Enter Hydra

towards (more) secure smart contracts

Philip Daian, Ari Juels
Cornell [Tech]

Florian Tramer
Stanford

Lorenz Breidenbach
ETH Zurich, Cornell [Tech]

Smart Contract Security - The Prongs

Formal Verification (+Specification)

what are we building and how can we check it?

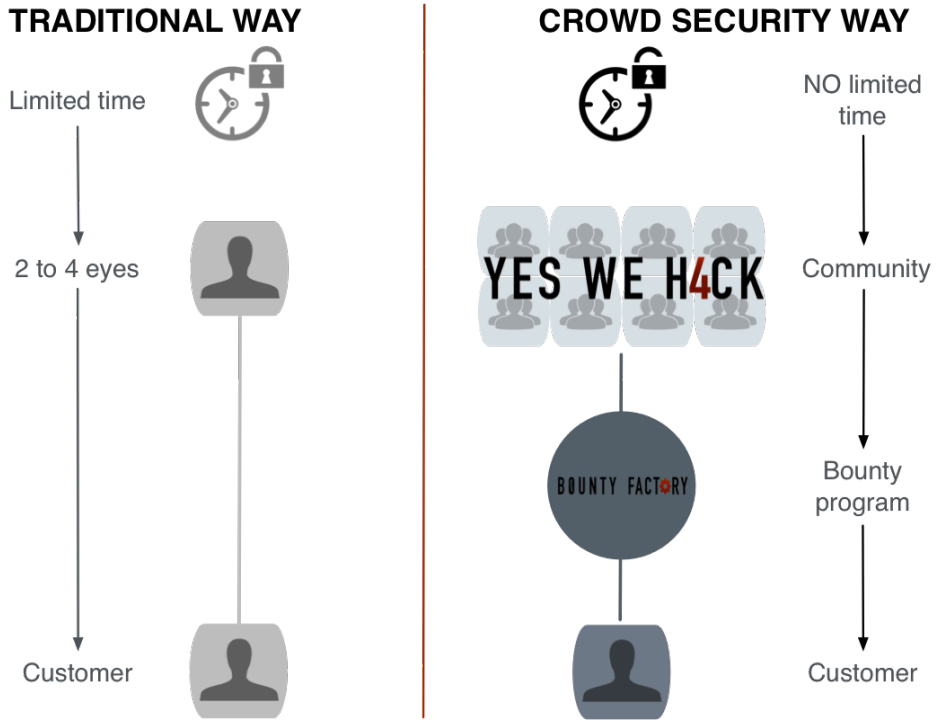
Escape Hatches

how can we react to the unforeseen?

Bug Bounties

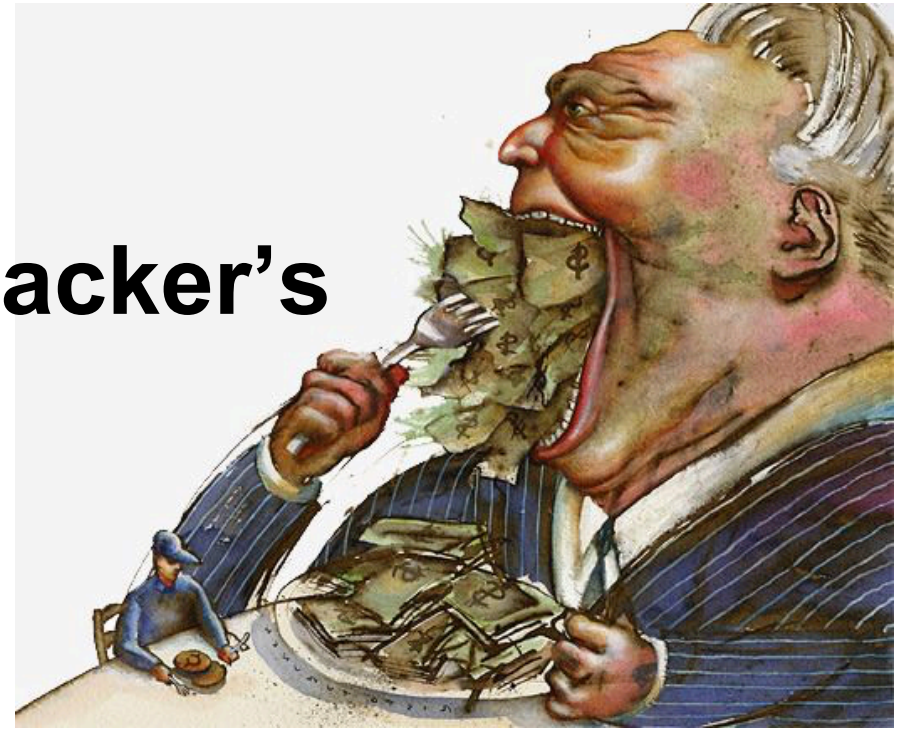
how can we address perverse incentives?

Why bug bounties?



Why bug bounties?

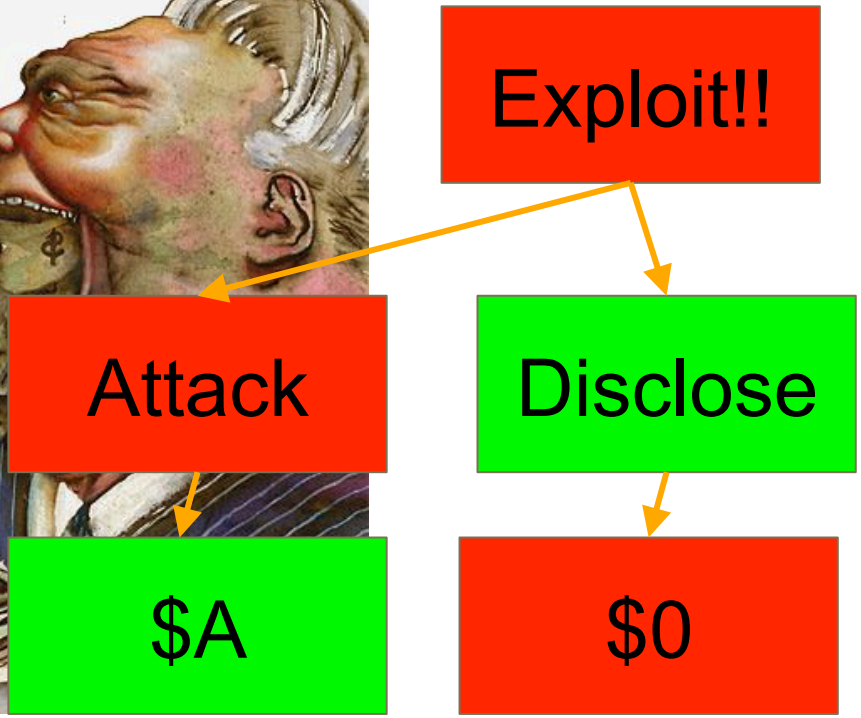
The rational attacker's game



Why bug bounties?

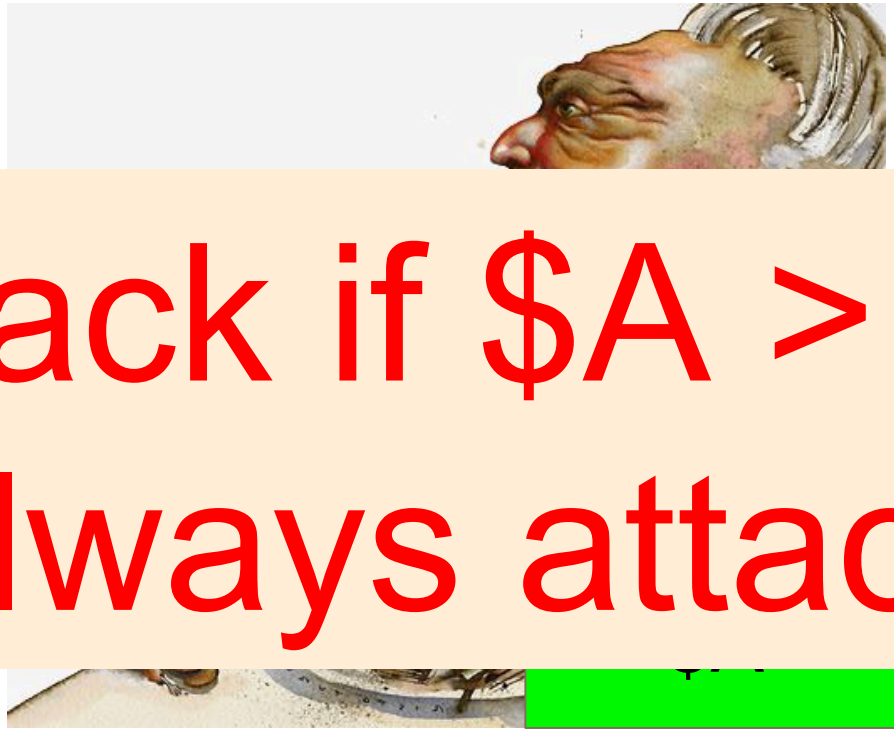
The rational attacker's game

No bounties



Why bug bounties?

The
ration
game
No bo



Exploit!!

close

0

Attack if $A > $0

Always attack

“Good enough” isn’t good enough

The rational attacker’s game

Classic bounty
Unknown payout

Exploit!!

Attack

Disclose

$A

$??

“Good enough” isn’t good enough

The
ratio
game

Attack if $A > $??

Classic bounty
Unknown payout

Exploit!!

se

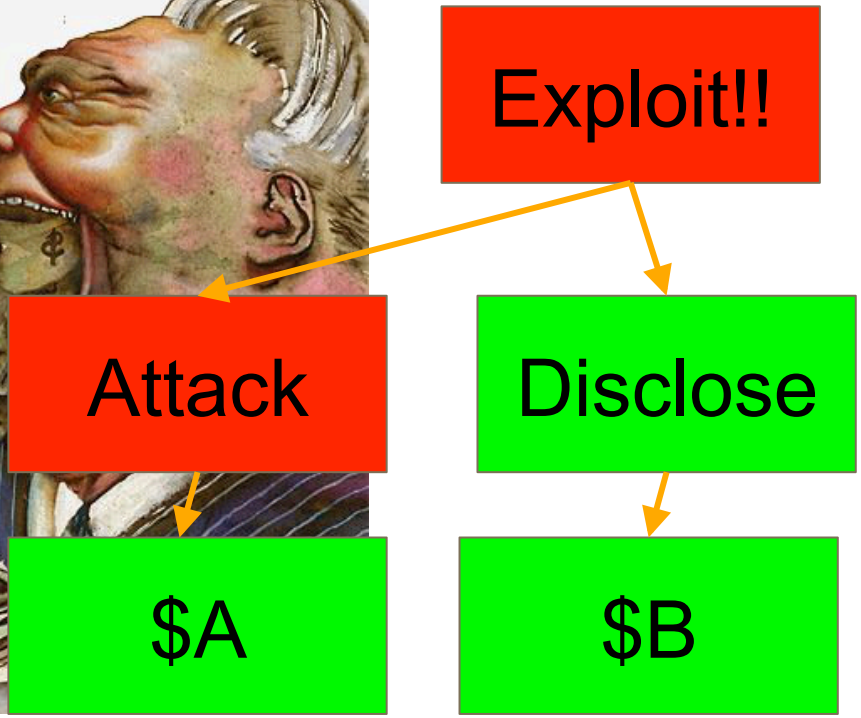
$A

$??

Towards a better game

The rational attacker's game

Classic bounty
Known payout



Towards a better game

The
ratio
game

Classic bounty
Known payout

Exploit!!

Attack if $A > $B

$A

$B

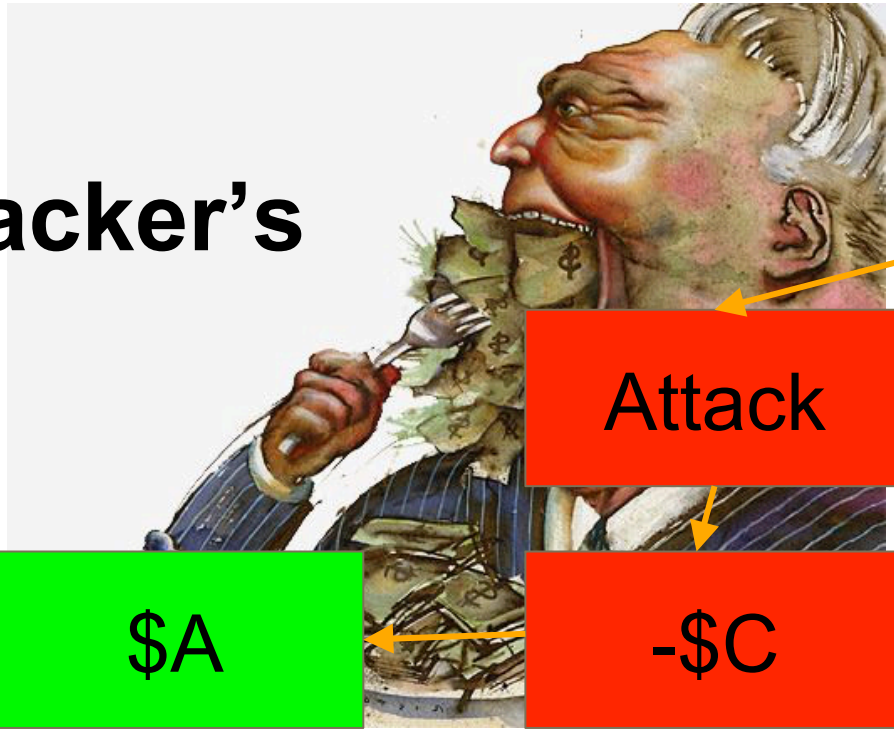
The ideal game

The rational attacker's game

Hydra bounty

Known payout

Gap to exploit



Exploit!!

Attack

Disclose

$A

-$C

$B

The ideal game

The
ra
ga
Hydra bounty
Known payout
Gap to exploit

Exploit!!

Attack if $A-$C > $B

$A

-$C

$B

The ideal game

The
ra
ga
Hy
Kn
Ga

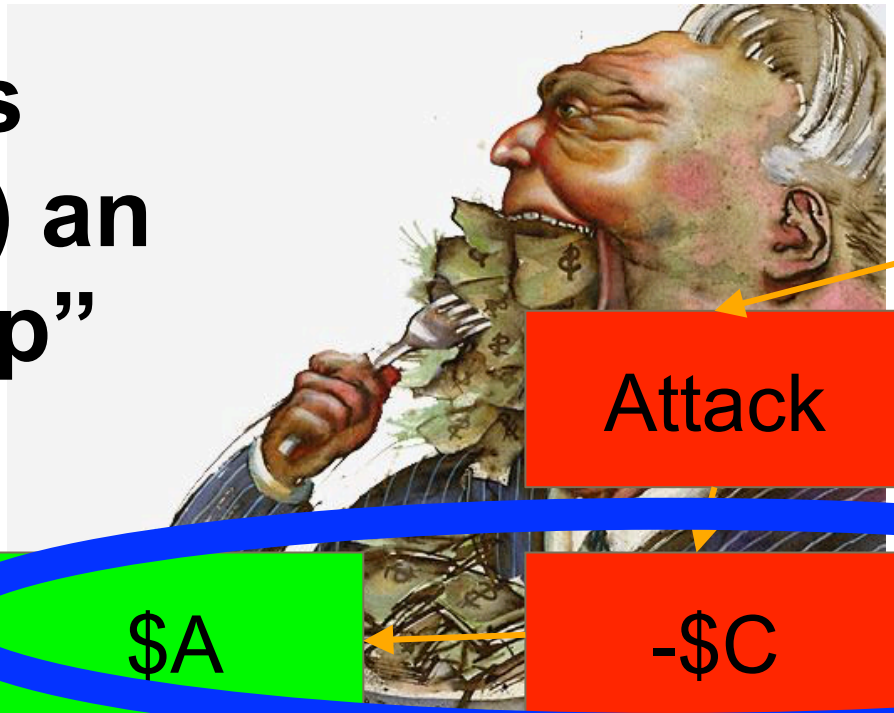
Exploit!!

Attack if $A-$C > $B

So, raise $C….

… mind the gap!

We call this barrier ($C) an "exploit gap"



Exploit!!

Attack

Disclose

$A

-$C

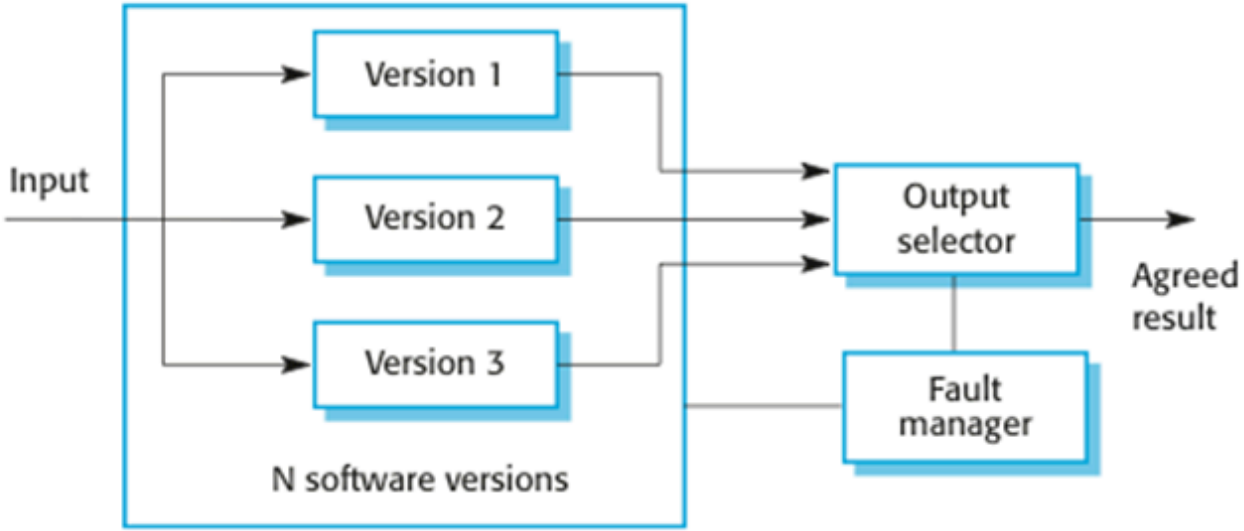
$B

Design Goals - The Perfect Bounty

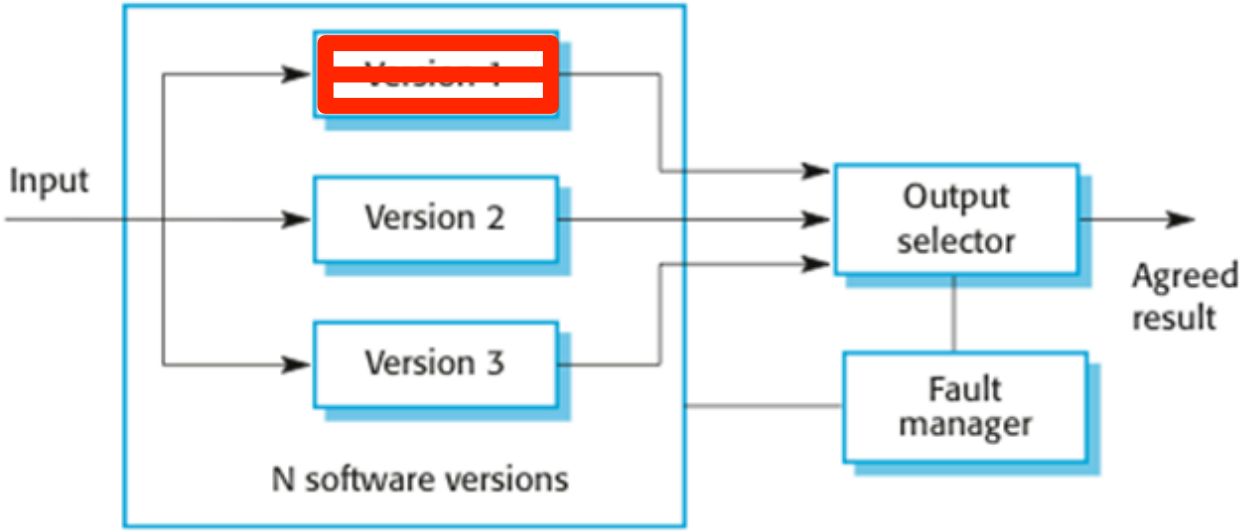
- Attack or disclose, not both (atomic)
- Predetermined payout (verifiable)
- Trustless payout (censorship resistant + verifiable)

Exploit Gap through Hydra Contracts

Chen & Avizienis, ‘78

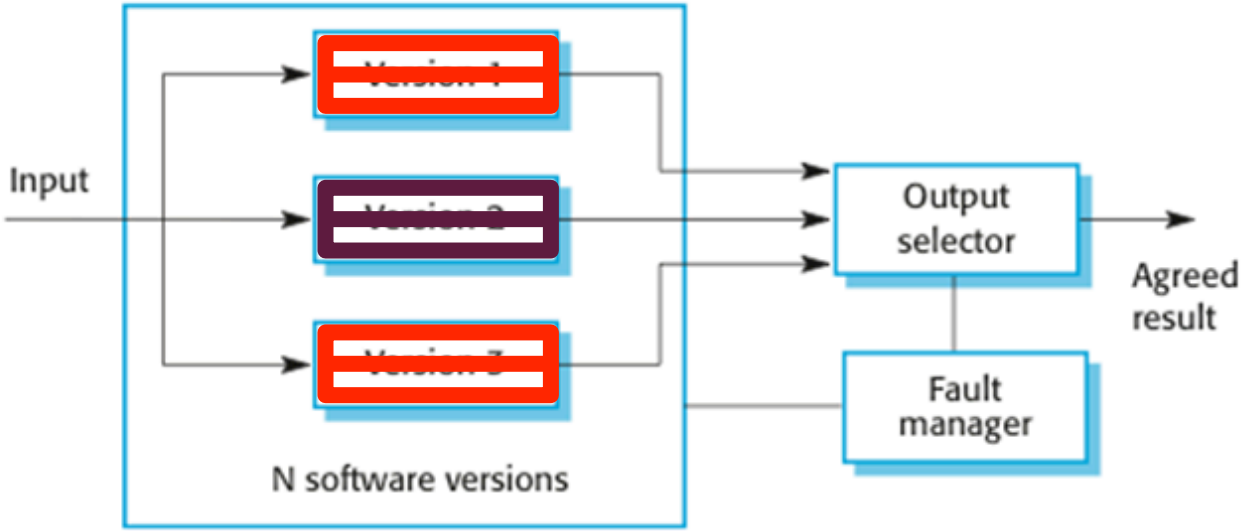


… Houston we have a gap (only one contract has bug)



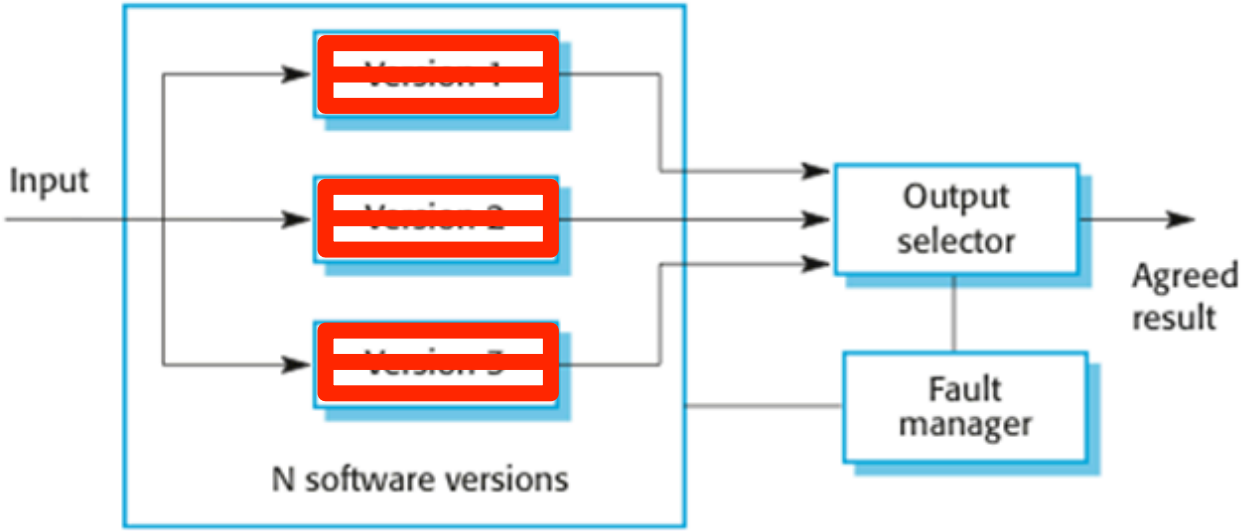
[assuming independence, composability of exploits, and many others]
[in the event of any disagreement, fault manager invoked]

… Houston we have a gap (contracts have different bugs)

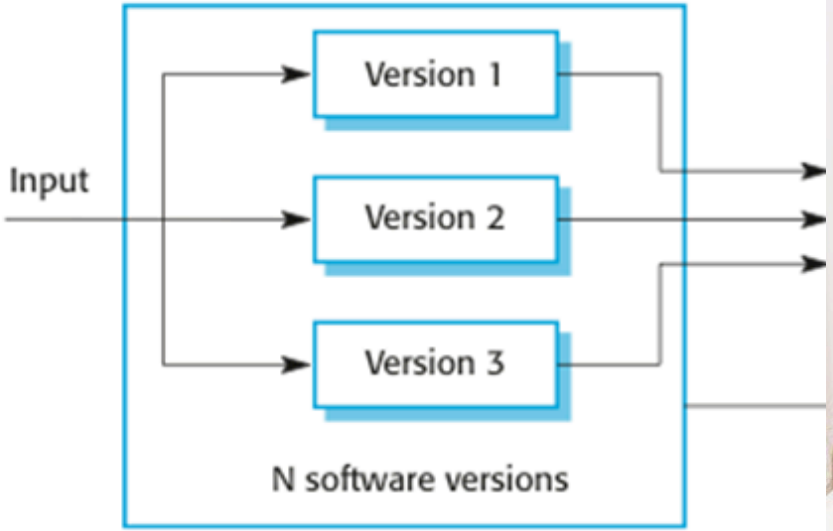


[assuming independence, composability of exploits, and many others]
[in the event of any disagreement, fault manager invoked]

… Houston we have no gap! Hydra fails!
(all contracts have same bug, empirically rare?)

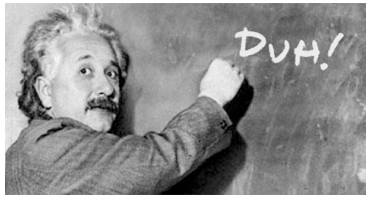


… let's bring back the 80's!



N-Version Programming Criticism

- Analysis assumes full independence of faults (correlations are annoying!)
- Knight-Leveson (‘86):
  « We reject the null hypothesis of full independence at a p-level of 5% »



- Eckhardt et al. (’91):
  « We tried it at NASA and it wasn’t *cost effective*»
  Worst-case: *3 versions = 4x fewer errors*

Cost, Availability & Reliability

- «Classical» N-Version Programming: **Availability >> Reliability**
  - **Majority Voting**: Always available, but may fail often

- Smart contracts: do we really car if it's down for a while?
  - N-out-of-N agreement: ***better no answer than the wrong one***
  - Empirically, there seem to be few *« harmless »* bugs

- *Numbers from Eckhardt et al. look much better:*
  - *For 3 versions,* **30 − 5087** times fewer failures
    (but some loss in availability…)

In practice as well as theory - preventable bugs

https://blog.ethereum.org/2016/06/19/thinking-smart-contract-security/

The DAO (obviously) **[language]**

The “payout index without the underscore” ponzi (“FirePonzi”) **[scam]**

The casino with a public RNG seed **[spec]**

Governmental (1100 ETH stuck because payout exceeds gas limit) **[programmer]**

5800 ETH swiped (by whitehats) from an ETH-backed ERC20 token **[language]**

The King of the Ether game **[language]**

Rubixi : Fees stolen because the constructor function had an incorrect name **[prg]**

Rock paper scissors trivially cheatable because the first to move shows their hand **[spec]**

Various instances of funds lost because a recipient contained a fallback function that consumed more than 2300 gas, causing sends to them to fail. **[spec/pltfrm]**

Various instances of call stack limit exceptions. **[programmer]**

In practice as well as theory - preventable bugs

https://blog.ethereum.org/2016/06/19/thinking-smart-contract-security/

The DAO (obviously) **[language]**

The “payout index without the underscore” ponzi (“FirePonzi”) **[scam]**

The casino with a public RNG seed **[spec]**

6-8/10 ain’t bad

(the rest are specification bugs or intentional backdoors)

consumed more than 2000 gas, causing sends to them to fail. **[spec/pitfall]**

Various instances of call stack limit exceptions. **[programmer]**

... so, the project

- Creation of trustless, decentralized bug bounty
- Increased security for mainnet contracts
  - Economic security through bounty program
  - Deployment with Hydra for exploit gap
- First rigorous, trustless incentive scheme for preventing smart contract attacks
- First decentralized incentives for defenders

Main Challenges for on-chain deployment

- Coordinating multiple smart contracts:
  - The coordinator should (hopefully) be bug free
  - Maintain consistent blockchain state
  - How to recover from a discovered bug => escape hatches

- Frontrunning (as always…)
  - Attacker can break the exploit gap by *witholding* bugs
  - Search for full exploit until someone tries to claim a bounty
  - Solution: Submarine sends!
    http://hackingdistributed.com/2017/08/28/submarine-sends/

Bug Withholding and Commit-Reveal

Sol 1: To claim bounty at time T, must *commit to bug* at time T- 1

Problem: Attacker commits in every round and only reveals if someone else does

Sol 2: To commit, you must pay $$ (in a verifiable way)

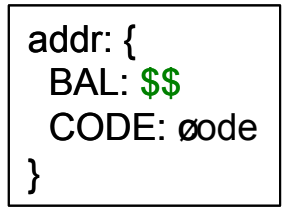
Problem: Attacker commits if someone else also commits

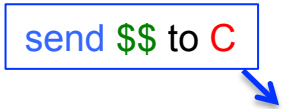
Sol 3: Hide commitments (e.g., proof of burn to random address)

Problem: Wasteful

Submarine Sends (post-metropolis version)

Goals: (1) only allow *committed* users to send a transaction to *C*
(2) being *eternally committed* is expensive
(3) attacker *can't know* if someone has committed
(4) money isn't wasted





Submarine sends:

Phase 1: compute *addr* = *H(C* || *nonce* || *code)* and send $$ to *addr*

Phase 2: reveal *addr* to *C*.

C verifies that addr got $$ in Phase 1

C creates a contract with the specified nonce and code

C collects $$ and allows transaction