The Hydra Framework for Principled, Automated Bug Bounties

Lorenz Breidenbach Philip Daian Florian Tramèr Ari Juels

Abstract

We present the Hydra Framework, a new principled approach to modeling and detecting security-critical bugs. By fusing a variant of classical N-version (redundant) programming with automated bug-bounty payouts, the Hydra framework is particularly appealing as a way to provide economically rigorous and cost-effective bounty protections for smart contracts.

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 monetary rewards for hackers to disclose software bugs privately so they can be patched before exploits arise.

Bug bounty programs today, though, have three basic flaws. First, bounties often fail to incentivize disclosure: Hackers can profit more from exploiting critical vulnerabilities or selling them on underground markets. Second, software vendors find it difficult to calculate bounty amounts in a well reasoned way, and often set them arbitrarily. Finally, there is a fair-exchange problem: Vendors sometimes renege on bounty payments or pay unfairly low bounties, thereby undermining community-wide bounty incentives.

We introduce the *Hydra Framework*, the first principled approach to bug bounties that addresses these challenges. It deters economically rational actors, including hackers, from exploiting bugs or selling them in underground markets. Its key idea is to *transform programs* so that bugs are *easy to detect, but hard to exploit*. One way to do this is to run differently coded versions of the same program in tandem. A bug is likely to affect some, but not all program versions. Diverging behavior among versions therefore reveals the bug, but makes it hard to exploit.

The Hydra Framework combines such program transformation with bug bounties. A hacker that discloses a bug causing program versions to differ in behavior receives a bounty *automatically*. As a result, economically rational hackers can hunt for bugs with the assurance of fair rewards for disclosure. Hackers are also incentivized to disclose bugs for modest bounties: Our analysis shows that competition for bounties makes it better to claim a bounty than to wait and try to weaponize a bug as a full-blown exploit.

It is well known that people do not always conform perfectly to a model of economic rationality, specifically in behaviors surrounding security vulnerabilities. Nonetheless, present bounties often assume hackers will be altruistic, or that fears of, e.g., external legal consequences, will motivate them to behave honestly. While these assumptions sometimes hold, economically motivated exploitation of important systems remains common. For example, Zerodium, one grey-market exploit dealer, openly offers seven figure sums for certain exploits. We therefore seek security in the stronger model of *economic rationality*. While this model has its limits, assuming attacker behavior that maximizes revenue is considerably stronger than assuming honest or legally compliant attackers.

The Hydra Framework is especially appealing for *smart contracts*, which often have high exploitable monetary value and a native ability to make automated bounty payouts. Smart contracts also naturally fit with a security model assuming economic rationality, a common assumption in

their design. We show how Hydra provides automatic payoff, without the need to trust any third party arbiters for payment or bounty value. This property is a key requirement for economically rational bounties, as finding a neutral arbiter trusted by both the attacker and contract author would in many cases be challenging or infeasible. The fact that smart contracts are explicitly designed to remove trusted third parties makes them an ideal deployment vehicle for the Hydra Framework.

2 Overview of the Hydra Framework

To solve these fundamental issues, the key idea of Hydra is to transform a vulnerable program into a program that has an *exploit gap*. An exploit gap is a gap between when an error is found in a program and when the program fails to act as expected. This corresponds to an increase in difficulty exploiting the program. In programs with high exploit gaps, an input or attack that a program fails to process is less likely to be a weaponizable exploit.

One practical example of an exploit gap uses an old technique from fault tolerance, previously explored for development of safety critical systems like spacecraft, called N-version programming. N-version programming deploys multiple versions of a program that are independently developed by distinct programmers.

In N-version programming, for every step, programs perform a majority vote to determine the output. If one version of a program contains an error, the remaining correct programs continue unaffected. Hydra extends this notion to *N-of-N-version programming* (NNVP). Instead of a majority vote, NNVP requires a *unanimous vote*, i.e. all program versions must agree on the output at every step. In the case of any disagreement, NNVP halts the program and invokes a fault manager. In Hydra's version of NNVP, we call each individual program a *head*, and the transformed new program the *Hydra program*. The Hydra program contains some additional code that synchronizes and coordinates the actions of the heads, which we call a *meta-program*.

Figure 1 shows an example of an NNVP-based Hydra program, represented by each outer green box. When the program receives an input X, it simulates the execution of "head" programs f_1, f_2, f_3 on X, and makes sure their outputs agree. If all the programs agree an output should be Y, Y is output (the case on the left). Otherwise, if for example head f_3 contains a bug and outputs some incorrect Y', and doesn't agree with other heads, a bounty is paid to whatever user discovered the exploit X, and the Hydra program is aborted (on the right).

We can mathematically define and measure the exploit gap of such a program. Informally, to do this, we take the probability an input X (as in the figure) will cause *any* head to output an undesired value, and divide it by the probability that X will cause the Hydra program composed of the heads to output an undesired value. If an exploit gap is 1000, a sampled exploit input X is 1000 times more likely to cause unwanted behavior in some head than to cause unwanted behavior in the Hydra program, which requires all heads to agree on the unwanted behavior. Although NNVP is one way of achieving an exploit gap, we discuss a number of other possible techniques in [BDTJ17].

Exploit gaps are highest in programs in which heads are independent of each other, and have some source of differing behavior on unexpected or unspecified inputs. A set of programs with correlated failures would be more likely to break in concert for any given exploit, while a set of programs with independent failures would be less likely to see all heads break identically. As a first use case for the Hydra, we focus on smart contracts, small computer programs that run on a blockchain and handle financial transactions. Smart contracts are a good experimental testbed for the Hydra, as they are often exploited for millions of dollars, with three famous hacks each exceeding USD100M in damage to honest users.

Smart contracts provide new properties helpful to securing them against economically rational attackers. These include:

1. *Language diversity*: Popular platforms allow development in multiple interoperable languages, often with starkly different designs. This provides a source of failure independence.

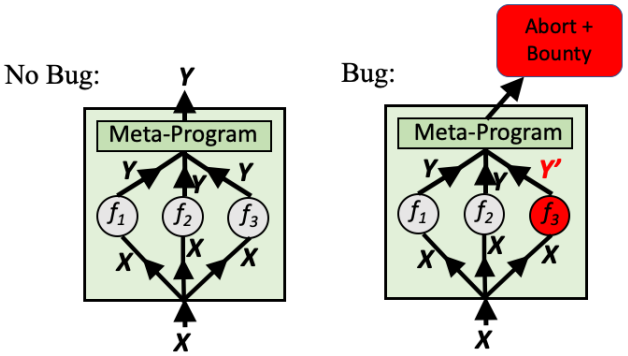

Figure 1: Hydra program with heads f_1, f_2, and f_3. Example on right shows effect of bug induced by input X in f_3.

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

Table 1: Selected smart contract failures and potential exploit gaps. The list is extended from [HSZ+17]. For each incident, we report the value of affected funds (data from Coinmarketcap). More details and analysis are in the extended version of this paper [BDTJ17]. Among others, a prominent bug in the DAO contract—which resulted in a loss of 150M USD—was due to language-specific anti-patterns and programmer mistakes that a Hydra contract might have prevented

.

2. *Payment systems*: The underlying blockchain infrastructure can be used to ensure fair and transparent bounty payments.

3. *Economic viability*: Implementing multiple heads is easier for smart contracts than traditional systems. Smart contracts tend to be short and valuable programs (some popular contracts hold up to 1M USD per line of code).

4. *Economic objectivity*: There is a precise value of all assets held by a contract, so an exploit's damage can be measured for precise pricing of bounties.

Table 1 enumerates previous cases of high profile smart contract failures, and notes whether applying the Hydra framework would have very likely prevented the issue (the items in green). In many prominent cases, exploits and the resulting loss of funds were not due to "logic" errors in the contract's intended functionality, but rather due to avoidable programmer mistakes or language anti-patterns that a Hydra contract could have addressed. In many cases, merely using different programming languages for each of the heads would change the behavior of the program under buggy inputs, leading to an exploit gap.

We have implemented the Hydra Framework for Ethereum, available at `thehydra.io`. It includes a full framework for developing Hydra contracts, tests, live bounties, and code examples.

3 Embracing the Gap - Hydra Economics

When evaluating the viability of any cybersecurity technique, a key question is whether its benefit justifies the cost of implementation for a given software target. The classic idea of N-version

programming has previously seen only limited deployments in security-critical systems, and is used exclusively on high value targets where failure is an option to be avoided at *any* cost.

N-version programming is often overlooked because it makes an assumption that component programs have weakly correlated failures [CA95]. If failures are correlated, it is somewhat likely that a majority vote will fail to agree on a correct value. This analysis, expanded on in [KL86, ECK+91], has caused N-version programming to be deemed uneconomical for most programs.

In the Hydra Framework, however, we relax the need for fault tolerance and instead focus on error detection and graceful termination of a faulty system. In scenarios where N-version programming was first considered—such as space shuttle flight systems—availability is a crucial feature: the system cannot afford to simply give up and shut down if a failure is detected in one of its components. This requirement of always producing a response, despite possible disagreements between components, makes N-version programming more likely to produce erroneous behaviors.

For financial systems such as smart contracts, however, producing *no* response is often more acceptable than an incorrect one. Stock markets, for example, carry built-in circuit breakers that stop trading if certain unusual activities or price changes are detected. By sacrificing full availability, we can extract far more value from (partially) correlated failures than with a majority vote. Since N-of-N-version programming (NNVP) requires a unanimous vote rather than a majority vote, we ensure that an incorrect behavior will only be triggered if *all* system components fail in the exact same way. This sacrifice of *availability* for additional *safety* is the key to NNVP's efficacy.

Past smart contract failures have also revealed a preference for safety over availability. For instance, when attackers found an exploit in the *Parity Multisig Wallet* [BDJS17] and started stealing 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.

A concrete analysis of our NNVP technique reveals its promise. We revisited the experimental setting of Eckhardt et al., which is often used to showcase the limited benefits of N-version programming [ECK+91]. Eckhardt et al., had found that for three program replicas ($N = 3$), majority voting only reduced the probability of certain faults by a factor of 5. Using NNVP instead, we find that faults are at least 75× less likely to occur in all three replicas simultaneously than in a strict subset of them. Thus, despite having correlated failures, the programs analyzed by Eckhardt et al. result in large exploit gaps. The true exploit gaps are probably even larger, as Eckhardt et al. did not distinguish whether correlated failures are identical or not (that is, if two programs both give different, wrong answers Eckhardt et al. consider this a correlated failure). In NNVP, a failure only occurs if all N versions produce *the same incorrect output*.

Programs with large exploit gaps, and with the ability of being gracefully shut down, exhibit an interesting economic property when combined with a bug bounty scheme, as in the Hydra. If an attacker finds a bug that compromises one head, they cannot profit from the vulnerability except by collecting the bounty. An attacker could forfeit the bounty, and continue searching for full exploits that affect all Hydra heads and therefore the full program. Yet in doing so, they risk having another user find and report a vulnerability, thereby collecting the bounty and shutting down the contract. Furthermore, a bug bounty provides an economic incentive not just for "black-hats" (i.e., malicious parties looking for exploits), but also for "white-hats" who only wish to disclose bugs. One result of a bug bounty therefore is more thorough communal scrutiny of a given program. Another is increased likelihood that bugs are found earlier rather than later in a system's life-cycle, when usage and costs associated with the system becoming unavailable are typically lower. Finally, experience shows that exploits frequently combine different buggy behaviors or are highly refined versions of simple bugs. This is borne out by many real-world bug bounty programs paying for security-relevant bugs, even if those bugs on their own are not exploitable.

Formalizing this economic analysis reveals a principled strategy for pricing bug bounties that incentivize economically rational attackers to disclose vulnerabilities. Intuitively, for programs with high exploit gaps—i.e., a discovered bug is very unlikely to affect all Hydra heads simultaneously—bounty amounts can be much lower than the value of a full exploit while still providing positive

incentives for early disclosure of vulnerabilities.

4 Economic Analysis of Hydra Bounties

By analyzing the exploit gap introduced by the Hydra contract, we derive a principled bug-bounty pricing model that incentivizes bug disclosure.

Hereafter, we assume that a party that finds a bug in a Hydra head can instantaneously collect the bounty. In this setting, our economic analysis reduces to an argument about the relative difficulty of finding a bug against a single Hydra head or against all heads simultaneously (i.e., an *exploit gap*). In the next section, we'll consider a refined analysis in the blockchain model, wherein a bug-finding party must first submit their bounty-claims to a smart-contract. As we'll see, when faced with an adversary with the ability to reorder network messages sent to smart contracts, incentivizing bug disclosure requires extra care.

Analyzing economic incentives. Our analysis considers a set of parties that search for bugs in a Hydra contract. If a bug affecting one or more of the Hydra heads is disclosed, the disclosing party is awarded a bounty and the Hydra contract is aborted. We pessimistically assume that any exploit against the full Hydra contract can be used to steal the contract's entire balance.

We distinguish two sets of parties. First, there are honest (i.e., "white-hat") parties that disclose any bug they find in one or more of the Hydra's heads. Second, there are maliciously inclined (i.e., "black-hat"), yet economically rational adversaries that will adopt whichever strategy is most profitable. That is, if an adversary finds a full exploit, it will always use it to steal the balance (assuming that the contract's balance exceeds the bounty, as is the case in any traditional bug-bounty scheme). The interesting case to analyze occurs when a malicious party discovers a partial bug that could be exchanged for a bounty but cannot be used to steal the contract's balance. If the party decides not to disclose the bug, and instead continue searching for a full exploit, she risks losing her guaranteed bounty payout if another party claims the bounty first. Thus, as long as the likelihood of finding a full exploit is somewhat small compared to that of any honest party finding a bug, even small bounties (relative to the contract's balance) hold the potential to incentivize bug disclosure.

In [BDTJ17], we formalize the above intuition by modeling bug finding as a Poisson process, which captures each party's work rate towards finding bugs. In this model, our exploit gap definition naturally surfaces as the ratio between the expected time until a party discovers either a full exploit or a non-exploitable bug in one of the Hydra heads.

Assuming independent program failures, we show that this exploit gap grows exponentially in the number of Hydra heads. More generally, the exploit gap can be estimated from empirical statistics on program failures. In particular, by revisiting the experiments of Eckhardt et al. [ECK+91] on classical N-version programming, we found that NNVP induces significant exploit gaps even with program failures exhibit strong correlations. For the programs they tested, a Hydra contract with three heads would exhibit bugs that are on average 4,400 times less likely to affect all heads simultaneously than a strict subset of the heads.

We then derive a simple formula for setting an appropriate bug bounty, as a function of the contract's balance, the exploit gap, as well as the relative work rates of honest and malicious parties. Intuitively, the higher the exploit gap or the work rate of honest parties, the more likely it is that a malicious party sitting on an undisclosed bug will be preempted by an honest bounty claim. For example, under the very conservative assumption that the malicious party's work rate is equal to the *combined* work rate of all honest parties, we derive a minimal bounty value that is a fraction $\frac{1}{1+\texttt{gap}}$ of the contract's balance (where `gap` is a lower-bound estimate on the exploit gap). When plugging in the empirical program failure rates observed in the experiment of Eckhardt et al. [ECK+91], we find that *a three-headed Hydra could sustain a bounty 3 to 4 orders of magnitude below an exploit's value.*

Our analysis also provides insight into why bounties are paid when bugs are not necessarily actively exploitable against the target system. If the bounty is too small, all economically rational

players will attempt to privately weaponize any partial exploits they develop. Traditional bounties operate off similar intuition, with tiers of exploit values to boost participation (e.g. [Eth18]).

5 The Bug-Withholding Problem

Our analysis in the previous section 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. In this model, we highlight a powerful *bug-withholding* attack and propose a defense called *Submarine Commitments*.

Front-running. Transactions sent to a blockchain such as Ethereum are public and need not be processed by a smart-contract in the exact same order as they were submitted to the network. This introduces two vulnerabilities for bug-bounties.

First, by monitoring network communication to the Hydra smart-contract, an adversary could observe users' bounty-claim transactions and immediately construct a bounty-claim of her own by simply copying the submitted bug. To get her transaction to be processed first (and thus collect the bounty), the adversary can ensure faster network propagation of her transaction or prioritize its processing in the underlying blockchain, e.g., by paying higher transaction fees or corrupting a miner. Such transaction re-ordering is commonly referred to as a *front-running attack* [Swe17]. Many of these attacks, including the one described above, can be thwarted using a *commit-reveal* scheme (see below) wherein transaction contents are temporarily hidden from other users.

Second, front-running opens up bug bounty systems to more pervasive *bug-withholding attacks*, that require more sophisticated defenses. Suppose an adversary has already found a non-exploitable 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. By front-running, though, the adversary can ensure it claims the bounty first, thus nullifying any economic incentives for early disclosure. Note that to mount this attack, the adversary need not observe the *contents* of a bounty-claim, but solely its *existence*.

Submarine Commitments. Front-running is a well-known attack vector against smart-contract-based auctions, lotteries, decentralized exchanges, among others [Swe17]. A simple countermeasure to many such attacks is to use a *commit-reveal* scheme. At a high level, users submit transactions (e.g., a bid in an auction) in two phases. First, users cryptographically *commit* to a transaction in a way that binds the transaction's content yet hides this content from other users. After a fixed delay (so that all committed transactions are guaranteed to have been transmitted over the network), users *reveal* the committed transaction contents which are then processed by the smart-contract. Intuitively, commit-reveal schemes prevent the first form of front-running described above because when the contents of a bounty-claim are finally revealed, a user wishing to copy the bug would not have been able to submit a commitment earlier in time.

This knowledge asymmetry introduced by cryptographic commit-reveal schemes is insufficient to thwart the stronger bug-withholding attacks. Indeed, the bug-withholding adversary need not observe the *contents* of a bounty-claim transaction, but just the *existence* of such a transaction. With a naïve commit-reveal scheme, the adversary can continue searching for a full exploit until she observes a user commit to a bug. At that point, the adversary can also commit to her own bug and later reveal it to claim the bounty.

We introduce a novel bug-withholding defense called a *Submarine Commitment*. This is a powerful, general solution to the problem of front-running that is 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 that temporarily hides the very existence of a transaction. Achieving Submarine Commitments is challenging in systems like Ethereum, however, because message contents and currency in all transactions are *in the clear*.

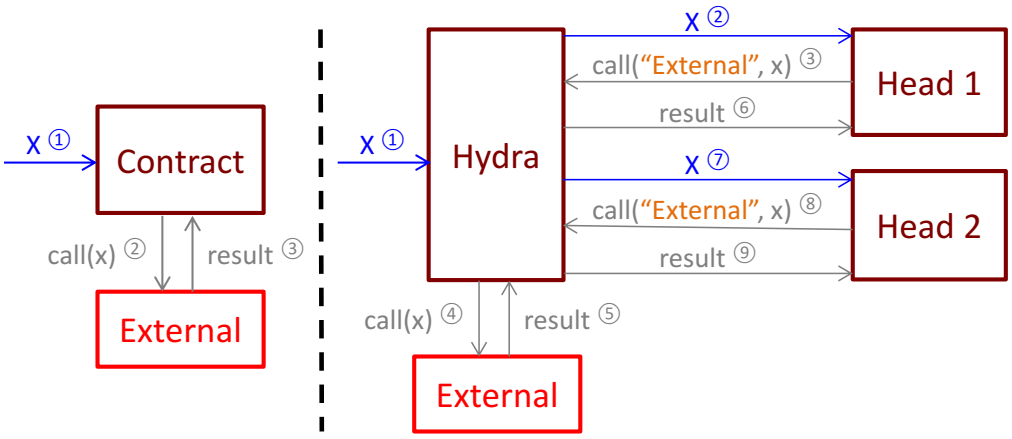

Figure 2: The Hydra NNVP transformation. (Left) A smart contract is invoked with input X and calls an external contract. (Right) A Hydra contract with two heads. The meta-contract acts as a proxy and delegates the incoming call with input X to each head. Due to the sequential nature of the Ethereum Virtual Machine, each head is executed in turn. Calls to external contracts are routed through the meta-contract and executed only once, with the obtained result being cached and replayed for each additional head.

We refer the interested reader to [BDTJ17] for a technical description and analysis of Submarine Commitments. Briefly, a user commits to a bounty-claim transaction by sending a commitment to the transaction contents, and a small amount of cryptocurrency, to a specially-constructed account. This account is designed to satisfy two properties: (1) it appears random to the adversary (who thus cannot infer that this particular transaction is actually a commitment to a bounty-claim); (2) the user can later reveal that this account (and the currency sent to it) is "owned" by the Hydra contract, at which point the transaction is allowed to be processed. A Submarine Commitment's small (and tunable) cost prevents the bug-withholding adversary from continuously committing to her bug, and thus forces her to "guess" when another user will issue a valid bounty claim.

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 full implementation is available under an open-source license at `thehydra.io`.

Ethereum smart contracts are small programs (a typical contract weighs a few hundred lines of code) that are stored on the blockchain and executed by a specially designed virtual machine, the Ethereum Virtual Machine (EVM) [Woo14]. The EVM execution model is *Turing complete*, *sequential*, *deterministic*, and *single-threaded*. Contracts can perform computation, access ephemeral memory that disappears after execution completes, read and write permanent on-chain storage, and interact with other contracts exchanging data and/or currency. Ethereum uses an economic approach to prevent a contract's execution from consuming too many computational resources: Each EVM operation has an associated *gas cost* that is charged to the transaction's originator.

As discussed in Section 2, a Hydra contract consists of multiple "head" programs and a *meta-contract* that coordinates and supervises the execution of the heads. The key implementation challenges are to design the meta-contract and to statically instrument the heads so that the meta-contract can supervise them.

Figure 2 shows the high-level architecture we implemented for the Ethereum variant of the Hydra. Normal operation is shown on the left: contracts interact directly with an external untrusted environment through function calls, and an input X is provided by an Ethereum transaction. Operation with the Hydra framework is shown on the right: all environment operations are mediated and cached by the Hydra meta-contract. Each head's output is compared to ensure the

same result is emitted regardless of how the head is implemented. A contract's external operation is unchanged by use of the Hydra, except for increased computational costs incurred by this redundancy.

There are several engineering and research challenges relevant to deploying the Hydra:

Implementation of the meta-contract. The meta-contract acts as the central point for any users or contracts that wish to interact with the Hydra contract. Hence, a Hydra contract can act as a stand-in replacement for a "normal" contract. Users don't need to be aware that they are interacting with a Hydra contract, a crucial property for the adoption of the Hydra framework.

The meta-contract delegates all incoming calls to each head, and verifies that the obtained outputs match. If so, it returns that output. Otherwise, it uses a special feature of the EVM to revert any changes made during its execution.

The meta-contract also acts as a receiver for any bounty claims. It verifies any claim by running the heads on the associated input and checking that their outputs *do not* match. If the bounty claim is valid, it will revert any changes made during the execution of heads, pay out the bounty and enter an escape-hatch mode.

Instrumentation of the heads. 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, simply checking that the heads' outputs match in the meta-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 meta-contract must thus guarantee consistency among the heads.

To resolve these issues, the heads are *instrumented* prior to deployment so that all interactions with the blockchain are mediated by the meta-contract. This allows the meta-contract to verify that not just the heads' outputs but all of their interactions with the blockchain match. 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).

TCB size. Our design is generic, and covers the vast majority of contracts in use on Ethereum today (see Section 7). The meta-contract (written in EVM assembly) and the instrumenter for Hydra heads (written in Haskell) consist of 1500 lines of code. It should also be relatively easy to write a formal specification for the simple functionality of the meta-contract and instrumenter, although we have not attempted this.

Escape hatches. Ideally, bugs could be patched *online*. This is hard in Ethereum as smart contract code cannot be updated after deployment [MJ16]. Best practices [Eth] 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.

7 Evaluation

We test the soundness, applicability, and performance of our implementation of the Hydra framework for smart contracts on multiple workloads: First, our implementation passes all supported tests of the official EVM test suite lending credence to its soundness. Second, to study our implementation's applicability, we examine all contracts invoked on Ethereum during a sample period: Our implementation cannot deal with all contracts used on Ethereum, e.g. because they rely on their exact bytecode representation and are thus not amenable to instrumentation or because they depend on rarely used functions of the EVM which we haven't implemented. In practice, however, our implementation is applicable to the vast majority of contracts: 76% of all contracts invoked between December 7^{th} 2017 and February 7^{th} 2018 are supported. For instance, this includes most contracts compliant with the ERC20 token standard, described below, and which collectively hold billions of dollars in value. Finally, we developed and evaluated two representative smart contracts:

- *The Hydra ERC20 token:* ERC20 is the most widely adopted API standard for token contracts, which are the most popular type of contract on Ethereum. Tokens, similar to shares in a corporation, are units of value whose ownership is typically associated with certain rights or privileges, and are commonly used for voting, paying for services and as important building blocks for complex cryptoeconomic mechanisms. The ownership of the tokens and the associated privileges are tracked by the token contract. As of December 2018, the combined market cap of the top ten such contracts deployed on Ethereum exceeds 2 billion USD. Our three-headed Hydra ERC20 contract is deployed on the main Ethereum network and is funded with an initial bounty of 3000 USD. It could be used as a drop-in replacement for any ERC20 contract.

- *A Hydra Monty-Hall lottery:* In this popular 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 a fixed number of 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.

  One of the authors was tasked with writing 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 before the game completes, 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.

For both of these representative applications, three authors independently developed one Hydra head in each of *Solidity*, *Serpent*, and *Vyper*, the main programming languages in Ethereum at the time. To develop these versions, the authors agreed on a high-level text specification and interface, wrote code and tests, and combined their test suites to test all three programs differentially against each other. This is only one possible development methodology naturally compatible with Hydra. These languages have different design tradeoffs (in terms of ease-of-use, low-level features and security) and are by themselves a valuable source of diversity between our Hydra heads.

Gas costs. Running mulitple copies (i.e., heads) 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 many common, small workloads, the main gas cost of a transaction is the fixed "transaction base fee". As our Hydra meta-contract calls all the heads sequentially in a single transaction, this fee is amortized, leading to *sub-linear* scaling of the gas-cost as the number of Hydra heads increases.

Lessons learned from 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 a certain EVM anti-pattern that was discovered concurrently to our work. 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 game, ensuring compatibility between heads required writing a detailed specification, which revealed many blind spots in our original design. Moreover, *differential testing* [McK98] (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 Conclusion

The Hydra Framework lends three major insights into the future of securing software. Firstly, smart contract systems present a novel environment for reasoning about security, with convenient properties that are rare in traditional software. The often precise and unambiguous value of smart contracts is in stark contrast to most real-world programs, whose value is hard to reliably estimate. The smart-contract payment infrastructure also helps construct direct incentives that influence attacker behavior. We therefore expect smart contracts to be an increasingly important platform for studying novel security techniques.

Secondly, techniques such as N-version programming —that are often dismissed as ineffective for fault tolerance—may prove more useful for applications with relaxed availability requirements. In these applications, safely shutting down a system in the face of a plausible bug is a viable and often desirable option. We predict that many techniques from fault tolerance can be similarly repurposed to create an exploit gap, and hope to explore the possibility of applying such techniques to the Hydra Framework. We expect security measures tailored to systems with relaxed availability requirements to differ significantly from those traditionally deployed in availability-critical systems.

Lastly, and most surprisingly, bug bounty programs built on top of our framework can increase application security in a clear model of economic rationality. For moderate bounties, disclosing a non-critical vulnerability has higher expected payout than searching for a full exploit. Our analysis obviously has limits: malicious actors do not always behave according to models of rationality, and external incentives are hard to model. Nonetheless, this form of economic security is promising, and provides more confidence than assuming attackers will honestly and altruistically report zero days in exchange for bounties. We encourage future systems to adopt such formal economic models of behavior as core to their security model.

There are several open questions and research directions towards proving the efficacy of today's Hydra Framework. We would like to measure the level of independence between failures in various types of programs, thereby quantifying achievable exploit gaps in application-specific scenarios. While we have been able to estimate exploit gaps from prior experiments on N-version programming, precisely characterizing these values without operating real-world examples of NNVP is challenging. We would also like to formally verify and externally audit the code of our minimal Hydra Framework, which is responsible for coordinating heads and paying bounties. This could increase developers' assurance of correctness of our framework, thus making Hydra a more realistic option for deployment. Lastly, we hope to improve the efficiency in both development time and computational costs for Hydra contracts deployed in production.

References

[BDJS17] Lorenz Breidenbach, Phil Daian, Ari Juels, and Emin Gün Sirer. An in-depth look at the Parity multisig bug, Jul. 2017. `http://hackingdistributed.com/2017/07/22/deep-dive-parity-bug/`.

[BDTJ17] Lorenz Breidenbach, Philip Daian, Florian Tramèr, and Ari Juels. Enter the Hydra: Towards principled bug bounties and exploit-resistant smart contracts. Cryptology ePrint Archive, Report 2017/1090, 2017. `https://eprint.iacr.org/2017/1090`.

[CA95] Liming Chen and Algirdas Avižienis. N-version programming: A fault-tolerance approach to reliability of software operation. In *Fault-Tolerant Computing*, page 113. IEEE, 1995.

[ECK+91] Dave E Eckhardt, Alper K. Caglayan, John C. Knight, Larry D. Lee, David F. McAllister, Mladen A. Vouk, and John P. J. Kelly. An experimental evaluation of software redundancy as a strategy for improving reliability. *IEEE TSE*, 17(7):692–702, 1991.

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

[Eth18] Ethereum. Ethereum bug bounty, Jun. 2018. `https://bounty.ethereum.org/`.

[HSZ+17] Everett Hildenbrandt, Manasvi Saxena, Xiaoran Zhu, Nishant Rodrigues, Philip Daian, Dwight Guth, and Grigore Rosu. KEVM: A complete semantics of the Ethereum Virtual Machine, 2017.

[KL86] John C Knight and Nancy G Leveson. An experimental evaluation of the assumption of independence in multiversion programming. *IEEE Transactions on software engineering*, (1):96–109, 1986.

[McK98] William M McKeeman. Differential testing for software. *Digital Technical Journal*, 10(1):100–107, 1998.

[MJ16] Bill Marino and Ari Juels. Setting standards for altering and undoing smart contracts. In *RuleML*, pages 151–166. Springer, 2016.

[Swe17] Martin Holst Swende. Blockchain frontrunning, Jul. 2017. `http://www.swende.se/blog/Frontrunning.html`.

[Woo14] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger, 2014.

Author Biographies

- Lorenz Breidenbach is an SNF BRIDGE fellow in the department of Computer Science at ETH Zürich. His research interests include security, applied cryptography, and blockchains/cryptocurrencies. Breidenbach received a MSc in Computer Science from ETH Zürich. Contact him at `lorenz.breidenbach@inf.ethz.ch`.

- Philip Daian is a PhD student in Computer Science at Cornell Tech. His research interests include security and applied cryptography, with a focus on cryptocurrencies and smart-contracts. Daian received a BSc in Computer Science from the University of Illinois Urbana-Champaign. Contact him at `phil@cs.cornell.edu`.

- Florian Tramèr is a PhD student in Computer Science at Stanford University. His research interests include cryptography and security, in particular machine learning security and cryptocurrencies. Tramèr received a MSc in Computer Science from The École polytechnique fédérale de Lausanne (EPFL). Contact him at `tramer@cs.stanford.edu`.

- Ari Juels is a Professor of Computer Science at the Jacobs Technion-Cornell Institute at Cornell Tech. His research interests include blockchains, cryptocurrency, and smart contracts, as well as applied cryptography and cloud security. Juels received a PhD in computer science from U.C. Berkeley. Contact him at `juels@cornell.edu`.