Contract

Hydra consists of two program transformations. The first, $\mathbb{T}_{NNVP}$, uses the NNVP paradigm to yield an availability-preserving exploit gap. $\mathbb{T}_{NNVP}$ combines N smart contracts (or heads) $f_1, \dots, f_N$ into a contract f^*, which delegates incoming calls to each head. If all outputs match, f^* returns the output; otherwise, f^* reverts all state changes and returns $\perp$.

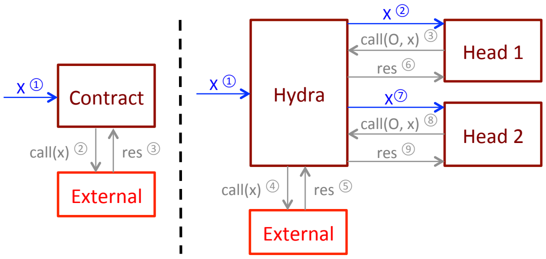

Figure 2: The Hydra NNVP Transformation. (Left) a smart contract that calls an external contract. (Right) a Hydra contract with two heads. The meta-contract acts as a proxy and delegates calls to each head in turn. Calls to external contracts are routed through the meta contract and executed only once, with the obtained result being replayed for each head.

The idea is depicted in Figure 2. The heads are individually deployed and only interact with the Hydra *meta-contract* (MC), which is the logical embodiment of the contract functionality (i.e., the MC holds all assets, and interfaces with external contracts and clients). To maintain consistency while interacting with external contracts, the MC checks that all heads agree on which external interaction to perform, executes the interaction *once*, and distributes the obtained response (if any). Our design and implementation of the $\mathbb{T}_{NNVP}$ transformation for Ethereum smart contracts is described in Section 6.

The second transformation $\mathbb{T}_{Bounty}$ 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*.

Escape hatches. Ideally, bugs could be patched *online*. This is hard in Ethereum as smart contract code cannot be updated after deployment [36]. Best practices [19] suggest enhancing smart contracts with an *escape hatch* mode, which enables the contract's funds to be retrieved, before it's eventual termination and redeployment.

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

- *Security:* The escape hatch's correctness requires special care, as it will not be protected by NNVP.
- *Availability:* The escape hatch must be available for the contract's entire lifetime, or assets could end up stuck.
- *Distributed trust:* All assets should be returned to their rightful owners, or distributed among multiple parties.

For instance, contract funds could be sent to an audited *multisig* contract (possibly implemented as a Hydra contract itself), to distribute trust among multiple parties.

4 Economic Analysis of Hydra Bounties

We formally analyze the exploit gap induced by the Hydra contract, and derive a bounty pricing model to incentivize bug disclosure. We assume that bounties are paid out immediately upon bug disclosure. In Section 5, we refine our analysis in the blockchain model, wherein an adversary may reorder messages sent to smart contracts.

4.1 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 $f_1, \dots, f_N$. 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 $\mathbb{I}$ (although the internal state of f^* may have been corrupted earlier). That is, x is an exploit if $\mathtt{run}(f^*, X \sqcup [x]) \neq \mathtt{run}(\mathbb{I}, X \sqcup [x])$, where X is the sequence of all inputs previously submitted to f^* and $\sqcup$ denotes concatenation.

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

We model bug finding as a Poisson process with rate λ_i, which captures a party's work rate towards finding bugs. We assume that parties sample inputs x from a common distribution of potential exploits $\mathcal{D}$. We then 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

$$\lambda_i \cdot \Pr_{x \in \mathcal{D}}\left[x \in E(f^*, \mathbb{I}) \mid x \in \bigcup_{i=1}^N E(f_i, \mathbb{I})\right]$$

$$= \lambda_i \cdot \frac{\Pr_{x \in \mathcal{D}}\left[x \in E(f^*, \mathbb{I}) \ \wedge x \in \bigcup_{i=1}^N E(f_i, \mathbb{I})\right]}{\Pr_{x \in \mathcal{D}}\left[x \in \bigcup_{i=1}^N E(f_i, \mathbb{I})\right]}$$

$$= \lambda_i \cdot \frac{\Pr_{x \in \mathcal{D}}\left[x \in E(f^*, \mathbb{I})\right]}{\Pr_{x \in \mathcal{D}}\left[x \in \bigcup_{i=1}^N E(f_i, \mathbb{I})\right]} = \lambda_i \cdot \mathtt{gap}^{-1} \ . \quad (2)$$

Let us first consider the strong assumption of independent program failures. For a head f_i, let p be the probability that an input $x \in \mathcal{D}$ is an exploit for f_i. We get

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

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

The gap can be empirically estimated using Equation (1). For the test suites considered in the experiments of Eckhardt et al. [17], the average gap for three program variants is 4400 (see Appendix B for details).

4.2 Analyzing 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 in all heads (i.e., a full exploit) cannot be detected and rewarded by the meta-contract f^*. 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 deplete the contract's balance. W.l.o.g, we model a single adversary $\mathcal{A}$ with work rate λ_{M}. Indeed, for m (non-colluding) adversaries with work rates $\lambda_1, \dots, \lambda_m$, it suffices to analyze the party with rate $\lambda_{\mathrm{M}} = \max_{1 \leq i \leq m} \lambda_i$. If the bounty incentivizes this party to act honestly, less efficient parties will have the same incentive.

Let T_{H} be denote the waiting time until an honest party finds a bug. T_{H} is exponentially distributed with rate λ_{H}. Let T_{M} be the waiting time until $\mathcal{A}$ finds an exploit against f^*, which is exponential with rate $\lambda_{\mathrm{M}} \cdot \mathtt{gap}^{-1}$. We analyze two cases: (1) $\mathcal{A}$ finds an exploit against f^*, and (2) $\mathcal{A}$ finds a bug for a strict 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

$$\Pr[T_{\mathrm{M}} < T_{\mathrm{H}}] = \frac{\lambda_{\mathrm{M}} \cdot \mathtt{gap}^{-1}}{\lambda_{\mathrm{H}} + \lambda_{\mathrm{M}} \cdot \mathtt{gap}^{-1}} = \frac{\lambda_{\mathrm{M}}}{\lambda_{\mathrm{H}} \cdot \mathtt{gap} + \lambda_{\mathrm{M}}} \ ,$$

which naturally decays as the exploit gap increases.

In the second case, a bounty can incentivize early disclosure. Suppose $\mathcal{A}$ found a bug in a head. If $\mathcal{A}$ discloses it, her payout is $\mathtt{payout}_{\mathrm{H}} :=$ \$bounty. Instead, if she conceals the bug and continues searching for exploits, she risks a payout of 0 if another party claims the bounty first. Her expected payout, $\texttt{payout}_\text{M}$, is thus

$$\Pr[T_\text{M} < T_\text{H}] \cdot \$\text{balance} = \frac{\lambda_\text{M}}{\lambda_\text{H} \cdot \mathsf{gap} + \lambda_\text{M}} \cdot \$\text{balance}\ .$$

Let $\alpha := \frac{\lambda_\text{H}}{\lambda_\text{M}}$. Then, honest behavior is incentivized if

$$\frac{\texttt{payout}_\text{H}}{\texttt{payout}_\text{M}} > 1 \iff \$\text{bounty} > \frac{1}{\alpha \cdot \mathsf{gap} + 1} \cdot \$\text{balance}\ .$$

We may assume that $\lambda_\text{M} = \lambda_\text{H}$ (i.e., $\mathcal{A}$'s work rate is equal to the *combined* work rate of honest parties). Then, for independent program failures (see Equation (3)) the bounty decays exponentially in the number of heads N.

Thus, given estimates of α and gap, we get a principled bounty pricing that incentivizes bug disclosure. For example, in the experiment of Eckhardt et al. [17], a three-headed Hydra could sustain a bounty 3 *to* 4 *orders of magnitude below an exploit's value*.

5 The Bug-Withholding Problem

Our analysis in Section 4 assumed that a bounty is paid immediately when a bug is claimed. Hereafter, we refine our analysis by modeling bounty smart-contract execution with respect to a powerful adversary, that can cheat users by exploiting blockchain network protocols. We highlight the *bug-withholding* attack and propose and analyze a solution called *Submarine Commitments*.

Front-running. The issue is that transactions may not be ordered in blocks by network submission time. When a user sends a bounty-claim transaction τ to the network, an adversary may *front-run* the user, and insert its own bounty-claim τ' earlier in the chain [47]. It does this by ensuring faster network propagation of τ' or by causing a miner to order τ' before τ, e.g., by paying a higher fee (more gas in Ethereum) or corrupting the miner.

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 against all heads. If another party in the meantime claims the 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 ensure it claims the bounty first, thus nullifying any economic incentives for early disclosure.

We propose a formal model for blockchain security, expressed as an ideal functionality $\mathcal{F}_\mathsf{withhold}$. It captures front-running, but is far stronger than previous models (e.g., Hawk [30]). We present a basic bug-bounty contract BountyContract in $\mathcal{F}_\mathsf{withhold}$. Refining our analysis of Section 4, we show how bug withholding breaks incentives for bug disclosure in BountyContract. We show that commit-reveal schemes are an insufficient defense, and therefore introduce *Submarine Commitments*. We prove, in an $\mathcal{F}_\mathsf{withhold}$-hybrid world, that *using Submarine Commitments for* BountyContract *drastically reduces the payoff of a bug-withholding adversary*.

5.1 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* any transaction by a bounded number of blocks.

This reflects an adversary's ability to monitor transactions, mount network-level attacks, control client accounts, and even corrupt or bribe miners to alter legitimate blocks. Previous models, e.g., [30], considered weaker attacks in which $\mathcal{A}$ can arbitrarily reorder transactions in a pending block. They are equivalent to weak history-revision attacks with only a single block.

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 reorder 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* $\mathcal{F}_\mathsf{withhold}$ characterizing an (δ, ρ)-adversary $\mathcal{A}$. We give details on $\mathcal{F}_\mathsf{withhold}$ in Appendix D.1.

Notation. Let $\mathbb{B} = \{B_1, \ldots, B_{\mathbb{B}.\mathsf{Height}}\}$ be a *blockchain*, i.e., an ordered sequence of *blocks*. Here, $\mathbb{B}.\mathsf{Height}$ is the number of blocks in $\mathbb{B}$. A block $B_i = \{\tau_{i,1}, \ldots, \tau_{i,s}\}$ is an ordered sequence of s transactions, i.e., B_i has blocksize s. For simplicity, we assume no forks. If a fork occurs, $\mathcal{A}$ may operate on what it believes to be the main chain.

Let $\mathcal{P} = \{P_0, P_1, \ldots, P_m\}$ be 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 $\mathcal{A}$.

5.2 The BountyContract Smart Contract

Within the $\mathcal{F}_\mathsf{withhold}$-hybrid model, we specify a contract BountyContract to administer a single bug bounty, using a simple *commit-reveal* scheme to prevent adversarial copying and resubmission of bugs. BountyContract has parameters $\Delta > \delta + \rho$, \$deposit and \$bounty. It takes as input a commitment to a bug in some block B_i (via transaction "commit"), which 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"

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

Init: CommitList, RevealList $\leftarrow \emptyset$

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

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

On receive $\tau$ = ("claim", height) from $P_i$:        // $P_i$ tries to claim bounty
  assert (height; $P_i$) $\in$ RevealList
  assert $\mathbb{B}$.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
```

Figure 3: The BountyContract **smart contract.**

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 that determines whether a submitted bug is valid. In the $\mathcal{F}_{\mathsf{withhold}}$-hybrid model, BountyContract is fed a height-n blockchain $\mathbb{B}$, which is replayed after being generated by $\mathcal{F}_{\mathsf{withhold}}$, i.e., transactions are executed as ordered by $\mathcal{F}_{\mathsf{withhold}}$ in $\mathbb{B}$.

Bug withholding in BountyContract**.** The contract in Figure 3 uses a cryptographic commit-reveal scheme, a simple folklore solution to certain front-running attacks [26]. This works if $\mathcal{A}$ *cannot post a valid commitment itself until it sees a victim's reveal.* For instance, BountyContract prevents $\mathcal{A}$ from trying to learn and steal the committed bug from an honest player P_0.

Unfortunately, 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 until P_0 sends a "commit". $\mathcal{A}$ then knows that P_0 is trying to claim a bounty, and can *front-run* P_0*'s commitment* by posting her own "commit" ahead in the blockchain.

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

Impact of bug withholding. In our analysis of Hydra bug bounties in Section 4.2, we assumed that $\mathcal{A}$ risks forfeiting a payout of $bounty if she conceals a bug. However, front-running has the potential of removing incentives for early disclosure, as $\mathcal{A}$ can *ensure* a payout of $bounty by front-running the honest bounty hunter.

If $\mathcal{A}$ conceals a bug, she finds a full exploit before the bounty is claimed with probability $q := \Pr[T_{\mathrm{M}} < T_{\mathrm{H}}]$. Otherwise she front-runs and steals the bounty. Her expected payout is $q \cdot \$\mathrm{balance} + (1-q) \cdot \$\mathrm{bounty}$.

If $\mathcal{A}$ discloses the bug, her payout is $bounty. To incentivize disclosure, we need $\$\mathrm{bounty} > \$\mathrm{balance}$, as in a standard bounty with no exploit gap. We now show a solution that thwarts bug-withholding attacks in Ethereum, thus re-instantiating positive incentives to disclose bugs.

5.3 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 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 $\$\mathrm{val} \geq \$\mathrm{deposit}$ 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 (honest bounty hunters could use a *mixer*), and $val is sampled from an appropriate distribution of values $\$\mathrm{val} \geq \$\mathrm{deposit}$, $\mathcal{A}$ cannot distinguish transaction τ from other sends to fresh addresses. As we show in Appendix C.2, 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. Two concrete Submarine Commitment constructions are in Appendix C.

5.4 Analysis of Submarine Commitments

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. Details are in Appendix D.1, although our model is understandable without detailed knowledge of $\mathcal{F}_{\mathsf{withhold}}$.

Withholding game: $\mathbf{Exp}_{\mathcal{A}}^{\mathsf{bntyrace}}$. Figure 4 shows the simple game used in our security analysis, denoted by $\mathbf{Exp}_{\mathcal{A}}^{\mathsf{bntyrace}}$. 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 $\mathbb{B}$ of length $n' = n + \Delta$.

In this game, a player can send only two messages: $(\text{“commit”}, \$\text{deposit})$, and “reveal”. To model Submarine Commitments, we assume that P^*'s commit message is opaque to $\mathcal{A}$, i.e., $\mathcal{A}$ cannot detect its presence in a block and it does not count toward the block's size.

For clarity's sake, we first analyze Submarine Commitments outside the Poisson framework of Section 4. Our results also hold in that setting, with a slightly tighter bound for Theorem 3 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 $(\text{“commit”}, \$\text{deposit})$, 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 she posts a valid “commit” before P^* does, and also posts a corresponding “reveal” to claim the bounty. We let $p_{\mathsf{wins}} = \Pr[(\text{TRUE}, \cdot) \leftarrow \mathbf{Exp}_{\mathcal{A}}^{\mathsf{bntyrace}}]$. As a first goal, an economically rational adversary $\mathcal{A}$'s aims to *maximize its expected payoff*, namely

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

Of course, $\mathcal{A}$ can always post a “commit” in B_1 followed by a “reveal” within Δ blocks, in which case it achieves $p_{\mathsf{wins}} = 1$ with $\$\text{payoff} = \$\text{bounty} - \$\text{deposit}$, which is optimal. But then it achieves no withholding.

Results. A compelling withholding strategy for $\mathcal{A}$ is to reveal a bug only by *front-running* P^*, i.e., a *pure front-running* strategy. That is, if P^* sends a “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” 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 parameters, $\mathcal{A}$ achieves no benefit, i.e., positive expected payoff, via pure front-running. Intuitively, this is because front-running is expensive: Since $\mathcal{A}$ observes a “commit” message from P^* too late to remove it by rewinding, $\mathcal{A}$ must post “commit” messages continuously to ensure that it can front-run P^*. We prove the following in Appendix D.

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

Init: $n \leftarrow n' - \Delta, \$\text{cost} \leftarrow 0, \mathsf{commblock}_{P^*} \leftarrow_{\$} [1, n]$
$\mathcal{A}^{\{\mathbb{B} \leftarrow \mathcal{F}_{\mathsf{withhold}}(\{P_0 = P^*, P_1\}, n, \delta, \rho, s)\}}$    //$\mathcal{A}$ 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 $\$\text{deposit}$
if $\big(\exists (1 \leq i \leq \mathsf{commblock}_{P^*} \wedge i \leq j \leq \min(i + \Delta, n))$ 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$(\text{FALSE}, \$\text{payoff} := -\$\text{cost})$
```

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

Theorem 3. *Let $\Delta \geq 4$ and $\$\text{deposit} > \frac{10(\Delta+1)}{9n} \cdot \$\text{bounty}$. Then a pure front-running adversary has $\mathbb{E}[\$\text{payoff}] < 0$.*

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

Example 1. *Consider a bounty in Ethereum, 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 $\$\text{deposit} \geq$ 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 use other strategies. We consider in Appendix D a generalized α-*revealing* strategy that involves conditional *preemptive* bug disclosure. We show that this strategy does no better than pure front-running.

6 Design and Implementation

We implemented a decentralized automated bug bounty for Ethereum smart contracts. We describe the main technical deployment challenges, and explain our design.

The EVM. The Ethereum Virtual Machine (EVM) is a simple stack-based architecture [53]. Smart contracts can access three data structures: a stack, volatile memory, and 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 currency. The EVM executes the contract's code in a sequential, deterministic, single-threaded fashion. Operations can read and write to stack, memory or storage, and spawn a new call frame (with a fresh 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, revert all changes made in the current call frame (e.g., storage updates, transfers of ether), and report an exception to the callee.

6.1 An EVM Execution Environment

To achieve the full power of our Hydra bug bounty, N smart contract versions are run on the blockchain. While we could also run a bounty program off-chain (for a single deployed contract), this would not provide an exploit gap, a key property in our analysis of attacker incentives.

The main challenge is the implementation of the "Execution Environment" [13, 6], the agent that coordinates the N heads and combines their outputs. Its complexity should be minimal, as it is part of the Trusted Computing Base (TCB) of our application: a bug in the coordinating agent is likely an exploit against the Hydra contract.

A proxy meta-contract. As we showed in Figure 2, the logical embodiment of a Hydra contract is a proxy *meta-contract* (MC), which coordinates N deployed contract versions (or heads). Clients and other contracts only interact with the MC. The heads only respond to calls from the MC, and do not hold any ether themselves.

The MC delegates all incoming calls to each head, and verifies that the obtained outputs match. If so, it returns that output. Otherwise, it throws an exception, to revert all changes made by the heads. The $\mathbb{T}_{Bounty}$ transformation described in Section 3.3 is implemented as a simple wrapper around the MC, which catches the above exception, pays out a bounty, and enters an escape-hatch mode.

Maintaining consistent blockchain interactions. As the EVM execution is deterministic, the result of a contract call is fully determined by the call's input, the contract code and the current blockchain state. If smart contracts were executed in isolation, the above proxy contract would thus be sufficient. However, most smart contracts also interact with the blockchain, e.g., by accessing information about the current transaction (such as the sender's address) or by calling other contracts, and the MC must thus guarantee consistency among the heads.

We illustrate the issue in Figure 5 with a Solidity code snippet (top-left) and corresponding EVM opcodes (bottom-left). The function $f(x)$ makes a call to $g(x)$ in the calling contract (`msg.sender`) and reimburses any sent ether (`msg.value`). If used as a head in a Hydra contract, this code snippet presents multiple issues.

1. `CALLVALUE` and `CALLER` are modified when the MC delegates a call to the head. `CALLER` will now be the MC's address, and `CALLVALUE` will be zero.
2. The heads cannot send ether as they do not hold any.
3. With N heads, $g(x)$ is called N times instead of once. The heads might also obtain different return values.

To resolve these issues, the heads are *instrumented* prior to deployment so that all interactions with the blockchain are mediated by the MC. While these modifications could be made in a high-level language (e.g., Solidity), we opt for a more generic, automated, and globally applicable solution that operates on the EVM opcodes of a compiled contract (the instrumentation is thus agnostic to the language used to develop the heads). Opcode instrumentations are essentially of two types:

- *Environment Information.* We ensure that all heads share the view of a common Hydra contract. The `ADDRESS` opcode (which returns the current contract's address) is modified to return the MC's address. The heads reject all calls that do not emanate from the MC. The MC also forwards `CALLVALUE` and `CALLER` to the heads as extra call arguments, to make the proxy delegation transparent. These opcodes are overwritten accordingly in the heads to read from the call data.
- *System Operations.* Opcodes that interact with other blockchain entities (e.g., calling a contract, reading account balances, or logging messages) are rewritten as *callbacks* to the MC. The MC checks consistency among the heads' callbacks and issues the required operations on their behalf. The instrumentation requires some extra volatile memory to store callback arguments, so all memory accesses in the original code are shifted by a fixed offset to create a scratch space.

The instrumented heads are independently deployed on chain. We now discuss the callback mechanism, as well as the soundness and applicability of our approach.

Callbacks. Due to the sequential nature of the EVM, we designed the Hydra meta-contract to optimistically responds to callbacks. That is, when the first head runs, the MC executes all callbacks (e.g., external calls) and records the callback arguments and return values. When the remaining heads run, the MC verifies consistency of requested callbacks and replays the responses. If heads request different callbacks, the MC throws an exception, reverting all changes and triggering the bounty payment.

To maintain consistency between heads, and avoid potential *read-write inversions* (e.g., if heads send ether and read contract balances in different orders), the program specification is required to define a *total-ordering* of the read and write operations issued by the heads.

Tail-call optimization. A design pattern for smart-contracts ("Checks-Effects-Interactions" [19]) suggests that interactions with other blockchain entities should oc-

```
function f(int x) payable {
 // reimburse sender and call g(x)
 (msg.sender).g.value(msg.value)(x);
}
```

⇓

```
MSTORE(M, 0x7877b803) #store sig of g in memory
MSTORE(M+4, CALLDATALOAD(4)) #store x
PUSH32(0) #output size and memory location
PUSH32(0) #output memory
PUSH32(36) #input size
PUSH32(M) #input memory
CALLVALUE #use msg.value as the call value
CALLER #use msg.sender as the dest address
GAS
CALL #this opcode will be instrumented
```

⇒

```
MSTORE(M*, 0x7877b803) #store sig of g in memory
MSTORE(M*+4, CALLDATALOAD(4)) #store x
MSTORE(M*+36, CALLDATALOAD(36)) #store sender
MSTORE(M*+68, CALLDATALOAD(68)) #store value
PUSH32(0) #output size
PUSH32(0) #output memory
PUSH32(100) #input size
PUSH32(M*) #input memory
PUSH32(0) #send 0 ether
PUSH32(MCaddress) #destination address of the call
GAS
CALL #after call returns, cleanup stack
```

(⇑)

```
function f(int x, addr sender, uint val) {
 // send all call args to meta-contract
 MC.call(bytes4(sha3("g(int256)")),x,sender,val);
}
```

Figure 5: EVM instrumentation of Hydra heads (simplified example). (Top left) Solidity function that calls $g(x)$ in the calling contract (`msg.sender`) and sends back all ether (`msg.value`). (Bottom left) EVM bytecode for the call to $g(x)$. `MSTORE(a, v)` is syntactic sugar for {`PUSH32(a)`,`PUSH32(v)`,`MSTORE`} which writes value `v` to memory address `a`. `CALL` consumes 7 stack items: gas amount, address to call, ether amount to send, and memory location and size for call arguments and outputs. (Bottom right) Instrumented bytecode: `CALLVALUE` and `CALLER` are read from function arguments. All call data is stored in memory and used as arguments for a callback to the MC. (Top right) Functionally equivalent Solidity code for the instrumented bytecode.

cur last in a call. For contracts that follow this paradigm, a tail-call optimization can be applied to callbacks.

Instead of calling into the MC, the heads simply append any required call or log operations to the calls' return value. Operations that read blockchain state (e.g., balance checks) are not instrumented. The MC then collects the return values from all heads, verifies consistency, and executes all interactions before returning.

Exception handling. Recall that the EVM halts when contracts perform illegal operations, e.g., explicitly throwing exceptions or running out of gas. Ideally, we would classify any divergence in the heads' 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. Explicit exceptions are thus instrumented to return a special value to the MC, so as to be distinguished from an out-of-gas exception. If all heads throw an explicit exception, the MC propagates the exception to the caller.

6.2 Limitations.

Our Hydra head instrumenter, written in Haskell, applies simple opcode rewriting rules (see Figure 5), which are verified to preserve program invariants such as stack and memory layout. Our modifications impact the heads' gas consumption, yet the overhead is minor (see Section 7). Rewriting opcodes also modifies the layout of the bytecode, so all `JUMP` instructions are updated accordingly.

The instrumentation applies to contracts written in any high-level language that compiles to the EVM, and requires no changes to the EVM. We have not yet implemented callbacks for the infrequent `CREATE` and `SELFDESTRUCT` opcodes. We do not yet support opcodes that modify a head's code (e.g., `DELEGATECALL`). These are often used to load libraries into a contract. Using such opcodes would require the library code to be instrumented itself, which is possible in principle. We note however that code delegation is typically at odds with the multiversion programming philosophy: if all heads call the same library contract, a library bug could yield an exploit. We leave Hydra-based libraries to future work.

7 Evaluation

This paper's goal is not to rigorously measure correlations between smart contract faults, but to propose a novel principled bug bounty framework built upon an assumed *exploit gap*. We leave a thorough analysis of smart contract failure patterns to future work. We evaluate our framework under standard software metrics: TCB size, soundness, applicability and performance. We conclude with a discussion of our development process.

Workloads. To test soundness, applicability and performance of Hydra contracts, we use three workloads: (1) The official suite of test contracts for the EVM[1]; (2) All contracts used in Ethereum between Dec. 7 2017 and Feb. 7 2018; and (3) two representative smart-contract applications developed by the authors. Implementations of Submarine Commitments in Ethereum, and a thorough analysis of the resulting anonymity sets for bounty claiming transactions are in Appendix C.

We developed a generic ERC20 contract [51] for token transfers, and a *Monty Hall Lottery*, wherein two participants play a multi-round betting game [52]. In both cases, three authors independently developed one

[1] https://github.com/ethereum/tests/tree/develop/VMTests

Opcode	Contracts	Transactions	Difficulty
CODECOPY	50,147 (14%)	5,646,607 (27%)	medium
CALLCODE	30,109 (8%)	1,213,064 (6%)	hard
SELFDESTRUCT	24,707 (7%)	739,249 (4%)	easy
DELEGATECALL	19,749 (6%)	2,695,326 (13%)	hard
CREATE	11,559 (3%)	1,143,961 (5%)	easy
Other	6681 (2%)	195,569 (1%)	-
None	268,652 (76%)	12,780,929 (61%)	supported

Table 2: Frequency of main unsupported opcodes. For blocks 4690101 to 5049100 on the Ethereum network, we count how many transactions use an opcode that cannot currently be handled by our Hydra Framework. We further record the fraction of unique smart contract codes that contain those opcodes, and the difficulty in adding support for each opcode.

head in each of *Solidity*, *Serpent*, and *Vyper*, the main programming languages in Ethereum. These languages have different design tradeoffs (in terms of ease-of-use, low-level features or security) and are by themselves a valuable source of diversity between our Hydra heads.

- *The Hydra ERC20 token:* The ERC20 token-transfer API has been thoroughly peer reviewed [51], and is supported by most of the highest-dollar contracts in Ethereum (as of February 2018, the combined market cap of the top ten Ethereum tokens is over 20 billion USD [1]). Notably, the exploit in the DAO [11] was partially present in the code managing tokens.

  Our three-headed Hydra token is deployed on the main Ethereum network and can be used as a drop-in replacement for any ERC20 token, e.g., in the DAO [11] and ether.camp [35] contracts. When a user submits a token order, the MC delegates to all heads and validates the order upon agreement. Our initial bounty is 3000 USD, which we will increase as the contract undergoes further audit, review, and testing.

- *A Hydra Monty-Hall lottery:* In this game, one party, the *house*, first hides a reward behind one of n doors. The *player* bets on the winning door, and the house opens k other non-winning doors. The player may then change his guess. If he guessed correctly, the player wins the reward; otherwise the house collects the bet.

  A fourth author wrote a specification describing the contract's API and 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.

TCB size. Our design from Section 6.1 is generic, and covers both of our target applications (and the majority of our other workloads, see below). The instrumenter for Hydra heads is written in 1500 lines of Haskell, and applies simple code parsing and rewriting rules. The MC's proxy functionality is implemented in EVM assembly. We also wrote an MC in Solidity (185 lines) that applies tail-call optimization to callbacks. As the Hydra Framework is application-agnostic, we believe this is a reasonable TCB. It should also be relatively easy to write a formal specification for the simple functionality of the MC and instrumenter, although we have not attempted this.

Completeness and correctness. To evaluate completeness of our Hydra instrumenter, we consider all Ethereum transactions for blocks 4690101 - 5049100 (Dec. 7 2017 to Feb. 7 2018). For each transaction, we test whether our instrumenter supports the evaluated code (see Section 6.2 for unsupported opcodes). We find that 61% out of 21M transactions, or 76% of 270K unique smart contracts, are compatible with Hydra. Table 2 breaks down the contracts that Hydra currently cannot handle. This analysis supports the fact that Hydra could be usable for the majority of Ethereum contracts, both by deployed code and transaction volume.

We verify soundness by running the official EVM test suite[1] on Hydra contracts. That is, we replace every contract in the test suite by a Hydra contract, and ensure all observable side effects (e.g. logs, external calls, return values, computation outputs) are unchanged. This test suite is used to evaluate EVM implementations, including executable formal specifications of the virtual machine [22]. It is thus critical that the suite be comprehensive: any gap in coverage represents a potential consensus break among official EVM implementations, with impact far beyond Hydra. Hydra passes all tests for contracts it supports (6% of tests contain unsupported operations, see Section 6.2). This gives us extremely high confidence in the soundness of our transformation. We are extending the test suite and completeness of our framework towards maximal assurance for our TCB, including to all official Ethereum tests beyond VM tests.

Gas costs. Running N copies of a smart contract incurs an overhead on gas consumption. Some Ethereum projects, notably the Vyper language, already trade gas efficiency for security. Moreover, a transaction's gas cost can be offloaded onto the contract *owner*, thus dispensing users from Hydra's gas overhead. In any event, for small yet common workloads, the main gas cost of a transaction is the fixed "base fee". As the MC 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 6 compares gas costs for Hydra contracts with 1-5 heads to a linear scaling of a single non-instrumented contract. We show results for the five non-static calls in the ERC20 API, and for a full Monty Hall game (five transactions), with and without tail-call optimization.

For the ERC20 contract, the main cost is the transaction's base fee of 21,000 gas. A call to the MC incurs

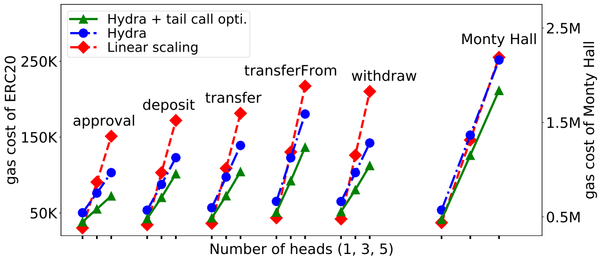

Figure 6: Gas cost of Hydra contracts with N heads. We compare the Hydra contract—with and without tail-call optimization for callbacks—to a linear scaling of a single contract for the ERC20 API (left) and a Monty Hall game (right).

an overhead of about 8000 gas (independent of the number of heads) or about 0.08 USD[2]. Each function call ends in a LOG callback to the MC (to log an "Approval" or "Transfer" event, as mandated by the ERC20 specification). The withdraw function also sends ether to the calling party. Applying the tail-call optimization results in significant savings for these callback-heavy functions.

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. The tail call optimization still results in savings at the end of the game, when the winnings are sent to the house or player.

Evaluation of gas costs (and anonymity set sizes) for Submarine Commitments are in Appendix C. These costs only affect the transaction that claims the bounty.

Observations on the development process. After writing three heads independently, we commonly tested our contracts for discrepancies and found multiple bugs in each head, *none of which* impacted all heads simultaneously. Examples include a misunderstanding of the ERC20 API, integer overflows, "off-by-one" errors in the Monty Hall game, and a vulnerability to an only recently discovered EVM anti-pattern that lets a contract silently increase another contract's ether balance via the SUICIDE opcode. Notably, all these bugs could have been exploited against a single contract, yet none of them appear useful against all heads simultaneously.

In addition to the exploit gap induced by Hydra, the NNVP development process itself increased the quality of our contracts. For the Monty Hall, ensuring compatibility between heads required writing a detailed specification, which revealed many blind spots in our original design. Moreover, *differential testing* [37] (verifying agreement between heads on random inputs) was remarkably simpler for exercising multiple code paths for the Monty Hall game, compared to a standard test suite.

8 Related Work

Software assurance and fault-tolerance are well-studied topics with an extensive literature. N-version programming [13, 6, 18] in particular was introduced decades ago and challenged in influential studies [17, 28] (see Section 2). Nagy et al. use N-version programming to construct honey-pots for detecting web exploits [40].

Bitcoin and, more importantly, Ethereum [10] have popularized smart contracts [48] and script-enhanced cryptocurrency [25]. Research on smart contract security is burgeoning and includes: Analysis of common contract bugs [15, 33, 5], static analysis and enhancements for Solidity [33], formal verification tools [8, 23, 3], design of "escape hatches" [36], DoS defenses for miners [34], trusted data feeds [54], and formal EVM semantics [22, 24]. While promising, none of these tools and techniques have yet seen mainstream adoption, nor do they relate directly to our explorations in this paper.

In a closely related work, Tramèr et al. [49] consider using smart contracts for bug bounties (using SGX), but not the converse, i.e., bounties for smart contracts.

Bug withholding is related to selfish-mining [20], where a miner withholds blocks to later nullify other miners' work. As selfish mining operates at the block level and bug withholding at the application level, they differ in their mechanisms, analysis, and implications.

Submarine Commitments hide bounty claims among normal Ethereum transactions and relate to *cover traffic* techniques such as anonymity networks (e.g., Tor [16]), network-based covert channels [39], steganography and watermarking [27]. Submarine Commitments differ in that they assume ultimate opening of a hidden value.

Several works [30, 26, 49] model blockchain-level adversaries. They consider an adversary that can reorder transactions within a given block, however, and not the much stronger model of chain-rewriting we explore here.

9 Conclusion

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

We have applied our framework to smart contracts, highly valuable and vulnerable programs that are particularly well suited for fair and automated bug boun-

[2] As of February 2018, 1 ether is worth roughly 1000 USD and a gas price of 10^{10} wei) is standard according to https://ethgasstation.info. A value of 1 ether corresponds to 10^{18} wei.

ties. We have formally shown that Hydra contracts incentivize bug disclosure, for bounties orders of magnitude below an exploit's value. We have modeled strong bug-withholding attacks against on-chain bounties, and analyzed Submarine Commitments, a generic defense to front-running that hides transactions in ordinary traffic.

Finally, we have designed and evaluated a Hydra Framework for Ethereum, and rigorously tested its soundness and applicability to the majority of Ethereum contracts today. We used this framework to construct a Hydra ERC20 token and Monty Hall game. The former is live in production on Ethereum, and represents the first principled and trust-free bug bounty offering.

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. Phil Daian is supported by a National Science Foundation Graduate Research Fellowship. Lorenz Breidenbach was supported by the *ETH Studio New York* scholarship.

References

[1] Cryptocurrency market capitalizations. `https://coinmarketcap.com/tokens/`.

[2] ABLON, L., LIBICKI, M. C., AND GOLAY, A. A. *Markets for cybercrime tools and stolen data: Hackers' bazaar*. Rand Corporation, 2014.

[3] AMANI, S., BÉGEL, M., BORTIN, M., AND STAPLES, M. Towards verifying Ethereum smart contract bytecode in Isabelle/HOL. *CPP. ACM. To appear* (2018).

[4] ARGHIRE, I. Researchers claim Wickr patched flaws but didn't pay rewards, Oct. 2016. `http://www.securityweek.com/researchers-claim-wickr-patched-flaws-didnt-pay-rewards`.

[5] ATZEI, N., BARTOLETTI, M., AND CIMOLI, T. A survey of attacks on Ethereum smart contracts (SoK). In *International Conference on Principles of Security and Trust* (2017), Springer, pp. 164–186.

[6] AVIŽIENIS, A. The methodology of N-version programming. In *Software Fault Tolerance*, M. R. Lyu, Ed. John Wiley & Sons Ltd, 1995.

[7] BANISADR, E. How $800k evaporated from the powh coin ponzi scheme overnight, 2018. `https://blog.goodaudience.com/how-800k-evaporated-from-the-powh-coin-ponzi-scheme-overnight-1b025c33b530`.

[8] BHARGAVAN, K., DELIGNAT-LAVAUD, A., FOURNET, C., GOLLAMUDI, A., GONTHIER, G., KOBEISSI, N., KULATOVA, N., RASTOGI, A., SIBUT-PINOTE, T., SWAMY, N., ET AL. Formal verification of smart contracts: Short paper. In *ACM PLAS* (2016), ACM, pp. 91–96.

[9] BREIDENBACH, L., DAIAN, P., JUELS, A., AND SIRER, E. G. An in-depth look at the Parity multisig bug, Jul. 2017. `http://hackingdistributed.com/2017/07/22/deep-dive-parity-bug/`.

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

[11] BUTERIN, V. Hard fork completed, Jul. 2016. `https://blog.ethereum.org/2016/07/20/hard-fork-completed/`.

[12] BUTERIN, V. Thinking about smart contract security, Jun. 2016. `https://blog.ethereum.org/2016/06/19/thinking-smart-contract-security/`.

[13] CHEN, L., AND AVIŽIENIS, A. N-version programming: A fault-tolerance approach to reliability of software operation. In *Fault-Tolerant Computing* (1995), IEEE, p. 113.

[14] DAIAN, P. Analysis of the DAO exploit, Jun. 2016. `http://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/`.

[15] DELMOLINO, K., ARNETT, M., KOSBA, A., MILLER, A., AND SHI, E. Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab. In *Financial Cryptography* (2016), Springer, pp. 79–94.

[16] DINGLEDINE, R., MATHEWSON, N., AND SYVERSON, P. Tor: The second-generation onion router. Tech. rep., Naval Research Lab Washington DC, 2004.

[17] ECKHARDT, D. E., CAGLAYAN, A. K., KNIGHT, J. C., LEE, L. D., MCALLISTER, D. F., VOUK, M. A., AND KELLY, J. P. J. An experimental evaluation of software redundancy as a strategy for improving reliability. *IEEE TSE 17*, 7 (1991), 692–702.

[18] ECKHARDT, D. E., AND LEE, L. D. A theoretical basis for the analysis of multiversion software subject to coincident errors. *IEEE TSE*, 12 (1985), 1511–1517.

[19] ETHEREUM. Security considerations. Solidity documentation. `http://solidity.readthedocs.io/en/develop/security-considerations.html`.

[20] EYAL, I., AND SIRER, E. G. Majority is not enough: Bitcoin mining is vulnerable. In *Financial Cryptography* (2014), Springer, pp. 436–454.

[21] HIGH-TECH BRIDGE SA. What's your email security worth? 12 dollars and 50 cents according to Yahoo, Sep. 2013. `https://www.htbridge.com/news/what_s_your_email_security_worth_12_dollars_and_50_cents_according_to_yahoo.html`.

[22] HILDENBRANDT, E., SAXENA, M., ZHU, X., RODRIGUES, N., DAIAN, P., GUTH, D., AND ROSU, G. KEVM: A complete semantics of the Ethereum Virtual Machine, 2017.

[23] HIRAI, Y. Formal verification of Deed contract in Ethereum name service, 2016.

[24] HIRAI, Y. Defining the ethereum virtual machine for interactive theorem provers. In *International Conference on Financial Cryptography and Data Security* (2017), Springer, pp. 520–535.

[25] JAKOBSSON, M., AND JUELS, A. X-cash: Executable digital cash. In *Financial Cryptography* (1998), Springer, pp. 16–27.

[26] JUELS, A., KOSBA, A., AND SHI, E. The Ring of Gyges: Investigating the future of criminal smart contracts. In *ACM CCS* (2016), ACM, pp. 283–295.

[27] KATZENBEISSER, S., AND PETITCOLAS, F. *Information hiding techniques for steganography and digital watermarking*. Artech house, 2000.

[28] KNIGHT, J. C., AND LEVESON, N. G. An experimental evaluation of the assumption of independence in multiversion programming. *IEEE Transactions on software engineering*, 1 (1986), 96–109.

[29] KNIGHT, J. C., AND LEVESON, N. G. A reply to the criticisms of the Knight & Leveson experiment. *ACM SEN 15*, 1 (1990), 24–35.

[30] KOSBA, A., MILLER, A., SHI, E., WEN, Z., AND PAPAMANTHOU, C. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In *IEEE S&P* (2016), IEEE, pp. 839–858.

[31] KRUG, J. A Serpent send exploit, Jun. 2016. http://www.joeykrug.com/home/a-serpent-send-exploit.

[32] LUU, L. PeaceRelay: Connecting the many Ethereum blockchains, Jul. 2017. https://medium.com/@loiluu/22605c300ad3.

[33] LUU, L., CHU, D.-H., OLICKEL, H., SAXENA, P., AND HOBOR, A. Making smart contracts smarter. In *ACM CCS* (2016), ACM, pp. 254–269.

[34] LUU, L., TEUTSCH, J., KULKARNI, R., AND SAXENA, P. Demystifying incentives in the consensus computer. In *ACM CCS* (2015), ACM, pp. 706–719.

[35] MANNING, J. Ether.Camp's HKG token has a bug and needs to be reissued, Jan. 2017. https://www.ethnews.com/ethercamps-hkg-token-has-a-bug-and-needs-to-be-reissued.

[36] MARINO, B., AND JUELS, A. Setting standards for altering and undoing smart contracts. In *RuleML* (2016), Springer, pp. 151–166.

[37] MCKEEMAN, W. M. Differential testing for software. *Digital Technical Journal 10*, 1 (1998), 100–107.

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

[39] MURDOCH, S. J., AND LEWIS, S. Embedding covert channels into TCP/IP. In *Information hiding* (2005), vol. 3727, Springer, pp. 247–261.

[40] NAGY, L., FORD, R., AND ALLEN, W. N-version programming for the detection of zero-day exploits. In *IEEE Topical Conference on Cybersecurity* (2006).

[41] O'LEARY, R. R. Metropolis today: The shifting plans for Ethereum's next big upgrade, Sep. 2017. https://www.coindesk.com/metropolis-today-shifting-plans-ethereums-next-big-upgrade/.

[42] RANDELL, B. System structure for software fault tolerance. *IEEE TSE*, 2 (1975), 220–232.

[43] REDDIT USER "JUPITER0". From the MAKER DAO slack: "today we discovered a vulnerability in the ETH token wrapper which would let anyone drain it.", Jun. 2016. https://www.reddit.com/r/ethereum/comments/4nmohu/.

[44] RO, S. 29 instances of a major world stock market shutdown, Mar. 2014. http://www.businessinsider.com/history-of-world-stock-market-breaks-2014-3.

[45] SOLANA, J. $500K hack challenge backfires on blockchain lottery SmartBillions, Oct. 2017. https://calvinayre.com/2017/10/13/bitcoin/500k-hack-challenge-backfires-blockchain-lottery-smartbillions/.

[46] STEINER, J. Security is a process: A postmortem on the parity multi-sig library self-destruct, 2017. https://blog.ethcore.io/security-is-a-process-a-postmortem-on-the-parity-multi-sig-library-self-destruct/.

[47] SWENDE, M. H. Blockchain frontrunning, Jul. 2017. http://www.swende.se/blog/Frontrunning.html.

[48] SZABO, N. Formalizing and securing relationships on public networks. *First Monday 2*, 9 (1997).

[49] TRAMÈR, F., ZHANG, F., LIN, H., HUBAUX, J.-P., JUELS, A., AND SHI, E. Sealed-glass proofs: Using transparent enclaves to prove and sell knowledge. In *IEEE EuroS&P* (2017), pp. 19–34.

[50] VAAS, L. PayPal refuses to pay bug-finding teen, May 2013. https://nakedsecurity.sophos.com/2013/05/29/paypal-refuses-to-pay-bug-finding-teen/.

[51] VOGELSTELLER, F., AND BUTERIN, V. ERC-20 token standard. Ethereum Improvement Proposal, Nov. 2015. https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20-token-standard.md.

[52] WIKIPEDIA. Monty Hall problem. https://en.wikipedia.org/wiki/Monty_Hall_problem.

[53] WOOD, G. Ethereum: A secure decentralised generalised transaction ledger, 2014.

[54] ZHANG, F., CECCHETTI, E., CROMAN, K., JUELS, A., AND SHI, E. Town Crier: An authenticated data feed for smart contracts. In *ACM CCS* (2016), ACM, pp. 270–282.

A Brief Analysis of Previous Exploits

We briefly justify why various smart contract exploits in Table 1 might 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 could have introduced independence in many cases.

Typos and trivial errors. Some exploits in our analysis are due to trivial programmer errors. For instance in HKG [35] 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 Vyper or Serpent. It is thus impossible that this mistake would have persisted in a multi-language contract.

The mistake in Rubixi [12] 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 Vyper 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 could have prevented these losses.

Re-entrancy. Re-entrancy is a flaw described in [14], 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 [14] and MakerDAO [43]. 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 available 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 Vyper) with regards to re-entrancy is detailed in [31]. 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 hacks [9] [46] involved a bad unforeseen interaction between a `delegatecall`—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 might have similarly contained 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 [45] is similar, with a complex mistake in a blockhash caching system's code by the developer. This caching system was required due to 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 prevented by a Hydra contract. One example in Table 1 is Governmental [12], 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 6.1), as they can be triggered at any time by users simply refusing to provide enough gas.

Another example is FirePonzi [12], 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 [12]—are not covered by our framework as the specification is common to all heads.

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 [17]. 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 [17], we compute the empirical probability $\tilde{P}_N$ that N programs (randomly chosen from the 20) fail simultaneously:

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

Comparing $\tilde{P}_N$ to $\tilde{P}_1$ (i.e., the expected failure rate for NNVP with N heads compared to the failure rate of a standalone program), we find that $30 \cdot \tilde{P}_3 \leq \tilde{P}_1 \leq 5087 \cdot \tilde{P}_3$ 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 $S_{1,0}$), which has the lowest failure rate overall.

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, $\mathtt{gap}$, defined in Section 4 is estimated as the ratio of $\tilde{P}_1$ and $\tilde{P}_N$ (averaged over the six workloads). We obtain $\mathtt{gap} = 4409$ for $N = 3$ and $\mathtt{gap} = 34{,}546$ 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 $\mathtt{gap} = 79$.

C Submarine Commitment Constructions

In this section, we present two constructions for Submarine Commitments. The first, in Section C.1, is our preferred construction. It is simple and efficient, but only realizable with changes to Ethereum awaiting adoption of EIP-86. The second, in Section C.2 is more involved and expensive, but realizable today.

We note that players could in principle conceal true commitments by sending dummy commitments with random values $\$\mathrm{val} \geq \mathit{deposit}$—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. These problems motivate the use of Submarine Commitments instead.

C.1 Submarine Commitments via Contract Creation.

Our simple realization of Submarine Commitments in Ethereum leverages a new Ethereum Virtual Machine (EVM) opcode, `CREATE2`, introduced in 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 encode the inputs to the commit, key and $addr(P)$. 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.2 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") [41]. More details are in Appendix C, including a less efficient alternative scheme that is realizable in Ethereum today.

C.2 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 [32]: all transactions in a block are organized in a Merkle-Patricia Tree [53] 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 2.08 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

```
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 $\widehat{\text{addr}}$
```

Figure 7: Algorithm to create a Forwarder at address $\widehat{\text{addr}}$.

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 the $bounty) and $\mathcal{A}$'s $deposit is given to P.

C.3 CREATE-Based Construction

In Section 5.3, 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% lower gas costs than the CREATE-based construction during deposit collection (75,000 gas vs 5,000,000 gas, or 0.75 USD vs 50.00 USD respectively[2]).

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 $\mathcal{E} : \{0,1\}^{\ell} \rightarrow \{0,\ldots,b-1\}^{k}$ 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

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

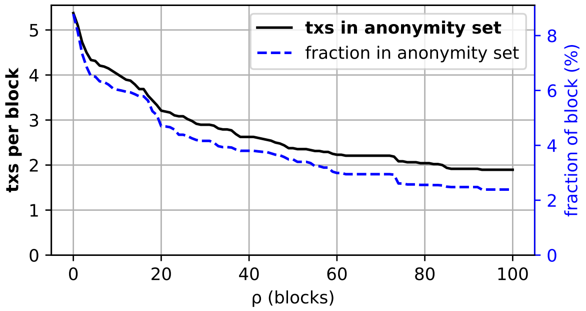

Figure 8: Size of anonymity set for Submarine Commitments. We show the number of transactions (left) and the fraction of transactions (right) per block that are a part of the anonymity set, as a function of ρ, the size of the commit window. Statistics are computed by averaging 48 block sequences of length ρ, starting at (hourly-spaced) blocks $4430000 + i \cdot 240$ for $i \in [0, 47]$.

- *Reveal:* P sends key and a Merkle-Patricia proof that she committed in the correct block (see Appendix C.2) to BountyContract.
- *DepositCollection:* BountyContract repeatedly calls the *Clone* function of appropriate Forwarder instances until a Forwarder is created at address $\widehat{\text{addr}}$. (See Figure 7 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. In our prototype implementation, a *DepositCollection* call costs 5,000,000 gas with these parameters.

C.4 Analysis of Anonymity Set Size

The Submarine Commitment constructions from Section 5.3 and Appendix C.3 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:

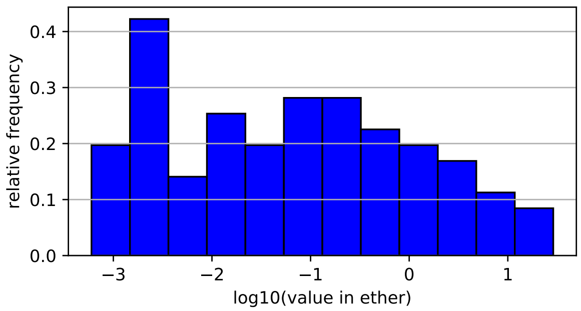

Figure 9: Histogram of transaction values in anonymity set for Submarine Commitments. We set $\rho = 100$ and take all transactions in the anonymity sets of 48 sequences of 100 blocks, starting at blocks $4430000 + i \cdot 240$ for $i \in [0, 47]$.

- 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 $\mathbf{Exp}_{\mathcal{A}}^{\mathsf{bntyrace}}$ analyzed in Section 5.4, a commitment is revealed after ρ blocks, where it is assumed that the adversary can rewind up to ρ blocks in the blockchain. Figure 8 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 9 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 10,000 USD^2) are easily concealed.

D Security Model and Proof Details

In this appendix, we present the detailed ideal function $\mathcal{F}_{\mathsf{withhold}}$. Within the $\mathcal{F}_{\mathsf{withhold}}$-hybrid model, we prove Thm. 3 of Section 5.4. We also extend our analyses beyond pure front-running adversaries.

D.1 Ideal Functionality $\mathcal{F}_{\mathsf{withhold}}$

The ideal functionality $\mathcal{F}_{\mathsf{withhold}}$, shown in Figure 10, supports three functions: "post", "add block", and "rewind". We assume a system-wide transaction buffer Mempool, from which transactions are selected for mining. 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 $\mathbb{B}$ by adding a new block that includes transactions from Mempool. The function "rewind" allows $\mathcal{A}$ to remove blocks from $\mathbb{B}$. This capability may seem redundant, as $\mathcal{A}$ controls the blocks added to $\mathbb{B}$. $\mathcal{A}$ may, however, wish to make retroactive modifications based on information in a fresh transaction submitted by P_0.

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

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

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

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

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

Figure 10: Ideal functionality $\mathcal{F}_{\mathsf{withhold}}$ for (δ, ρ)-adversary.

The function $\mathsf{ValidTx}(\tau; \mathbb{B}, \mathsf{Mempool})$ 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 $\mathbb{B}$ or Mempool. Thus any transaction in a set of valid transactions is unique.

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 $\mathbb{B}$. P_0 does not observe Mempool in our model, although variant models are possible of course.

Submarine Commitments in $\mathcal{F}_{\mathsf{withhold}}$. To model Submarine Commitments in $\mathcal{F}_{\mathsf{withhold}}$, we let players send a special message ("submarine-post", τ) 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 $\mathbb{B}$. Thus $\mathbb{B}$ can contain transactions that $\mathcal{A}$ does not see as it is constructing blocks. $\mathcal{A}$ cannot delay such messages but can still rewind $\mathbb{B}$ to evict them once the commitment is revealed.

D.2 Pure Front-Running Adversary.

Recall that $\mathsf{commblock}_{P^*} \leftarrow_\$ [1,n]$ in $\mathbf{Exp}^{\mathsf{bntyrace}}_{\mathcal{A}}$, 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 4. *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 5. *If* $\mathsf{commblock}_{P^*} < i$*, then* $Z_i = 0$.

Proof. Committing in block B_i will not enable $\mathcal{A}$ to front-run 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 6. $\Pr[Z_i = 1] \ge \Pr[\mathsf{commblock}_{P^*} \ge i \wedge X_i = 1]$.

Proof. This is an immediate corollary of Claim 4: If $\mathsf{commblock}_{P^*} \ge i$ and $X_i = 1$, then we have $Z_i = 1$. For events A, B with $A \implies B$ we have $\Pr[B] \ge \Pr[A]$. □

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 \$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 7. *Suppose for a given* $\mathcal{A}$ *that* $q = \Pr[X_{i+k} = 1 \wedge X_i = 1] > 0$ *for some* $1 < k < \Delta$ *and* $i + k \le n$*. Then there exists an adversary* $\mathcal{A}'$ *such that* $\mathbb{E}_{\mathcal{A}'}[\$\mathrm{cost}] < \mathbb{E}_{\mathcal{A}}[\$\mathrm{cost}]$ *and* $p_{\mathsf{wins}}(\mathcal{A}') \ge p_{\mathsf{wins}}(\mathcal{A})$.

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 front-run, then $\mathcal{A}'$ can also front-run P^*. Thus $p_{\mathsf{wins}}(\mathcal{A}') \ge p_{\mathsf{wins}}(\mathcal{A})$.

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 \$cost at least \$deposit less than $\mathcal{A}$. Thus, $\mathbb{E}_{\mathcal{A}}[\$\mathrm{cost}] \ge \mathbb{E}_{\mathcal{A}'}[\$\mathrm{cost}] + \frac{q \cdot \$\mathrm{deposit}}{n}$. □

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

Proof. Recall that $\mathcal{A}$ is a pure front-running 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,

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

using Claim 4 and independence of X_i and $\mathsf{commblock}_{P^*}$. □

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

Lemma 9. $\mathbb{E}[\$\mathrm{cost}] \ge \$\mathrm{deposit} \cdot \sum_{i=1}^{n} \left(\frac{n-i+1}{n}\right) p_i$.

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

$$\mathbb{E}[\$\mathrm{cost}] = \sum_{i=1}^{n} \mathbb{E}[\$\mathrm{cost}_i] = \sum_{i=1}^{n} \$\mathrm{deposit} \cdot \Pr[Z_i = 1]$$

$$\geq \$\text{deposit}\cdot\sum_{i=1}^{n}\Pr\left[X_i=1\wedge C_i\right]$$

$$=\$\text{deposit}\cdot\sum_{i=1}^{n}\Pr\left[X_i\right]\cdot\Pr\left[C_i\right]$$

$$=\$\text{deposit}\cdot\sum_{i=1}^{n}\left(\frac{n-i+1}{n}\right)p_i\,,$$

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

Let us restate Theorem 3:

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

Proof. By Claims 4, 5 and 6, $\mathcal{A}$'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 $\mathbb{E}[\$\text{cost}]$.

Let $p=\sum_{i=1}^{n}p_i$. By Lemma 8, $p_{\mathsf{wins}}\leq\frac{(\Delta+1)\cdot p}{n}$.

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 9 states that

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

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 7, 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:

$$\mathbb{E}[\$\text{cost}]\geq\$\text{deposit}\cdot\sum_{i=1}^{k}\left(\frac{n-\Delta(i-1)+1}{n}\right)$$

$$=\$\text{deposit}\cdot\left(k-k/n-\frac{\Delta(k-1)(k-2)}{2n}\right)$$

$$>\$\text{deposit}\cdot\left(k-\frac{k^2}{2n}\right)$$

$$=\$\text{deposit}\cdot\frac{np_{\mathsf{wins}}}{\Delta+1}\left(1-\frac{p_{\mathsf{wins}}}{2(\Delta+1)}\right)$$

$$\geq\$\text{deposit}\cdot\frac{9np_{\mathsf{wins}}}{10(\Delta+1)},$$

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

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

The theorem follows. □

D.3 Extension: α-Revealing Adversary

In Section 5, we consider an adversary trying to maximize its profit in $\mathbf{Exp}_{\mathcal{A}}^{\mathsf{bntyrace}}$. 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 $\mathbf{Exp}_{\mathcal{A}}^{\mathsf{bntyrace}}$ from delaying disclosure of its bug. So $\mathcal{A}$ would like to emit a "reveal" message as late as possible, to maximize its withholding period. For such an adversary, a natural strategy is 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 3 immediately yields the result:

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

Of course, the space of strategies for an economically rational adversary $\mathcal{A}$ is a superset of α-revealing. $\mathcal{A}$ 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-revealing. 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^*, something we now consider.

D.4 Extension: Exponential Distributions

Finally, we sketch a proof of an analog Theorem 3 where $\mathsf{commblock}_{P^*}$ follows an exponential distribution rather than a uniform one. From our Poisson model of bug finding in Section 4, 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 $\Pr\left[\mathsf{commblock}_{P^*}\in[i,i+\Delta]\right]$ in the proof of Lemma 8 and $\Pr\left[\mathsf{commblock}_{P^*}\geq i\right]$ in the proof of Lemma 9. For the exponential distribution, we get

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

$$\Pr\left[\mathsf{commblock}_{P^*}\geq i\right]=e^{-i\cdot\lambda_{\mathrm{H}}}\,.$$

Therefore, the analogs of Lemma 8 and Lemma 9 are:

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

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

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

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

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