Mètodes de Testeig Avançat i Recerca de Vulnerabilitats en Contractes Intel·ligents Basats en Fuzzing

Gianfranco Bazzani Valldepérez

Màster universitari Online de Ciberseguretat i Privadesa

Sistemes de Blockchain

Tutor/a de TF
Cristina Pérez Solà

Professor/a responsable de l'assignatura
Victor Garcia Font

11/06/2024

Universitat Oberta de Catalunya

FITXA DEL TREBALL FINAL

Títol del treball:	*Mètodes de Testeig Avançat i Recerca de Vulnerabilitats en Contractes Intel·ligents Basats en Fuzzing*
Nom de l'autor:	*Gianfranco Bazzani Valldepérez*
Nom del consultor/a:	*Cristina Pérez Solà*
Nom del PRA:	*Victor Garcia Font*
Data de lliurament (mm/aaaa):	*06/2024*
Titulació o programa:	*Màster universitari Online de Ciberseguretat i Privadesa*
Àrea del Treball Final:	*Sistemes de Blockchain*
Idioma del treball:	*Català*
Paraules clau	*Ciberseguretat, Contractes intel·ligents, Fuzzing*

Resum del Treball

Les tecnologies *blockchain* han permès la creació d'escassetat digital, la qual ha definit el valor digital. La seguretat és un factor crucial per preservar la integritat d'aquesta escassetat i assegurar el manteniment d'aquest valor.

Tecnologies *blockchain* programables, com Ethereum, permeten mitjançant contractes intel·ligents, la construcció de protocols financers autònoms que tenen la capacitat de posseir i transferir valor entre usuaris i altres protocols de manera *trustless* (sense necessitat de confiança). No obstant això, com qualsevol component de programari, els contractes intel·ligents no estan exempts de contenir vulnerabilitats, les quals en molts casos posen en risc el valor contingut en ells. D'aquesta manera, la seguretat esdevé un factor fonamental a tenir en compte en tot el cicle de vida dels contractes intel·ligents, especialment durant la fase de desenvolupament, a causa de la immutabilitat d'aquests una vegada són desplegats a la *blockchain*.

Aquest treball se centra en explorar l'ecosistema d'eines de testeig avançat basades en *fuzzing* per a contractes intel·ligents Ethereum, i com aquestes poden ser utilitzades tant per a la recerca de vulnerabilitats com per assegurar la correctesa de les propietats dels contractes intel·ligents. A més, s'explorarà la integració i l'automatització d'aquestes eines dins del procés de desenvolupament per millorar la seguretat durant tot el cicle de desenvolupament dels contractes intel·ligents. Amb la realització d'aquest treball, es pretén documentar els coneixements teòric-pràctics necessaris per a la construcció i automatització de *setups* de testeig avançat de contractes intel·ligents basats en *fuzzing* de manera eficient i professional.

Abstract

Blockchain technologies allow the creation of digital scarcity, which has defined digital value. Security is a crucial factor in order to keep the integrity of that scarcity and ensure the maintenance of that value.

Programmable blockchain technologies, such as Ethereum, allow through smart contracts the construction of autonomous financial protocols that have the capacity to own and transfer value between users and other protocols in a trustless manner.However, like any software component, smart contracts are not free from containing vulnerabilities, which in many cases put at risk the value contained within them. In this way, security becomes a fundamental factor to consider throughout the life cycle of smart contracts, especially during the development phase, due to the immutability of these once they are deployed on the blockchain.

This work focuses on exploring the ecosystem of advanced fuzzing based testing tools for Ethereum smart contracts, and how these can be used both for vulnerability discovery and ensuring the correctness of the properties of smart contracts. Additionally, the integration and automation of these tools within the development process will be explored to enhance security throughout the entire development cycle of smart contracts. With the completion of this work, it is intended to document the theoretical-practical knowledge necessary for the construction and automation of advanced testing setups for smart contracts based on fuzzing in an efficient and professional manner.

Índex

Llista de figures

Llista de figures

1 Introducció

1.1. Context i justificació del Treball

Blockchains com Bitcoin han obert la porta als seus usuaris per emmagatzemar i transferir valor digitalment d'una manera *trustless*, segura, transparent i anònima [1]. Tot i això, la seva limitada capacitat de programabilitat i una cultura de desenvolupament més conservadora restringeixen Bitcoin en certs casos d'ús. Una altra aplicació de les tecnologies *blockchain* es troba en plataformes com Ethereum, dissenyada per funcionar com una màquina programable de propòsit general, capaç d'executar codi de qualsevol complexitat amb funcionalitats integrades per a la gestió de valor.

Ethereum abstreu els mecanismes tècnics específics de les tecnologies *blockchain* i proporciona un entorn d'execució determinista i segur per al desenvolupament d'aplicacions financeres descentralitzades. La EVM (*Ethereum Virtual Machine*) és una màquina d'estats de propòsit general basada en *stack* que permet executar programes arbitraris escrits com a sèrie d'instruccions anomenades bytecodes. L'execució de la EVM i l'evolució del seu estat representen una sola instància global executant-se de manera replicada en tots els nodes de la xarxa, aquest comportament ha fet que Ethereum guanyi el títol de *"The World Computer"* [2].

Els programes que s'executen en aquest entorn s'anomenen contractes intel·ligents, i aquests poden ser escrits amb llenguatges d'alt nivell com Solidity o Vyper, que després es compilen a bytecodes del conjunt d'instruccions EVM. Aquests permeten l'automatització en la gestió de valor i la construcció de mecanismes financers autònoms, els quals poden emmagatzemar i transferir valor digital a altres contractes intel·ligents o als usuaris. Aquests tipus de mecanismes s'agrupen sota el nom de DeFi (de l'anglès, *Decentralized Finance*) [3]. Gràcies a aquestes propietats, la construcció de protocols *blockchain* és relativament fàcil i no és necessari tenir coneixements de baix nivell sobre les complexes tecnologies que s'utilitzen sota aquesta capa d'abstracció. No obstant, aquest fet pot ser una arma de doble tall ja que és fàcil escriure codi, però és difícil escriure codi de qualitat i segur.

Degut a l'alt valor que els protocols DeFi gestionen, la seguretat s'ha convertit en un element fonamental al llarg de tot el cicle de vida d'aquest tipus de productes. A més, la limitació per actualitzar la lògica dels protocols una vegada estan en producció, a causa de la immutabilitat del codi dels contractes intel·ligents, fa que sigui extremadament important minimitzar els riscos en fases prèvies al desplegament.

En un protocol DeFi, la seguretat mai és suficient, i la presència d'una vulnerabilitat i l'explotació d'aquesta per part d'entitats malicioses pot suposar directament la pèrdua de valor dels usuaris i una gran pèrdua reputacional per al protocol. L'explotació de contractes és la causa més comuna de pèrdua de fons en protocols DeFi [4]. El terme *Rekt* s'ha encunyat en l'ecosistema per descriure una pèrdua financera severa. L'explotació d'una vulnerabilitat pot suposar el pas d'un protocol en producció, aparentment sa, a estar completament *Rekt* després d'una pèrdua crua i brutal de valor.

A continuació, amb la finalitat de quantificar l'impacte de la presència de vulnerabilitats en protocols DeFi i justificar la importància del camp d'estudi proposat, s'ha realitzat una anàlisi del valor robat a protocols DeFi en els darrers tres anys i comparat amb el TVL (de l'anglès, *Total Value Locked*) emmagatzemat en contractes en diferents plataformes DeFi.

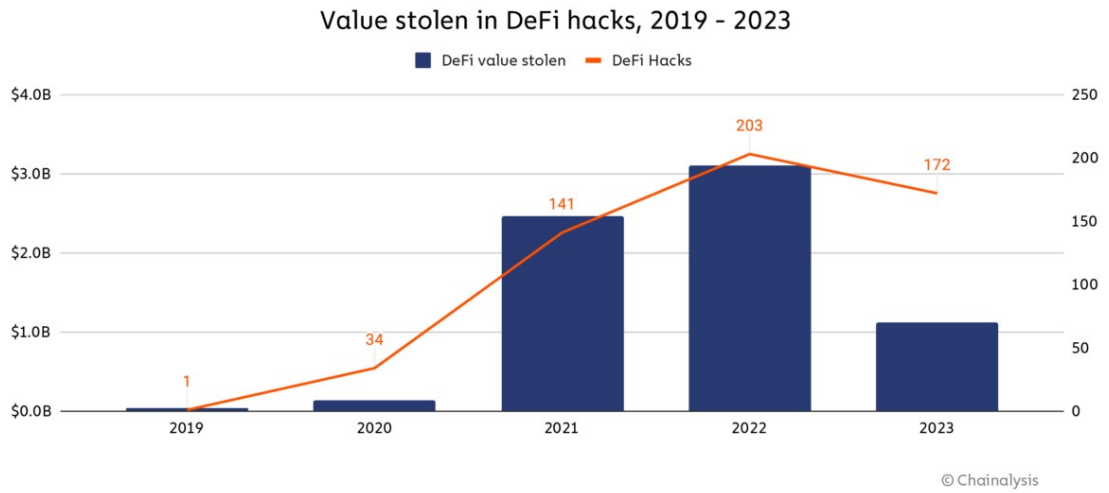

Figura 1: Valor total robat en hacks DeFi, 2019 - 2023
Font: Chainalysis [5]

La Figura 1 mostra el valor total robat als protocols DeFi en el període entre els anys 2019 i 2023. La Figura 2 mostra el TVL mensual en plataformes DeFi en el període entre gener de 2021 a desembre de 2023. Si calculem una mitjana anual per als anys 2021, 2022 i 2023, s'obté un TVL mitjà anual de $100B, $90B i $45B, respectivament, dels quals un 2,5%, 3,3% i 2,2% són robats. Com es pot notar, els percentatges són considerables i es parla de pèrdues de l'ordre de milers de milions de dòlars.

Una adopció massiva d'una economia oberta basada en tecnologies *blockchain* requereix que els projectes es centrin principalment en implementar i mantenir sistemes i mesures de seguretat per a fortificar els protocols durant tot el seu cicle de vida i sacrificar, si cal, estratègies de creixement que puguin posar en risc el valor dels usuaris.

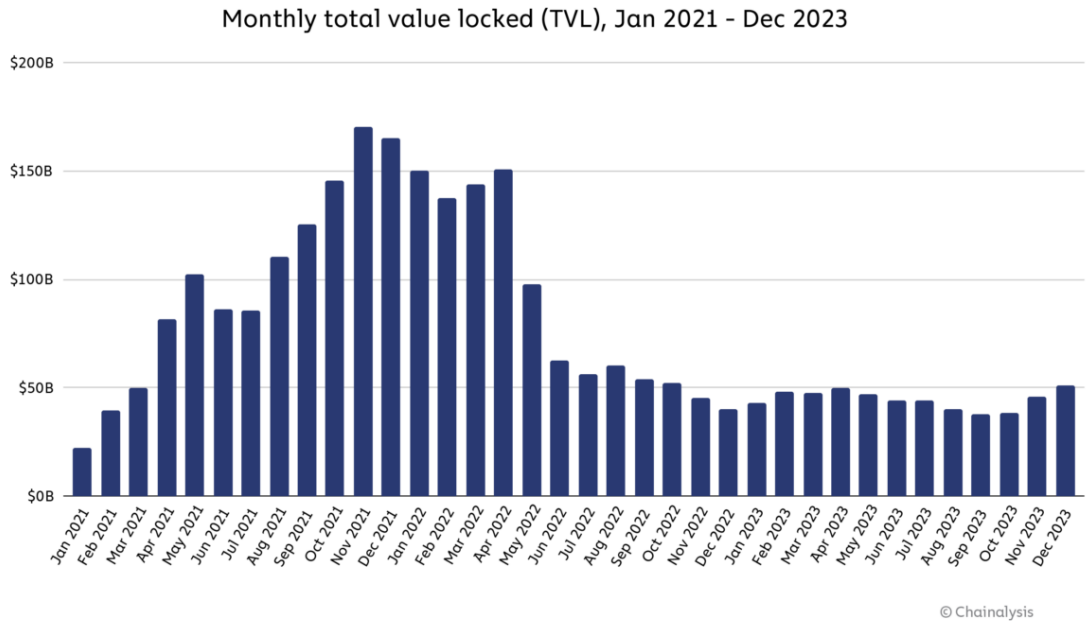

Figura 2: TVL mensual en DeFi, Gener 2021 - Desembre 2023
Font: Chainalysis [5]

En una revisió de seguretat manual dels contractes (*Security Review* en anglès), la garantia en el nivell de seguretat depèn de l'experiència i habilitats de l'equip revisor i del temps invertit en l'anàlisi. A més, aquesta només és vàlida per a una versió específica del codi, normalment indicada per un *commit* concret del repositori del projecte. Tot aquest procés s'invalida i es necessita d'una altre anàlisi manual en el moment que el codi es modifica, encara que sigui mínimament. Mitjançant la construcció ad hoc d'un *setup* de testing avançat basat en eines de *fuzzing*, és possible detectar *edge cases* que poden escapar-se en revisions manuals. A més, el *setup* és reusable i permet assegurar que les propietats del sistema es mantinguin en futures modificacions, així com l'automatització del testeig mitjançant la integració contínua (CI). Amb això no es pretén substituir de cap manera les revisions manuals sinó combinar tots els possibles mètodes per millorar la seguretat dels contractes intel·ligents durant el seu procés de desenvolupament.

El *fuzzing* és una tècnica de testeig de programari que implica proporcionar dades d'entrada aleatòries, inesperades o invàlides a un programa per descobrir errors i vulnerabilitats. Aquest treball pretén explorar l'ecosistema d'eines de testeig avançat basades en *fuzzing* per a la detecció prematura de vulnerabilitats en contractes intel·ligents d'Ethereum i com aquestes es poden integrar en el cicle de desenvolupament dels contractes intel·ligents per protegir contra la inclusió de vulnerabilitats en els sistemes en producció.

La gran comunitat de desenvolupament *opensource* que s'ha construït al voltant de la plataforma Ethereum i la seva cultura de desenvolupament ràpidament innovadora i centrada en el futur ha posicionat la plataforma a l'avantguarda de la tecnologia. Això ha fet que molts projectes en la indústria adoptin el conjunt d'instruccions EVM com un estàndard per, d'alguna manera, heretar aquesta comunitat fàcilment. Per aquest motiu, aquest treball se centra en l'ecosistema d'Ethereum i la EVM, els quals, en definitiva, conformen un àmbit extens que abasta diversos llenguatges de programació d'alt nivell i un ecosistema ric en eines i aplicacions.

Com a motivació final, s'ha triat aquest tema d'estudi per al TFM ja que actualment em trobo treballant per a una companyia que ofereix serveis en l'àmbit de la ciberseguretat *blockchain*. Aquesta elecció representa una oportunitat significativa de creixement professional dins de la meva trajectòria en aquest sector.

1.2. Objectius del Treball

Amb la realització d'aquest treball es pretén assolir els següents objectius:

- Estudiar els mètodes de *fuzzing* de contractes intel·ligents i contrastar les diferències entre els mètodes de *fuzzing* convencionals.
- Estudiar l'arquitectura d'eines de *fuzzing* de contractes intel·ligents àmpliament utilitzades al sector.
- Realitzar una comparativa entre les diferents eines, identificant els seus avantatges i desavantatges específiques, així com la seva eficàcia en la detecció de vulnerabilitats.
- Demostrar amb un cas pràctic l'ús d'aquestes eines en la construcció de *setups* de tests per detectar vulnerabilitats en contractes intel·ligents.
- Demostrar amb un cas pràctic la integració en CI d'aquestes eines en el cicle de vida del producte.

Amb la realització d'aquest treball no es pretén:

- Estudiar eines de *fuzzing* de contractes intel·ligents en ecosistemes diferents a Ethereum, l'abast d'aquest estudi es limitarà a l'ecosistema Ethereum i als ecosistemes compatibles anomenats *EVM compatibles*.
- Estudiar altres mètodes de detecció de vulnerabilitats en contractes intel·ligents, l'abast d'aquest estudi es limitarà a l'estudi d'eines basades en *fuzzing*, encara que l'ecosistema d'eines i mètodes és molt més extens, incloent anàlisi estàtic, verificació formal i verificació manual.

- Promoure l'ús dels mètodes estudiats per a la recerca de vulnerabilitats en protocols en producció per explotar-les de manera maliciosa. El crim digital és il·legal i comporta conseqüències greus que poden variar des de multes substancials, danys a la reputació, fins a penes de presó, depenent de la gravetat.

1.3. Impacte en sostenibilitat, ètic-social i de diversitat

La UOC (Universitat Oberta de Catalunya) està públicament compromesa amb els ODS (Objectius de Desenvolupament Sostenible per al 2030, de l'ONU) i defineix una sèrie de dimensions amb les quals aquest treball està també alineat: la competència de compromís ètic i global (CCEG). En aquesta secció, s'analitza com aquest treball s'alinea amb les diverses dimensions del CCEG.

Sostenibilitat

L'ecosistema de desenvolupament d'Ethereum ha demostrat un fort compromís amb la responsabilitat de desenvolupar infraestructures sostenibles i un consum energètic responsable. Esdeveniments com el "*Merge*" (15 setembre 2022) són un clar exemple d'aquest compromís. Amb aquest, es van introduir canvis tecnològics al protocol que eliminaven la necessitat d'assegurar la xarxa mitjançant la mineria, la qual té un consum intensiu d'energia.

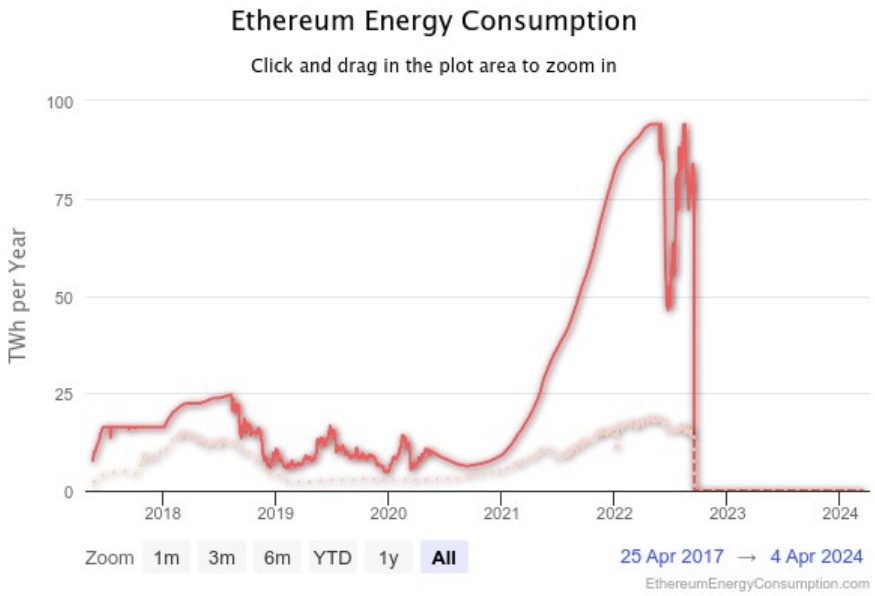

Figura 3 Consum energètic de la xarxa Ethereum, Abril 2017 - Abril 2024
Font: Digiconomist

L'organització EUBOF estima que el consum energètic de la xarxa s'ha reduït en un 99,98% [6]. Les estimacions de consum eren de 93 TWh per any, amb un mínim de 19 TWh al maig de 2022 (*pre-Merge*). Després del *Merge*, els consums estimats han baixat a 0.0096 TWh per any, amb un mínim de 0.00249 TWh a l'octubre de 2022 (*post-Merge)*.

Comportament ètic i responsabilitat social

Les propietats de les tecnologies *blockchain* faciliten la creació de sistemes econòmics digitals sostenibles, segurs, no permissionats, amb una alta disponibilitat i un accés global sense restriccions de fronteres nacionals. Aquestes característiques milloren l'accés als serveis financers i fomenten un creixement econòmic sostingut e inclusiu, facilitant a les persones de baix nivell adquisitiu o aquelles en regions marginades l'accés a serveis financers. El camp d'estudi d'aquest treball se centra específicament en mètodes per a la fortificació d'aquests sistemes financers per millorar-ne la seguretat i la confiança.

Diversitat i drets humans

Un sistema econòmic obert i transparent basat en tecnologies *blockchain* facilita la traçabilitat dels fluxos de valor destinats a activitats que violen els drets humans, com la guerra. Això desincentiva governs i empreses a invertir fons en aquestes activitats, ja que pot suposar una violació de principis ètics i legals de manera pública, i tenir grans conseqüències reputacionals.

A més a més, característiques com l'accés lliure i l'anonimat permeten l'ús de productes financers per a totes les persones de manera igualitària, dificultant la discriminació per sexe, raça, orientació sexual, religió, entre altres, i oposant-se a la desigualtat.

1.4. Enfocament i mètode seguit

Per al desenvolupament del projecte, es seguirà una metodologia mixta que combina l'enfocament teòric i pràctic.

La fase inicial se centrarà en l'estudi i documentació de les tècniques de Fuzzing convencionals, així com en la comparació i anàlisi de les diferències amb les tècniques específiques per a contractes intel·ligents. Aquest procés inclou una revisió bibliogràfica de la literatura relacionada, així com una anàlisi comparativa de les diferents tècniques.

Un cop completada la fase d'estudi, passarem a la fase de planificació i disseny, on es definiran els requisits del sistema de Fuzzing per al projecte Diva Staking.

En la següent fase, d'implementació i desenvolupament, es realitzarà la construcció del setup de Fuzzing basant-se en el disseny proposat i en el coneixement adquirit durant la fase d'estudi.

Durant totes les fases, es duran a terme reunions periòdiques amb el supervisor per validar els avenços i ajustar l'enfocament si cal.

1.5. Planificació del Treball

El flux de feina del projecte estarà sincronitzat amb els lliuraments PAC del semestre acadèmic:

- PAC1 12 Març:
  - Confecció de la proposta inicial del tema del treball final.
  - Definició dels objectius i planificació del treball.
  - Confecció inicial de la memòria acord amb els requeriments de la PAC1.
- PAC2 09 Abril
  - Introducció al *fuzzing*.
  - Estudi f*uzzing* de contractes intel·ligents i ecosistema d'eines disponibles.
  - Ampliació memòria PAC2.
- PAC3 07 Maig
  - Comparació d'eines.
  - Aplicació practica: construcció de setup de *fuzzing* per a projecte Diva Staking.
  - Ampliació memòria PAC3.
- PAC4 11 Juny
  - Aplicació practica: depuració i finalització dels components del seutp definits en l'abast.
  - Anàlisis de resultats.
  - Finalització memòria.

La Figura 4 mostra el diagrama de Gantt confeccionat per a traçar la realització de les tasques del projecte.

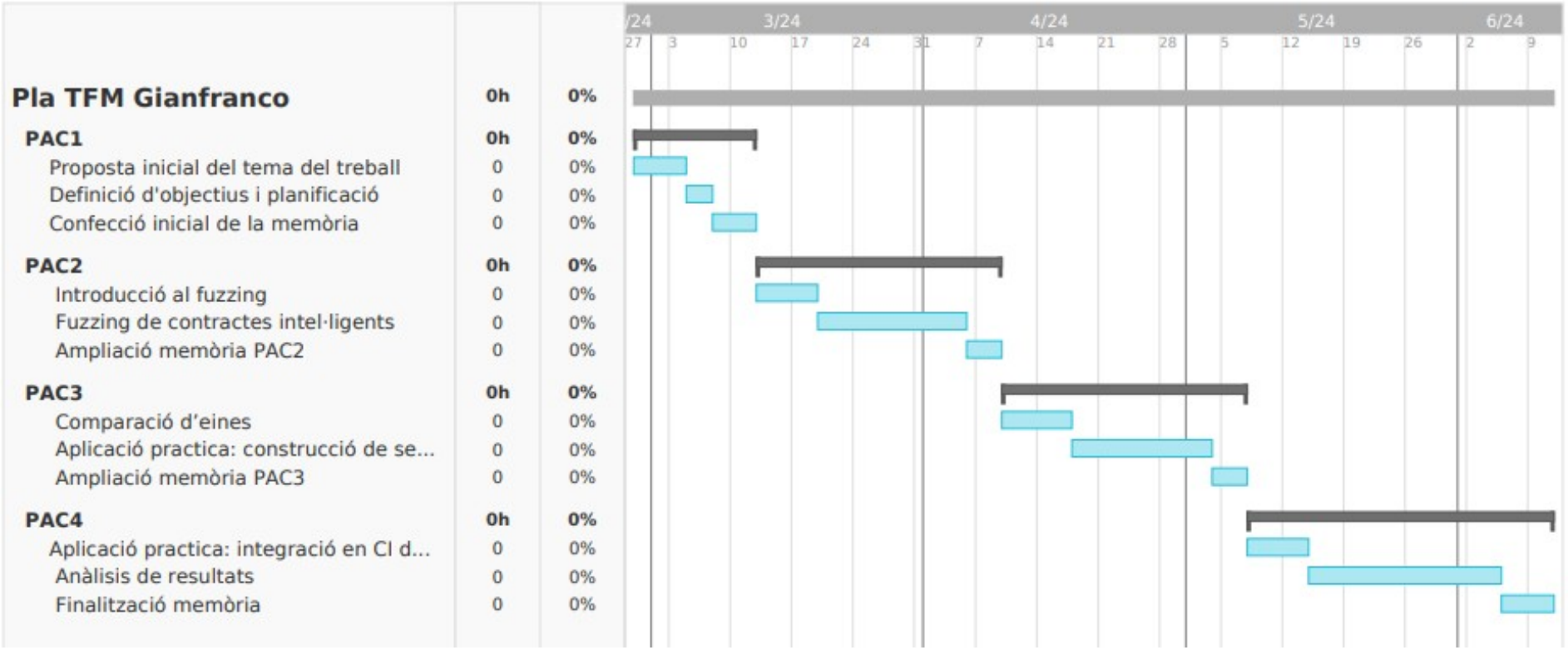

Figura 4 Diagrama de Gantt de la planificació del projecte.

1.6. Breu sumari de productes obtinguts

Com a resultat de la part teòrica del treball, s'ha elaborat un document tècnic que sintetitza l'estat de l'art en els mètodes de *fuzzing* per a contractes intel·ligents.

Com a resultat de la part pràctica del treball, s'ha desenvolupat un *setup* de *fuzzing* de qualitat i funcional, capaç de detectar vulnerabilitats en tres dels contractes principals del projecte Diva Staking.

1.7. Breu descripció dels altres capítols de la memòria

A continuació es descriuen lleugerament els continguts de cada apartat d'aquesta memòria:

1. **Introducció:** Conté una descripció del context i justificació del treball, així com els objectius plantejats inicialment i la planificació per a assolir-los.
2. **Introducció al Fuzzing:** Presenta els conceptes i mètodes generals del fuzzing, així com els seus orígens. L'objectiu d'aquest apartat és descriure el fuzzing per tenir una visió general i poder comparar i entendre les diferències i especificitats del fuzzing de contractes intel·ligents.
3. **Fuzzing de contractes intel·ligents:** Descripció de l'estat de l'art de les eines i mètodes de fuzzing de contractes intel·ligents. Aquest apartat és el resultat de l'estudi de les pràctiques observades en l'ecosistema de seguretat d'Ethereum.
4. **Resultats:** En aquest apartat es descriuen els resultats obtinguts d'aplicar els mètodes descrits en l'apartat anterior en un projecte basat en contractes intel·ligents real.
5. **Conclusions i treballs futurs:** Reflexió sobre els resultats del treball i proposta de possibles àrees a seguir explorant.
6. **Glossari:** Definició de termes específics utilitzats al llarg del treball.
7. **Bibliografia:** Índex de recursos bibliogràfics citats al llarg del treball.

2 Introducció al *Fuzzing*

2.1. Descripció general.

El *Fuzzing* és una tècnica de testeig dinàmic de programari on es prova l'execució d'un component específic de programari amb entrades generades de manera aleatòria. El programa és monitoritzat durant l'execució per detectar excepcions, fallades o altres comportaments inesperats. Aquesta tècnica permet automatitzar la identificació d'errors, bretxes de seguretat o possibles fallades del sistema provant un gran nombre de entrades del domini de les possibles entrades del programa que amb altres tipus de mètodes seria inviable provar. A més, els *fuzzers* també poden provar entrades no esperades o malformades.

2.2. Origen

El terme fuzzing va ser introduït pel professor de la Universitat de Wisconsin, Barton P. Miller l'any 1988. Aquest es va adonar que, durant una nit de tempesta mentre es connectava remotament a través de la línia telefònica al seu ordinador de la universitat, la tempesta causava soroll en la línia, produint comandes UNIX aleatòries i no esperades pel programes fent-los fallar contínuament. Bart va definir aquest tipus d'entrades aleatòries i no estructurades amb el terme *fuzz*, i va proposar un exercici de programació als seus alumnes, el qual esdevindria en la creació i ús dels primers *fuzzers*.

Amb les paraules del professor Miller: "Encara que la nostra estratègia sembli ingènua, la seva habilitat per descobrir errors fatals en programes és impressionant. Si considerem un programa com una màquina d'estats complexa, la nostra estratègia de testeig es pot veure com un passeig aleatori per tots els possibles estats a la recerca d'estats indefinits" [7].

2.3. Arquitectura

Generalment, un *setup* de *fuzzing* està format per tres components principals descrits a continuació:

- **El sistema instrumentat sotmès a prova o *SUT* (*System Under Test*):** Compost principalment pel component que es vol provar i els mecanismes de monitoratge que permeten detectar irregularitats.
- **El generador d'entrades o *Fuzzer Runtime*:** És responsable de generar les entrades que es lliuraran al sistema. Aquestes entrades es poden generar utilitzant diferents algoritmes i s'enviaran al h*arness*.
- **Mecanisme de lliurament o *Harness*:** És responsable de prendre les entrades aleatòries del generador d'entrades i executar el *SUT* amb aquestes com a entrades.

La figura 5 mostra l'estructura bàsica d'un *setup* de *fuzzing*.

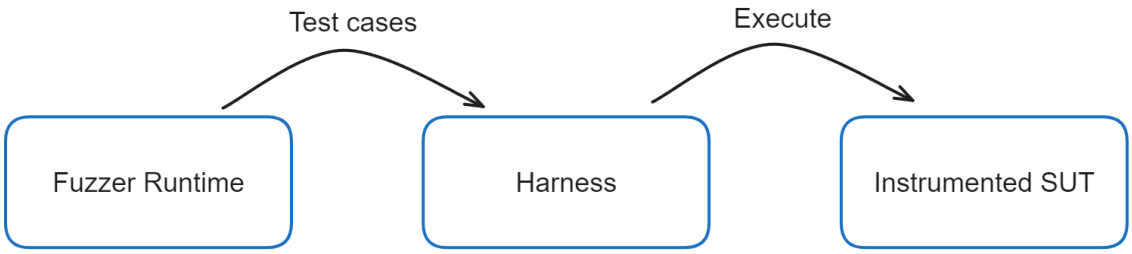

Figura 5: Arquitectura general d'un *setup* de *fuzzing*.

2.4. Tècniques de *fuzzing*

Depenent del nivell de coneixement que es tingui sobre el sistema, la tècnica d'anàlisi serà diferent i comportarà variacions en el procés de fuzzing així com en els resultats obtinguts. De manera general, podem categoritzar les tècniques en dos grans grups: caixa negra (*Black Box*) i caixa blanca (*White Box*).

Black-Box o caixa negra

L'objectiu d'aquesta tècnica és derivar els tests a partir de l'especificació del sistema. No es té accés al codi font ni a cap coneixement sobre el funcionament intern, es disposa únicament del binari del programa.

L'avantatge del testeig *black-box* és que permet derivar tests a partir de l'especificació, els quals seran independents de qualsevol implementació concreta. No obstant això, l'anàlisi està limitat a provar entrades i analitzar les sortides o possibles errors, i al no disposar del codi, no és possible instrumentar-lo.

El procés de *fuzzing* és més lent i menys eficient, i podria no cobrir tots els detalls d'una implementació concreta. Si l'objectiu és realitzar una anàlisi de seguretat del sistema, aquest tipus de *fuzzing* no és l'opció més adequada.

White-Box o caixa blanca

A diferència del testeig de caixa negra, en el testeig de caixa blanca els tests es deriven analitzant la implementació concreta, de manera que es disposa del codi d'aquesta.

Permet introduir el concepte de cobertura de codi (***code coverage***), que es refereix a la quantitat de codi del sistema objectiu que s'ha executat durant la campanya de *fuzzing*. Tenint accés al codi, podem instrumentar-lo i analitzar-lo durant l'execució per rastrejar quins segments del codi s'han executat. Després, aquesta informació es pot passar al programador, qui llavors pot centrar-se en escriure tests que cobreixin el codi encara no cobert.

També permet la introducció del concepte de *fuzzer* guiat per retroalimentació (***feedback-guided***), que consisteix en l'aplicació de tècniques que modifiquen els casos de tests durant l'execució de la campanya de *fuzzing* per a maximitzar-ne la cobertura

Els principals avantatges són que permeten realitzar campanyes de *fuzzing* més localitzades en diferents funcions del sistema en lloc del sistema complet. A més, es pot actuar fins i tot en els casos on l'especificació no aporta prou detalls, la qual cosa ajuda a identificar els *edge cases*.

2.5. Oracles

Tot i que una alta cobertura del codi ens indica que la campanya de *fuzzing* abasta una gran part del codi sotmès a prova, això no implica que aquest sigui lliure d'errors. El programa podria retornar valors incorrectes sense arribar a fallar, i el *fuzzer* podria no ser capaç de detectar aquests errors de funcionament. Els oracles són un element del *setup* del *fuzzing*, encarregats de verificar els resultats i assegurar que el *SUT* es comporti com s'espera en tot moment.

2.6. Guiat del *fuzzer* mitjançant mutació

La majoria de programes esperen que les dades en les seves entrades segueixin un format específic altament estructurat per a poder processar-les correctament. Mitjançant la generació d'entrades purament aleatòries, és molt poc probable que es produeixin entrades vàlides per al programa testejat i alhora forçar els diferents camins d'execució vàlids.

L'alternativa a crear entrades aleatòries és iniciar el procés de *fuzzing* amb una sèrie d'entrades vàlides conegudes (***seed***) i mutar-les al llarg de la campanya de *fuzzing*. Per a maximitzar la cobertura, el programa es testeja amb mutacions de la *seed* mentre es monitoritza la cobertura. Les entrades mutades interessants que provoquen nous camins d'execució diferents en el programa testejat s'emmagatzemen en el que es coneix com a **corpus**. El corpus es pot utilitzar com a *seed* per a iniciar futures campanyes més exhaustives partint d'una alta cobertura.

Al llarg de la campanya, el *fuzzer* quantificarà la capacitat dels elements del corpus per produir camins d'execució nous o menys probables, i aquestes entrades seran prioritzades durant la campanya. El paper «Coverage-based Greybox Fuzzing as Markov Chain» [8] descriu una metodologia mitjançant la qual s'assigna una **energia** específica a cada element del corpus, la qual especifica el nombre de mutacions que es generen a partir de cada element. Això permetrà generar mutacions del corpus que recorrin camins d'execució diferents en el programa amb major probabilitat i obtenir una major cobertura de manera més eficient.

2.7. *Fuzzing* sintàctic

El fuzzing sintàctic es centra en la creació de dades que segueixen un cert grau de la sintaxi o estructura esperada de les entrades vàlides del programa testejat. Per a realitzar fuzzing sintàctic, necessites entendre quina és la sintaxi d'entrada esperada per programa testejat. L'objectiu és generar dades d'entrada que siguin formalment correctes per a maximitzar l'exploració de camins d'execució vàlids en el programa.

3 Fuzzing de contractes intel·ligents

3.1. Detecció d'errors i vulnerabilitats en contractes intel·ligents.

Encara que l'abast d'aquest treball es centra en mètodes basats en *fuzzing*, en aquest apartat es pretén donar una visió general dels mètodes disponibles per a garantir la seguretat en contractes intel·ligents. La implementació de mesures de detecció d'errors i vulnerabilitats és essencial i s'ha de considerar des de les primeres etapes del cicle de vida d'un projecte de desenvolupament de software. Aquesta importància es magnifica en l'àmbit dels contractes intel·ligents, on el codi és públicament accessible i aquests gestionen valors econòmics de manera nativa, augmentant els riscos de seguretat i la probabilitat que els atacants descobreixin vulnerabilitats. A continuació, es descriuen aquests mètodes, agrupats en quatre categories principals, detallant les característiques de cada un. No obstant això, cap d'ells per si sol és suficient per garantir la seguretat completa. Per tant, es recomana una combinació de tots aquests mètodes per assegurar la màxima qualitat i robustesa en termes de seguretat, així com la protecció contra la inclusió de vulnerabilitats i/o errors en entorns de producció.

Testeig Unitari (*Unit Testing*)

El testeig unitari [9] implica escriure codi específic per provar diferents parts (unitats) dels contractes. Els desenvolupadors creen tests per verificar cada funció o mòdul de manera aïllada per a assegurar-se que funcionin correctament sota diferents condicions. Encara que es pot aconseguir una alta cobertura amb aquests mètodes, les proves es realitzen generalment amb valors concrets que només cobreixen els *happy paths*. Un *happy path* es refereix a un escenari de prova que segueix el flux d'execució esperat, sense errors ni excepcions

Si visualitzem com una línia recta l'espai de totes les possibles entrades per a una funció específica, el testeig unitari només provaria un petit conjunt de punts d'aquesta línia. Aquesta limitació fa que, regularment, els *edge cases* més interessants es passin per alt en el testeig unitari.

Com a exemple d'eines de testeig unitari de contractes intel·ligents EVM es poden destacar les següents:

- **HardHat** [10]: *Framework* de desenvolupament de contractes intel·ligents escrit en **JavaScript** i que corre sobre **Node.js**. Aquesta configuració facilita una integració eficient amb multitud de paquets npm (gestor de paquets de Node.js), incloent paquets de testeig unitari com **Chai** i **Mocha**, entre altres.
- **Foundry** [11]: *Framework* de desenvolupament de contractes intel·ligents escrit en **Rust**, fet que el fa altament confiable i ràpid. Aquest *framework* inclou una llibreria anomenada **forge-std** que permet la definició dels tests en **Solidity**, generalment el mateix llenguatge en el qual estan escrits els contractes. Això redueix la fricció en l'escriptura dels tests, ja que elimina la necessitat de canviar de context entre llenguatges per als desenvolupadors.

Revisió Manual (Manual *Review)*

La revisió manual consisteix en l'inspecció detallada del codi per part d'experts en seguretat, que busquen patrons de codi problemàtics, errors lògics i altres vulnerabilitats que altres mètodes podrien passar per alt. Aquesta revisió sol ser profunda i permet la detecció de qualsevol tipus d'errors, inclús en lògiques complexes. El nivell de qualitat d'aquestes revisions dependrà de l'expertesa dels revisors, sent els més reconeguts els més cars del sector. Els principals inconvenients d'aquestes revisions és que tenen una llarga duració i solen tenir preus alts. A més a més, només tenen efecte sobre una versió específica del codi, fet que fa que es vegi invalidada després de la inclusió de qualsevol canvi i només es realitzin durant les fases properes al desplegament dels contractes a producció.

En el grup de les revisions manuals s'engloben els següents serveis:

- **Revisió de seguretat (Security Review)**: Són revisions que típicament són realitzades per empreses especialistes o revisors independents, en les quals es realitza un informe de seguretat i es proposen els canvis necessaris per a reparar els problemes dels contractes. En el mercat existeix una gran quantitat d'empreses que realitzen aquests serveis, amb preus proporcionals a la seva reputació. Com a exemple, podem nombrar **OpenZeppelin** [12], **Trail of Bits** [13] o **SpearBit** [14].

- **Competicions Publiques (Public Contests)**: Són competicions organitzades per diferents plataformes que duren un cert temps, on experts en seguretat cerquen errors en els contractes a canvi d'una retribució econòmica en funció de l'impacte de la vulnerabilitat. Com a exemple d'aquestes plataformes es poden nombrar **Code4rena** [15], **Cantina** [16] o **Sherlock** [17].
- **Bug Bounty**: A diferència de les anteriors, les campanyes de Bug Bounty es realitzen amb contractes que ja es troben en producció i solen ser de duració indefinida. L'objectiu d'aquestes campanyes és incentivar als hackers *White Hat* perquè cerquin i reportin vulnerabilitats en contractes en producció a canvi d'una retribució econòmica. La plataforma principal és **Immunefi** [18].

Anàlisis Completament Automatitzada *(Fully Automated Analysis)*

Aquest conjunt engloba tot el set d'eines que no requereixen cap intervenció humana directa i són fàcilment integrables en el cicle de vida del producte. Majoritàriament són eines d'anàlisi estàtic que escanejen el codi font buscant vulnerabilitats conegudes, errors de codi i altres problemes de seguretat. Tenen l'avantatge de ser ràpides i fàcilment integrables; no obstant això, poden generar falsos positius, necessiten actualitzacions regulars per detectar noves vulnerabilitats i només poden detectar patrons de vulnerabilitats prèviament coneguts. Com a exemple d'aquestes eines es pot nombrar **Code Inspector** [19], **Slither** [20] o **Aderyn** [21]

Anàlisis Semi-Automatitzada (*Semi Automated Analysis*)

L'anàlisi semi-automatitzada combina elements de revisió manual amb eines automatitzades. Requereix una intervenció humana inicial per a la construcció d'un *setup ad hoc*, però després aquest es pot integrar en el cicle de vida del producte. En aquest grup podem englobar tant eines de *fuzzing* com **Echidna** [22], **Foundry** o **Medusa** [23], com eines de verificació formal com **Certora Prover** [24] o **Halmos** [25]. A diferència de les revisions manuals, aquests mètodes permeten garantir les propietats del sistema en qualsevol versió dels contractes, ja que una vegada realitzada la inversió inicial per a la construcció del *setup*, aquest es pot utilitzar de manera automàtica. A més a més, permeten descobrir errors desconeguts que trenquen les propietats dels contractes mitjançant el testeig per a un gran nombre d'entrades possibles, maximitzant la detecció d'*edge cases*.

3.2. *Fuzzing* tradicional vs *Fuzzing* de contractes intel·ligents.

En programes tradicionals, el *fuzzing* té com a objectiu principal provocar l'error del sistema mitjançant el descobriment d'*edge cases* que puguin fer-lo fallar. Essencialment, es busca **trencar el sistema** per identificar problemes com desbordaments de memòria o errors de lògica que trenquin el programa en temps d'execució.

Els contractes intel·ligents es poden veure com a màquines d'estat perpetues, les quals disposen d'un codi i d'una màquina virtual per executar-lo, així com d'un espai d'emmagatzematge permanent anomenat s*torage* que representa l'estat del contracte. Les transferències d'estat es realitzen mitjançant la crida de funcions del codi (a través de transaccions de la *blockchain*), les quals modifiquen el *storage* de manera diferent, depenent de la lògica específica del codi de cada contracte. Aquesta execució es realitza de manera replicada en tots els nodes de la *blockchain*, seguint les regles d'execució estipulades per consens de la màquina virtual. Aquest fet implica que **els contractes intel·ligents no tenen errors inesperats en temps d'execució**, i qualsevol execució registrada a la *blockchain* és correcta, final e immutable, reflectint el principi «*code is law*» [26]. La Figura 6 representa un esquema conceptual del flux d'execució d'un contracte intel·ligent.

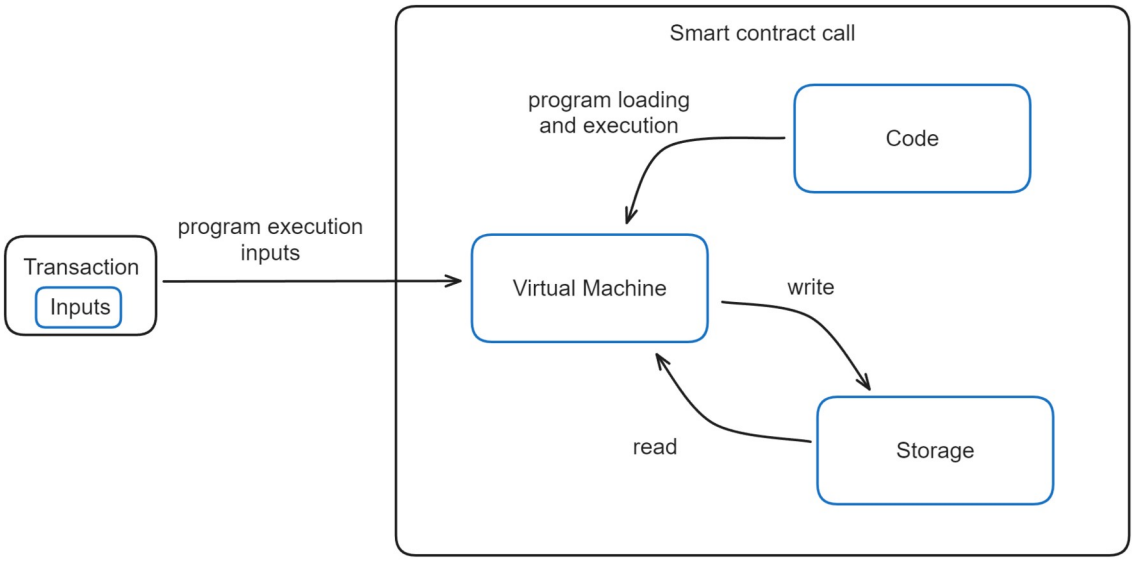

Figura 6 Esquema conceptual d'execució d'un contracte intel·ligent.

El *fuzzing* aplicat a contractes intel·ligents no se centra tant en provocar fallades del sistema, ja que **els contractes intel·ligents no fallen** en el sentit tradicional. Encara que les transaccions puguin fallar, això no implica necessàriament una fallada del contracte intel·ligent en si.

3.3. Testeig basat en propietats

En lloc de buscar fallades, el *fuzzing* en contractes s'orienta cap a la validació de propietats específiques del sistema. Això es fa mitjançant el que es coneix com a **testeig basat en propietats**, on es defineixen certes propietats o

invariants que sempre han de ser certes, independentment de les entrades proporcionades i de l'estat actual del contracte.

El *fuzzer* intenta validar si aquestes propietats es mantenen o no, cosa que equival a verificar si el sistema pot arribar a estats computacionalment correctes però lògicament invàlids respecte a les propietats teòriques del sistema. Aquest enfocament ajuda a assegurar que el contracte mantindrà les seves propietats de disseny davant qualsevol execució possible.

3.4. Fuzzing d'estats (*stateful fuzzing*)

Degut a la naturalesa perpètua dels contractes intel·ligents, una aproximació convencional de *fuzzing* no seria efectiva, ja que l'objectiu és verificar les propietats del sistema no només davant qualsevol entrada sinó també en qualsevol qualsevol combinació d'estat possible.

El fuzzing d'estats, o *stateful fuzzing*, és una tècnica que es basa en testejar els contractes intel·ligents utilitzant corpus d'entrades aleatòries. No obstant això, en lloc de testejar cada entrada amb un estat inicial del contracte idèntic, s'executen en un ordre aleatori contra l'estat resultant de l'execució de l'entrada aleatòria anterior. L'objectiu d'aquesta tècnica és testejar les propietats dels contractes mentre es simula l'activitat real generada per les transaccions dels usuaris que el contracte experimentarà una vegada desplegat en producció. La Figura 7 mostra la diferència entre testejar els contractes amb mètode *stateful* i *stateless*, que seria el cas contrari.

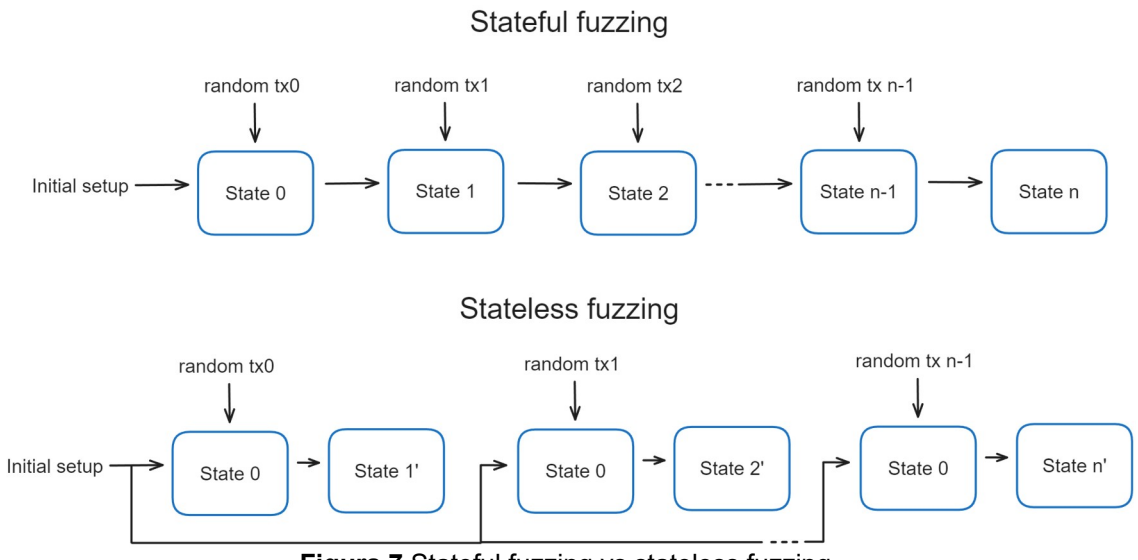

Figura 7 Stateful fuzzing vs stateless fuzzing.

3.5. Estat del Art

La majoria de contractes intel·ligents desplegats a producció són *open source* per diverses raons clau. Com que gestionen actius digitals valuosos, és crucial que siguin transparents. Quan el codi és obert, qualsevol persona pot revisar-lo i verificar que el contracte compleix amb el que promet, cosa que ajuda a construir confiança entre els usuaris. A més, això permet que experts en seguretat i altres desenvolupadors examinin el codi en busca de vulnerabilitats i errors. Per tant, és fonamental per a la reputació dels projectes de blockchain que el codi dels seus contractes sigui *open source*. Tan és així que fins i tot existeixen serveis de verificació com **Etherscan** [27], els quals verifiquen públicament que un codi font correspon al *bytecode* desplegat a la *blockchain*.

La naturalesa de la cadena de blocs fa que el *bytecode* dels contractes sigui intrínsecament públic. Encara que això no és el mateix que tenir accés al codi font complet en Solidity (o altres llenguatges d'alt nivell), existeixen eines fer enginyeria inversa o decompilar els *bytecodes* a Solidity.

Per aquestes raons, totes les eines de *fuzzing* de contractes intel·ligents existents en el mercat adopten el mètode de ***White-Box***.

Actualment, existeixen principalment tres eines de *fuzzing* per a contractes intel·ligents en el mercat. A continuació, es presenten les tres eines, descrivint lleugerament les propietats de cadascuna d'elles.

Foundry

Desenvolupat per l'empresa **Paradigm**, **Foundry** no és només una eina de *fuzzing,* és un *framework* de desenvolupament de contractes intel·ligents per a Ethereum, que inclou diverses eines. Entre elles està **Forge**, un *framework* de testeig per a contractes Ethereum, amb funcionalitats de *fuzzing* i testeig de propietats integrades. Està escrit en Rust i és conegut per la seva modularitat, portabilitat i rapidesa. A més, inclou la llibreria **forge-std**, que ofereix una àmplia gamma d'utilitats de testeig per a contractes en Solidity, utilitzant el mateix llenguatge, Solidity.

Echidna

Desenvolupat per l'empresa Trail of Bits, és una eina de *fuzzing* basada en propietats per al testeig de contractes Ethereum. Està escrita en Haskell i disposa d'una UI interactiva en la terminal, que permet monitorar la campanya de *fuzzing* en temps real. És coneguda per ser la primera solució que permet la realització professional de campanyes de *fuzzing* en contractes Ethereum. La Figura 8 mostra la UI interactiva d'Echidna durant una campanya de *fuzzing*.

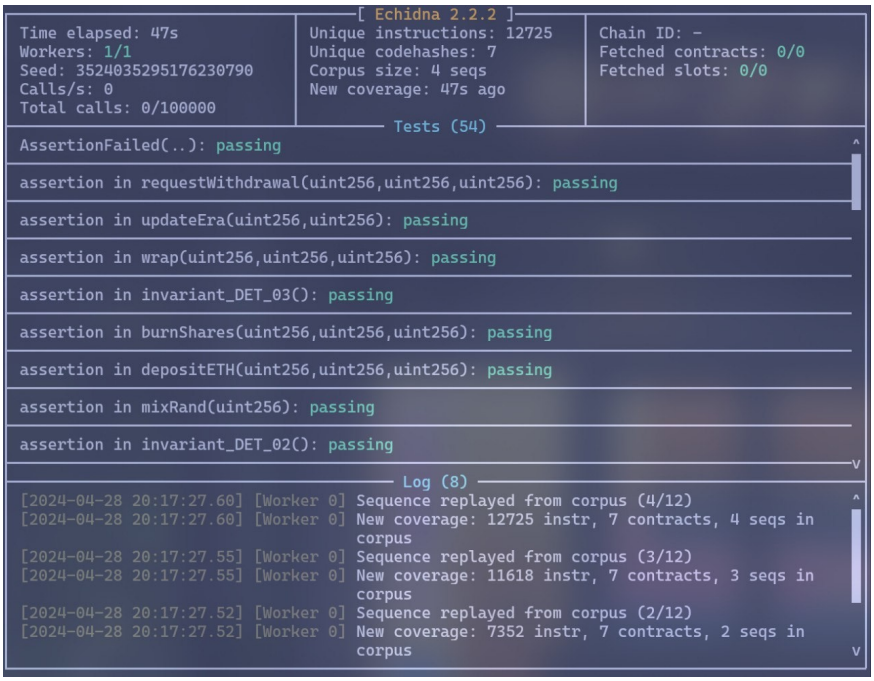

Figura 8 Exemple de UI d'Echidna.

Medusa

També desenvolupada per l'empresa Trail of Bits, és una eina de *fuzzing* basada en propietats per al testeig de contractes intel·ligents d'Ethereum, que pretén ser una millora d'Echidna. Està escrita en Go i és completament compatible amb *setups* escrits per a Echidna. A més, és coneguda per les seves altes capacitats de paral·lelització que permeten executar una gran quantitat de tests simultàniament.

3.6. Propietats de les eines de *fuzzing* de contractes

Encara que cadascuna d'elles té detalls d'implementació diferents, configuracions i estan escrites en llenguatges diferents, la manera en què s'utilitzen és molt similar. A continuació, es descriuen els punts en comú de les eines i les funcionalitats de les quals disposen.

Instància d'Ethereum local amb cheatcodes

Per al desplegament i testeig del *SUT*, les eines de *fuzzing* arranquen una instància local de la xarxa Ethereum. Això proporciona un entorn d'execució pràcticament idèntic a la xarxa real, però permet testejar els contractes de manera ràpida i eficient, ja que l'execució es realitza únicament a la màquina local en lloc d'estar replicada a tots els nodes de la xarxa. L'estat d'aquesta xarxa local només viu durant la campanya de *fuzzing*. Un gran avantatge d'aquest tipus de solució és que, a més de permetre'ns desplegar els contractes que volem testejar, també ens permet escriure els *harness* en el mateix llenguatge dels contractes intel·ligents, reduint la fricció a l'hora de desenvolupar *setups* de testeig. Bàsicament, els *harness* seran també contractes intel·ligents que s'utilitzaran com a interfície entre el *fuzzer* i el *SUT*,

i permetran tant la personalització dels mecanismes de lliurament com la definició d'oracles per a verificar les propietats del *SUT*.

Aquestes instàncies locals també incorporen diverses utilitats anomenades *cheatcodes*, que permeten realitzar accions específiques durant els tests, com modificar la xarxa local, el número de bloc o el *timestamp* de la *blockchain*, així com eines per a impersonar o finançar comptes. Aquestes utilitats són accessibles mitjançant la interacció entre contractes, de manera que es poden utilitzar en els *harness*. No totes les eines disposen dels mateixos cheatcodes, **Foundry** és la més completa en aquest sentit. No obstant això, hi ha un subconjunt d'aquests que sí comparteixen totes tres eines. Si volem construir un *setup* compatible amb les tres, estem limitats únicament a l'ús d'aquest subconjunt.

La taula següent mosta alguns *cheatcodes* a mode d'exemple i compara la seva disponibilitat en cada eina:

cheatcode	Descripció	Foundry	Echidna	Medusa
prank	Permet impersonar qualsevol direcció en la instància d'Ethereum local	O	O	O
deal	Permet financiar amb ETH qualsevol direcció en la instància d'Ethereum local	O	O	O
warp	Permet modificar el número de bloc de la instància d'Ethereum local	O	O	O
roll	Permet modificar el *timestamp* de la instància d'Ethereum local	O	O	O
etch	Permet modificar el bytecode de qualsevol direcció en la instància d'Ethereum local	O	X	O
record	Permet enregistrar totes les interaccions amb el *storage* dels contractes.	O	X	X
fee	Permet modificar la *fee* de la xarxa de la instància d'Ethereum local	O	X	X

Selecció de contractes objectiu

Per a realitzar un *fuzzing* selectiu, les eines permeten seleccionar de manera granular els contractes i funcions que volem atacar. Això facilita realitzar un desplegament complex del *SUT*, tot configurant una campanya de *fuzzing* selectiva i localitzada. Aquesta configuració ens permet canalitzar les entrades generades pel *fuzzer* a través del *harness*, obtenint així un alt nivell de personalització en les campanyes.

Foundry permet aquesta configuració a través de les següents funcions de la llibreria **forge-std**:

- **excludeContract(address newExcludedContract_)**
- **targetContract(address newTargetedContract_)**

Echidna permet aquesta configuració amb l'ús de l'argument **--contract <Nom del contracte objectiu>**.

Medusa permet aquesta configuració mitjançant el paràmetre **targetContracts** del fitxer de configuració **medusa.json**.

Definició de propietats o invariants

Les propietats o invariants són propietats del *SUT* que han de mantenir-se sempre certes al llarg de la campanya de *fuzzing*. Aquestes s'expressen com a funcions Solidity i han de ser definides de manera *ad hoc* pel desenvolupador, en funció de les propietats del *SUT* que es volen testejar. El *fuzzer* testarà aquestes funcions durant la campanya, a més d'atacar les funcions del *SUT*.

Foundry permet la definició d'invariants utilitzant el prefixe **invariant_**, les invariants que fallin hauran de revertir. Les assercions de la llibreria forge-std es poden utilitzar per aquest propòsit. La Figura 9 mostra un exemple d'invariant utilitzant la sintaxi de **Foundry**. En aquest cas, l'invariant assegura que en cap moment el balanç d'un usuari pot superar l'emissió total d'un token que s'està testejant.

```
function invariant_example() external {
    assertLte(
        token.balanceOf(msg.sender),
        token.totalSupply(),
        "User balance higher than total supply"
    );
}
```

Figura 9 Exemple d'invariant amb **Foundry**.

En canvi, a Echidna i Medusa, les invariants s'expressen com funcions que retornen un valor booleà. Echidna utilitza el prefix **echidna_** mentre que en Medusa és configurable, amb un valor per defecte de **fuzz_**. La Figura 10 mostra la mateixa invariant de la Figura 9, però expressada amb la sintaxi d'Echidna/Medusa.

```
function echidna_example() external returns(bool) {
    return (token.balanceOf(msg.sender) <= token.totalSupply());
}
```

Figura 10 Exemple d'invariant amb Echidna/Medusa.

Fuzzing sintàctic a partir de l'ABI dels contractes

En contractes intel·ligents Solidity, la **ABI** fa referència a l'*Application Binary Interface* (Interfície Binària d'Aplicació). L'ABI és una especificació que descriu com les dades i les funcions d'un contracte intel·ligent es representen en binari.

Cada contracte té una ABI que defineix l'interfície pública del contracte. Aquesta interfície especifica com es codifiquen les funcions del contracte per a poder-les cridar selectivament en temps d'execució i com s'han de codificar els arguments de les funcions i les dades de retorn. En la ABI també s'especifica altres elements de comunicació externa com esdeveniments o errors.

L'ABI és generada pel compilador, i es pot trobar en el *Compilation Artifact*, fitxer .json resultant de la compilació. Aquest fitxer conté, entre altres elements, el *bytecode* per desplegar el contracte a la blockchain i l'ABI. La Figura 11 mostra un contracte d'exemple i la seva ABI. Com es pot observar, aquesta defineix el nom de la funció add i els tipus d'arguments acceptats per aquesta.

```
// SPDX-License-Identifier: GPL-3.0

pragma solidity ^0.8.0;

contract Test {
    function add(uint256 a, uint256 b) external pure returns (uint256){
        return a + b;
    }
}
```

```
"abi": [
    {
        "inputs": [
            {
                "internalType": "uint256",
                "name": "a",
                "type": "uint256"
            },
            {
                "internalType": "uint256",
                "name": "b",
                "type": "uint256"
            }
        ],
        "name": "add",
        "outputs": [
            {
                "internalType": "uint256",
                "name": "",
                "type": "uint256"
            }
        ],
        "stateMutability": "pure",
        "type": "function"
    }
]
```

Figura 11 Contracte exemple i ABI resultant.

L'ABI és utilitzada pel fuzzer, ja que defineix directament la sintaxi gramatical que aquest utilitza per generar entrades aleatòries que produeixin fluxos d'execució vàlids en els contractes testejats.

Mutació del corpus basat en anàlisis de cobertura

Per a maximitzar la cobertura del codi i garantir que es generen entrades que cobreixen tots els camins d'execució possibles dels contractes testejats, els conjunts de trucades que augmenten la cobertura són emmagatzemats al corpus com a entrades interessants. A partir d'aquí, es generen nous tests mitjançant la mutació de les entrades inicials proporcionades. Si una nova mutació explora un camí nou (produeix un augment de la cobertura total acumulada), el *fuzzer* categoritza aquesta entrada com a interessant i l'afegeix al corpus.

Informe de cobertura

L'informe de cobertura permet al desenvolupador que està construint el *setup* de *fuzzing* avaluar les línies del contracte que s'han durant la campanya de *fuzzing*. El desenvolupador pot identificar quines línies cobreix el *setup* e iterar per a conduir el fuzzer cap a les línies no cobertes. L'informe de cobertura proveeix una garantia que els tests s'executen de manera esperada.

La Figura 12 mostra un fragment d'un informe de cobertura generat amb l'eina Medusa a mode d'exemple. En l'informe es pot observar un fragment del *SUT*, el qual és una funció. Es pot veure clarament que les línies en verd s'han executat durant la campanya de *fuzzing* i la línia en vermell no. Aquest fet indica al desenvolupador encarregat de construir el *setup* de testing que no s'està cobrint una part del codi, i aquest haurà de modificar els mecanismes de lliurament per intentar cobrir també aquesta línia.

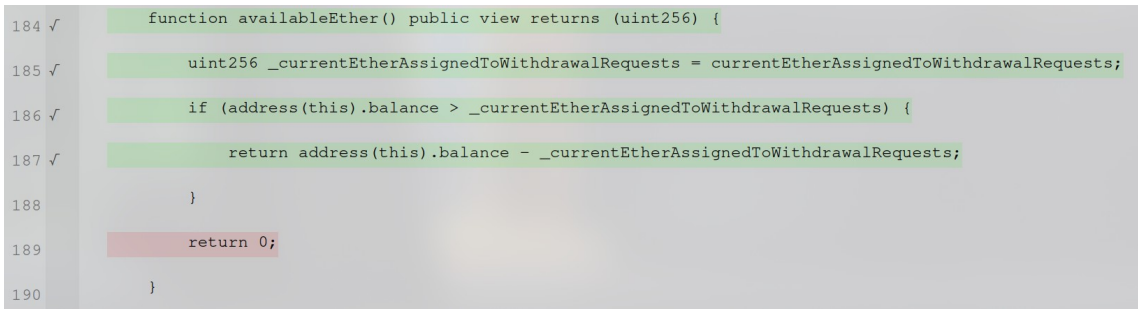

Figura 12 Exemple d'informe de cobertura generat per l'eina Medusa.

3.7. Arquitectura efectiva d'un *setup* de fuzzing de contractes intel·ligents

Com s'ha anunciat anteriorment, un *setup* de fuzzing està principalment format per tres components: el SUT, el Fuzzer i el Harness. En el cas d'un *setup* de fuzzing de contractes intel·ligents, l'eina de Fuzzing conté el Fuzzer i altres funcionalitats adicionals, com la traçabilitat de la cobertura o el desplegament d'una instància local d'Ethereum amb cheatcodes per allotjar el SUT en temps d'execució durant la campanya de fuzzing. El Harness també es desplegarà en la instància local i permetrà als desenvolupadors del *setup* escriure la implementació dels mecanismes de lliurament i els oracles (invariants i assercions) en el mateix llenguatge en què està escrit el SUT, en aquest cas Solidity.

La Figura 13 mostra l'arquitectura general proposada que es pot aplicar per a qualsevol *setup* de fuzzing de contractes. Es pot observar que s'han afegit dos components addicionals: actors i funcions objectiu, els quals es detallen a continuació.

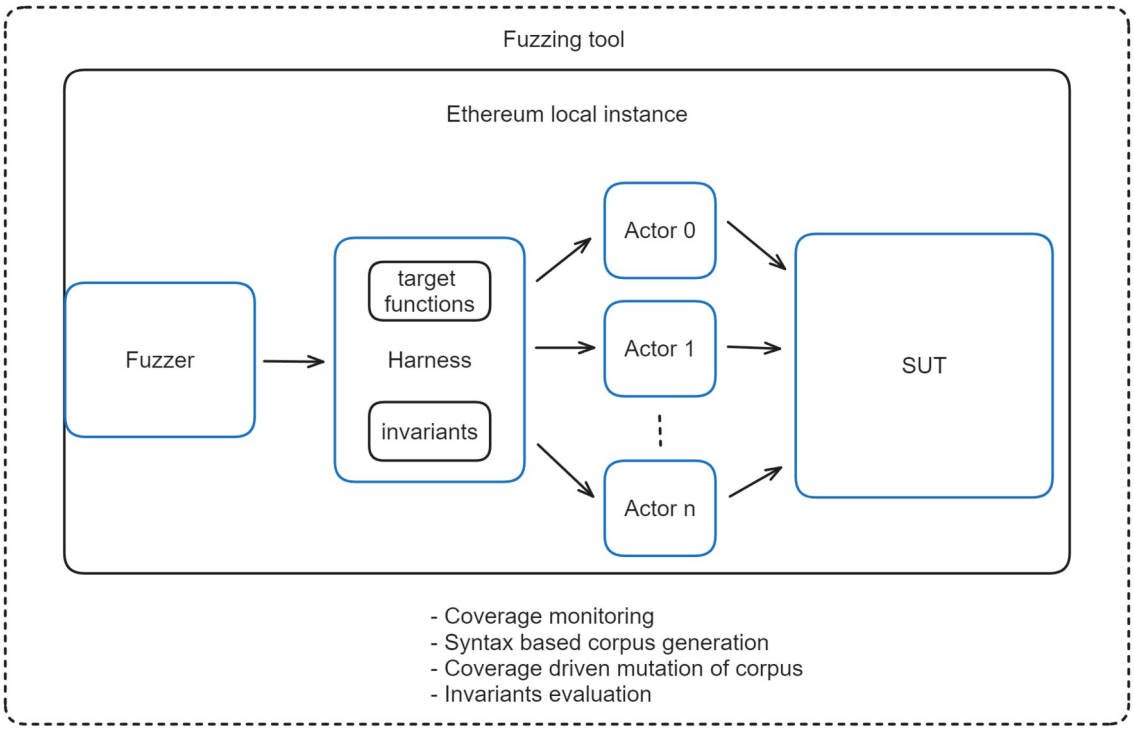

Figura 13 Arquitectura d'un *setup* de *fuzzing* de contractes intel·ligents.

Actors

Els actors rebran les crides de les *target functions* i les executaran. Els actors són petits contractes la única funcionalitat dels quals és simular l'activitat de diferents usuaris. Aquests es trien de manera aleatòria cada cop que el *fuzzer* cria a una *target function*. En Ethereum, cada contracte intel·ligent té una adreça d'Ethereum única, fet que fa que una mateixa funció del *SUT* sigui cridada aleatoriament per adreces d'Ethereum diferents (actors diferents) durant la campanya de fuzzing, simulant així l'activitat de diversos usuaris.

La Figura 14 mostra la implementació mínima d'un actor. Es pot observar que conté una funció anomenada **proxy** que rep com a paràmetre una adreça **_target** i una cadena de bytes **_calldata**. A continuació, realitza una crida de baix nivell a l'adreça **_target** (que serà un contracte del SUT) amb el paràmetre **_calldata** com a argument. Aquesta funció permetrà a la *target function* que cridi a l'actor executar qualsevol crida arbitrària a qualsevol adreça en nom de l'actor específic.

```
1  pragma solidity ^0.8.0;
2
3  contract Actor {
4      function proxy(address _target, bytes memory _calldata)
5          external
6          payable
7          returns (bool success, bytes memory returnData)
8      {
9          (success, returnData) = address(_target).call{value: msg.value}(_calldata);
10     }
11
12     receive() external payable {}
13 }
```

Figura 14 Implementació mínima d'un actor.

Target Functions

Les *target functions* són funcions Solidity del *Harness* que rebran les entrades aleatòries del *fuzzer*, s'encarregaran de filtrar-les i cridaran la seva funció mirall en el *SUT* a través dels actors.

Una *target function* generalment seguirà la següent estructura:

- **Pre-condicions:** Accions necessàries a realitzar abans de la crida al SUT, com per exemple el filtratge de valors aleatoris o la selecció d'actors.
- **Acció:** Acció en concret que s'està testejant, generalment una crida al SUT a través d'un actor aleatori.
- **Post-condicions:** Condicions necessàries que s'han de complir un cop s'ha realitzat la crida al SUT, com per exemple assercions que asseguren que el resultat de l'acció és l'esperat.

La Figura 15 mostra una *target function* d'exemple per a testejar la funció **transfer** d'un token estàndard ERC20 [28]. El primer que hem de tenir en compte és que aquesta funció serà cridada pel *fuzzer*, el qual entén la sintaxi dels arguments d'aquesta. Per tant, serà capaç de generar valors aleatoris per als tipus d'arguments de la funció, en aquest cas, enters **uint256** i **uint128**.

Entre les línies 3 i 7 es realitzen les accions prèvies a l'execució del test. En aquest cas, es seleccionen dos actors consecutius aleatoris utilitzant un valor generat pel *fuzzer* com a llavor, es capturen els balanços previs a realitzar l'acció per als dos actors i es constreny el valor de la quantitat a enviar a un

màxim del balanç de l'actor que realitzarà l'acció utilitzant l'operador mòdul (%). Aquesta darrera operació assegurarà que l'actor tindrà suficient balanç per a realitzar l'acció de manera exitosa. Aquest és un exemple de filtratge de valors aleatoris per a produir fluxos d'execució rellevants en el *SUT*.

Entre les línies 10 i 13 es realitza la crida a la funció **proxy** de l'actor seleccionat, el qual cridarà la funció **transfer** del token.

Finalment, a les línies 16 i 17 es verifica si els balanços s'han transferit de manera correcta; en cas contrari, el test fallaria.

```
1   function transfer(uint256 amount, uint128 i) public {
2       // Pre-conditions
3       Actor actor = getRandomActor(uint256(i));
4       Actor receiver = getRandomActor(uint256(i) + 1);
5       uint256 actorInitialBalance = token.balanceOf(address(actor));
6       uint256 receiverInitialBalance = token.balanceOf(address(receiver));
7       amount = amount % token.balanceOf(address(actor));
8
9       // Action
10      (bool result,) = actor.proxy(
11          address(token), abi.encodeWithSignature("transfer(address,uint256)", address(receiver), amount)
12      );
13      assert(result);
14
15      // Post-conditions
16      assert(actorInitialBalance - amount == token.balanceOf(address(actor)));
17      assert(receiverInitialBalance + amount == token.balanceOf(address(receiver)));
18  }
```

Figura 15 Exemple de *target function* del *Harness* per a testejar la funció **transfer** d'un token ERC20.

4 Resultats

En aquest apartat, es presenta els resultats de l'aplicació dels mètodes descrits anteriorment en un exemple real: el protocol Diva Staking [29]. Mitjançant l'ús de les tècniques de *fuzzing* descrites en els apartats anteriors, es procedirà a analitzar la robustesa dels contractes de Diva Staking, identificant possibles vulnerabilitats i errors que podrien comprometre la seguretat del sistema.

4.1. Descripció Protocol Diva Staking

En l'ecosistema Ethereum, el s*taking* [30] és l'acte de dipositar i bloquejar 32 ETH per activar un **validador**. Un validador és una entitat virtual d'Ethereum que participa en el consens del protocol. Els validadors estan representats per un balanç i una clau pública, entre altres propietats. Un node validador és una combinació de maquinari i programari que executa una instància de la xarxa Ethereum i que participa activament en el consens protocol.

Els validadors són responsables d'emmagatzemar dades, processar transaccions i afegir i validar nous blocs a la cadena. El sistema d'incentius està dissenyat de tal manera que, en cas que els validadors no actuïn d'acord amb les regles del protocol, se'ls penalitza econòmicament amb amb una fracció de l'ETH que tenen en *staking*, un fenomen conegut com a *slashing [31]*. En canvi, si participen de manera correcta, obtindran una recompensa. Aquesta mecànica desincentiva l'existència de validadors maliciosos i permet que la xarxa d'Ethereum sigui fiable i segura.

Tot i que inicialment bloquejar 32 ETH no constituïa un cost prohibitiu per a qualsevol persona interessada en allotjar i operar un node validador i participar activament en el consens, l'apreciació recent de la moneda (3500-4000 per unitat d'ETH en el moment d'escriure aquest treball) ha fet que aquest cost esdevingui prohibitiu. Com a solució a aquest problema i amb l'objectiu de democratitzar el *staking* d'ETH, han sorgit diferents protocols de *pooled staking*. Aquests protocols permeten als usuaris gaudir d'una part de les recompenses del *staking* mitjançant el dipòsit dels seus ETH en *pools*, on es combinen amb els ETH d'altres usuaris. Els protocols de *pooled staking* allotgen nodes validadors de manera permisionada utilitzant l'ETH de les *pools* dels usuaris per activar els validadors. Això permet que els usuaris gaudeixin dels beneficis del *staking* sense necessitat d'assolir el llindar de 32 ETH ni d'allotjar un node. En lloc d'això, confien en els protocols de *pooled staking* per a operar els nodes.

Tot i que aquests protocols permeten als usuaris obtenir recompenses d'*staking*, aquests no participen activament en el procés de validació i han de confiar en terceres entitats de confiança per a que aquestes operin els nodes validadors utilitzant l'ETH dels usuaris com a col·lateral d'*staking*.

El protocol **Diva Staking** ofereix una solució per democratitzar no només l'obtenció de recompenses del *staking*, sinó també l'operació d'un node validador.

Els validadors d'Ethereum autoritzen les seves accions en el protocol mitjançant un sistema de signatura digital Boneh–Lynn–Shacham [32] (BLS). Aquest criptosistema permet combinar múltiples claus públiques en una sola clau pública agregada, un procés conegut com a agregació de claus públiques. A més a més clau pública agregada és indistingible de qualsevol de les seves claus individuals. El sistema es pot configurar com un sistema BLS amb llindar, permetent constituir un sistema de signatura t-de-n, on almenys t participants d'un conjunt de n claus han de signar utilitzant una clau pública BLS agregada. Addicionalment, les claus es poden generar amb un sistema de Generació Distribuïda de Claus (DKG) que impedeix que un sol actor controli la clau privada agregada.

Addicionalment a les *pools* de *pooled staking*, Diva Staking proposa una capa d'abstracció addicional sobre el protocol Ethereum que permet que les tasques d'un validador no siguin realitzades únicament per un sol node validador, sinó per un subconjunt de nodes validadors Diva. Aquests nodes operen com un sol validador d'Ethereum mitjançant un sistema de signatura BLS amb llindar. Aquesta tecnologia s'ha denominat com a *Distributed Validator Technology* (DVT).

Diva proporciona les següents avantatges addicionals al protocol Ethereum sense comprometre la seguretat i la descentralització:

- ***Pooled Staking:*** Permet als usuaris depositar ETH en una *pool* i rebre recompenses sense la necessitat de depositar 32 ETH i operar un node validadors. Els usuaris reben a canvi del dipòsit un token anomenat divETH, el qual augmenta de valor amb el temps degut a les recompenses obtingudes del *staking*. Aquest fenomen es coneix com a *rebasing*.
- **Descentralització dels validadors:** En lloc de confiar únicament en una sola entitat operadora i un sol node validador (tant en programari com en maquinari) per a realitzar les tasques d'un validador d'Ethereum, les tasques d'un validador es repliquen en n nodes Diva, afegint redundància i permetent reduir la barrera dels 32 ETH. Els operadors de nodes Diva només hauran de dipositar un petit col·lateral d'almenys 1 divETH per a participar en el consens. A canvi, els operadors rebran recompenses del *staking* directament per part del protocol e indirectament amb el *rebasing* del divETH que tenen com a col·lateral.

La Figura 16 representa una comparativa entre les 3 generacions de *staking* que es poden diferenciar en el protocol Ethereum i les diferències conceptuals que introdueix el protocol Diva Staking. Com es pot observar, la diferència fonamental rau en la descentralització del node validador mitjançant l'ús de la tecnologia DVT i l'ús d'operadors no permisionats i sense necessitat de confiança.

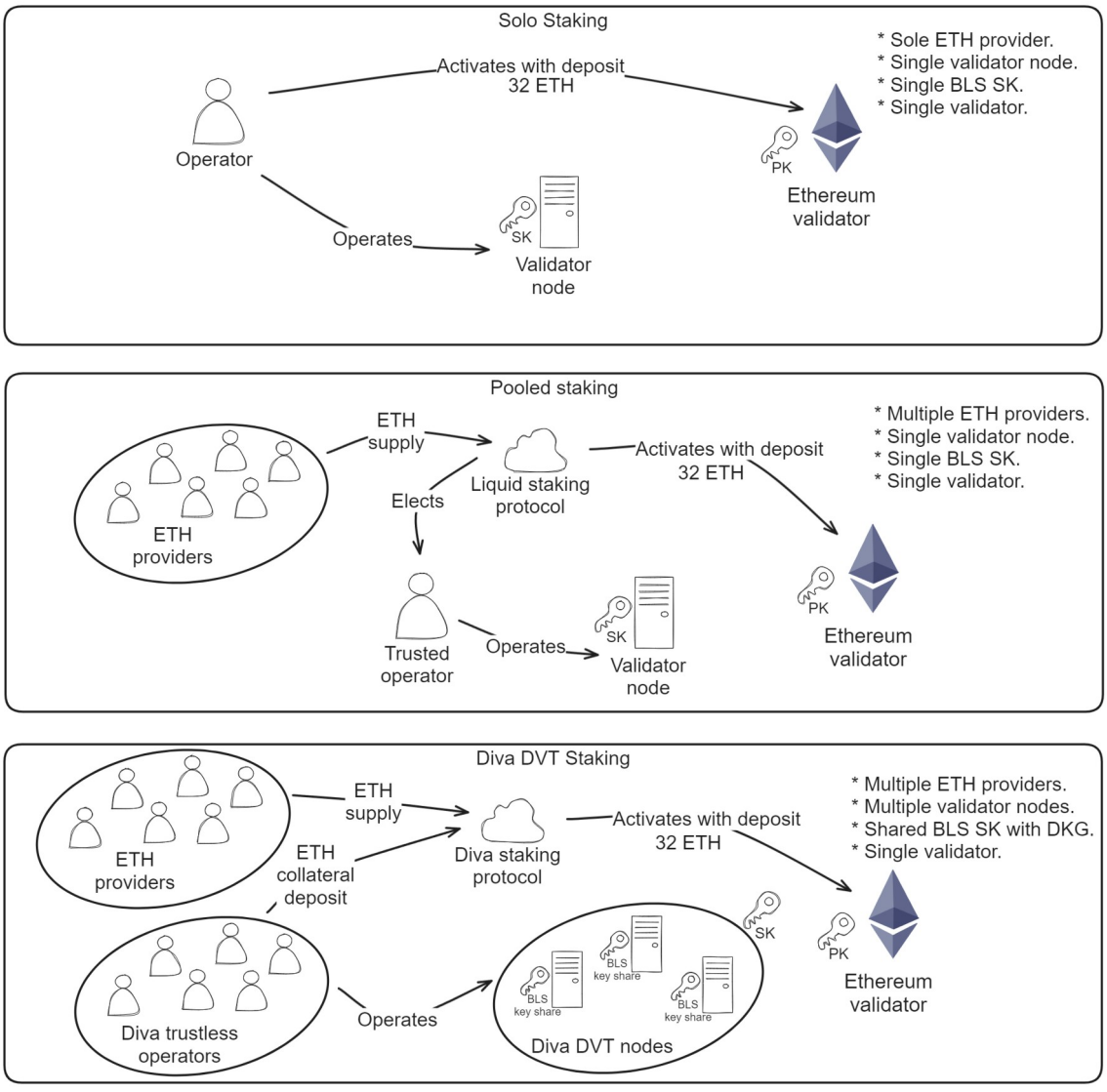

Figura 16 Comparativa conceptual entre *solo staking*, *pooled staking* i DVT proposat per D*iva*.

4.2. Contractes intel·ligents de Diva Staking

Tota la part del protocol que gestiona l'ETH i els validadors s'implementa mitjançant contractes intel·ligents, permetent la construcció d'un protocol descentralitzat i sense necessitat de tercers de confiança. De fet, per a activar un validador, els 32 ETH necessaris es dipositen en un contracte intel·ligent conegut com a Deposit Contract.

A continuació es presenta una taula on es descriuen els contractes principals del protocol.

Nom	**Descripció**
DivaEtherToken	Implementació de token amb interfície estàndard ERC20 anomenat **divETH**. El valor de divETH és sempre 1:1 respecte a ETH. El token disposa de funcionalitats per a dipositar ETH i encunyar divETH a canvi. Els usuaris que tenen divETH en el seu balanç són propietaris d'un cert nombre de participacions sobre l'ETH total dipositat en el protocol. Així, si el total d'ETH dipositat augmenta, les participacions dels usuaris representen una quantitat més gran d'ETH. Aquesta mecànica, coneguda com a rebasing, s'utilitza per repartir les recompenses del staking entre tots els usuaris.
AccountingManager	Gestiona la comptabilitat interna de l'ETH del protocol. Conté una funció que s'executa diàriament i que, en primer lloc, reparteix les recompenses de staking entre els usuaris mitjançant un rebasing del token divETH. A més, paga les recompenses als operadors enviant ETH al contracte OperatorsRewardsVault. També envia ETH al contracte ValidatorManager per a realitzar dipòsits al Deposit Contract i activar nous validadors, i finalment executa les sol·licituds de retirada enviant ETH al contracte WithdrawQueueEscrow.
ValidatorManager	S'encarrega d'interactuar amb el Deposit Contract i gestionar l'activació i desactivació de nous validadors.
OperatorManager	S'encarrega de gestionar el registre de nous operadors i nodes Diva al protocol.
CollateralCurve	S'encarrega de gestionar els col·laterals dels operadors. Els col·laterals són un component fonamental de la teoria de jocs que permet disposar dels nodes Diva sense necessitat de depositar confiança en els operadors. Els operadors estaran incentivats a actuar d'acord amb les normes del protocol, en cas contrari, arriscarien perdre el col·lateral dipositat necessari per a poder optar a ser operadors.
WithdrawQueueEscrow	S'encarrega de gestionar i executar les sol·licituds de retirada. Per a retirar ETH del

	protocol, els usuaris han d'intercanviar divETH per ETH i pagar una comissió. Aquest contracte s'encarrega de gestionar aquestes mecàniques.
ExecutionLayerRewardsVault	S'encarrega de rebre les recompenses del staking, aquestes despres son recollides d'aquest contracte pel AccountingManager.
OperatorsRewardsVault	S'encarrega de rebre les recompenses a pagar als operadors, les quals són distribuïdes per l'AccountingManager. El contracte reparteix aquestes recompenses als operadors. Els operadors poden retirar la seva recompensa en qualsevol moment cridant una funció en aquest contracte.
WrappedDivaEtherToken	Token ERC20 wrapper de DivaEtherToken, anomenat wdivETH, s'utilitza per embolicar el token divETH per garantir la compatibilitat amb protocols de tercers. Això es deu al fet que les mecàniques especials de divETH podrien ser incompatibles amb altres protocols.

4.3. *Setup* de Fuzzing

En aquest apartat es descriu el *setup* del fuzzing construït *ad hoc* per al protocol Diva Staking, aplicant les tècniques i eines descrites en els apartats anteriors. El *setup* consisteix principalment en un contracte monolític escrit en Solidity que actua com a h*arness*. Aquest contracte s'ha anomenat Target. Per evitar escriure tot el harness en un sol fitxer Solidity, s'ha construït de manera modular utilitzant la funcionalitat de composabilitat de Solidity coneguda com herència. D'aquesta manera, tot i que el harness és un contracte monolític, aquest hereta de diferents contractes pare que són modulars. Cada un d'ells realitza tasques diferents, autocontingudes, extensibles i fàcilment depurables.

Degut a l'extensió del protocol i la limitació de temps en la realització d'aquest treball, el *setup* només testeja els següents contractes. No obstant això, el *setup* és modular i fàcilment extensible, permetent així la seva ampliació per testejar el protocol complet en el futur.

- DivaEtherToken
- WrapedDivaEtherToken
- WithdrawQueueEscrow

A continuació es presenta una taula que descriu els contractes que formen el harness de fuzzing del protocol.

Nom	**Descripció**
TestStorage	Conté les variables d'estat utilitzades pel *harness*. Atès que estem realitzant *fuzzing* amb estat (*stateful fuzzing*), és necessari que el test tingui diferents variables d'estat que mantindran informació sobre el test entre transaccions. Un exemple d'ús seria tenir una comptabilitat paral·lela dels balanços dels tokens per assegurar que el contracte DivaEtherToken comptabilitza correctament els balanços o emmagatzemar les adreces del protocol desplegat en la xarxa de test.
CryticAsserts	Llibreria *d'assertions* específica per a les eines Echidna i Medusa, ja que aquest *setup* està dissenyat per ser utilitzat amb aquestes eines. L'abstracció de la interfície *d'assertions* permet definir una llibreria *d'assertions* amb la mateixa interfície que sigui compatible amb Foundry.
PropertiesDescriptions	Conté les propietats del protocol que volem verificar, escrites en llenguatge humà. Aquest contracte inclou una sèrie de cadenes de caràcters constants que descriuen les propietats (o invariants) que el *setup* de *fuzzing* està comprovant. S'utilitzen com a descripcions d'errors en cas que alguna d'elles falli.
TestUtils	Llibreria que conté diverses utilitats emprades durant el test.
<NomContracte>Properties	Conté les implementacions de les propietats (o invariants) d'un dels contractes que s'està testejant. El *setup* inclou un contracte d'aquest tipus per a cada contracte del protocol que estem testejant.
BeforeAfter	Conté *hooks* que s'executen abans i després de cridar el *SUT*. Això és útil per abstraure funcionalitats i evitar sobrecarregar les *target functions* amb codi repetit. Un exemple d'ús podria ser l'actualització de les variables d'estat del contracte TestStorage o el reinici de les variables internes del *SUT*.
PropertiesAggregator	Contracte simple, sense funcionalitat, que agrega tots els contractes <NomContracte>Properties.

<NomContracte>Handler	Conté les implementacions de les *target functions* d'un dels contractes que s'està testejant. El *setup* inclou un contracte d'aquest tipus per a cada contracte del protocol que estem testejant.
HandlersAggregator	Contracte simple, sense funcionalitat, que agrega tots els contractes <NomContracte>Handler.
Setup	Script que s'executarà inicialment per desplegar el protocol a la xarxa de test. Tota la inicialització del *SUT* està continguda en el constructor d'aquest contracte.
CryticProperties	Contracte adaptador que adapta totes les propietats agregades al PropertiesAggregator per tal que siguin compatibles amb la sintaxi d'invariants de les eines Echidna i Medusa. Si es vol fer el setup compatible amb Foundry, caldria definir un contracte adaptador específic per a fer les propietats compatibles amb la sintaxi de Foundry.
Tester	Contracte simple, sense funcionalitat, que agrega tots els contractes del *setup* de *fuzzing* i que s'utilitzara com a *target* per a l'eina de *fuzzing*.

La Figura 17 mostra els contractes que constitueixen el *harness* i la cadena d'herències proposada. Com es pot observar, el *setup* és fàcilment extensible a tots els contractes del protocol. Només cal afegir els contractes <NomContracte>Properties i <NomContracte>Handler corresponents a cada contracte nou, sense necessitat de modificar l'arquitectura del *harness*.

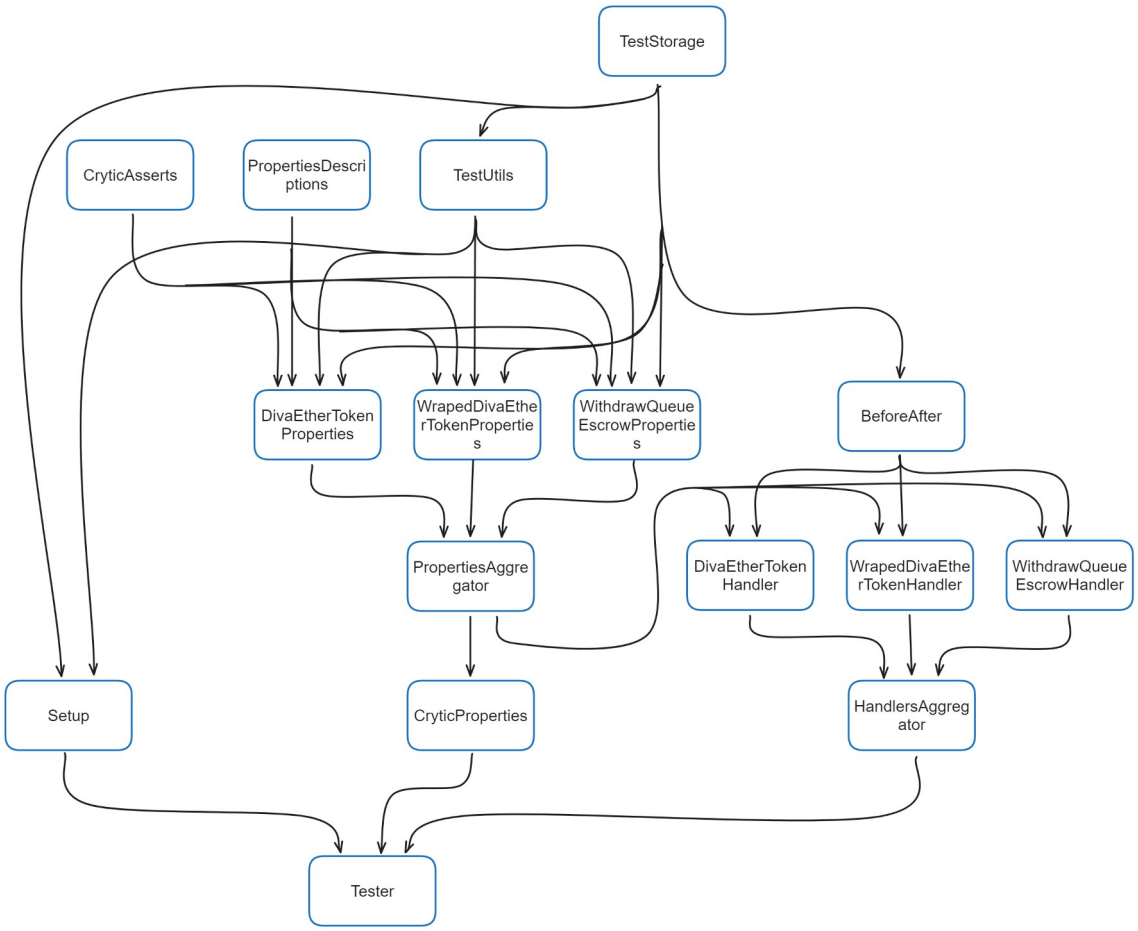

Figura 17 Cadena d'herència del contracte *Harness* Tester.

4.4. Propietats afirmades

Les següents taules mostren totes les propietats dels contractes del protocol Diva Staking que han estat afirmades pel *setup* de *testing*.

Contracte DivaEtherToken	
DET-01	User balance must not exceed total supply
DET-02	Sum of users balance must not exceed total supply.
DET-03	Self transfers should not break accounting (transfer).
DET-04	Self transfers should not break accounting (transferFrom).
DET-05	Transfers for more than available balance should not be allowed (transfer).
DET-06	Transfers for more than available balance should not be allowed (transferFrom).
DET-07	Zero amount transfers should not break accounting (transfer).
DET-08	Zero amount transfers should not break accounting (transferFrom).
DET-09	Transfers should update accounting correctly.
DET-10	TransferFrom should decrease allowance.
DET-11	Transfer for more than available allowance should not be allowed.
DET-12	User shares must not exceed total totalShares.
DET-13	Sum of users shares must not exceed total totalShares.
DET-14	divETH and totalEther should be related 1:1
DET-15	Self shares transfers should not break accounting (transferShares).
DET-16	Self shares transfers should not break accounting (transferSharesFrom).
DET-17	Transfers for more than available balance should not be allowed (transferShares).
DET-18	Transfers for more than available balance should not be allowed (transferSharesFrom).
DET-19	DET-19: Zero amount transfers should not break accounting (transferShares).
DET-20	Zero amount transfers should not break accounting (transferSharesFrom).
DET-21	Shares transfers should update accounting correctly.
DET-22	transferSharesFrom should decrease allowance.
DET-23	Transfer for more than available allowance should not be allowed (transferSharesFrom).
DET-24	Deposit should increase user shares.

DET-25	Deposit should increase ETH balance.
DET-26	Deposit should increase totalEther.
DET-27	Deposit should increase totalShares.
DET-28	Deposit should keep totalEther/totalShares relation constant.
DET-29	Zero value deposit should revert with `ErrZeroDeposit`.
DET-30	BurnShares should decrease user shares.
DET-31	BurnShares should decrease totalShares.
DET-32	BurnShares should decrease totalEther.
DET-33	BurnShares should keep totalEther/totalShares relation constant.

Contracte WrappedDivaEtherToken	
WDE-01	total supply should be equal to contract's divETH.
WDE-02	wrap should revert with `ErrZeroAmount` when zero amount.
WDE-03	wrap should increase contract's divETH balance.
WDE-04	wrap should mint wdivETH to user.
WDE-05	unwrap should revert with `ErrZeroAmount` when zero amount.
WDE-06	unwrap should increase users's divETH balance.
WDE-07	unwrap should burn wdivETH from user.
WDE-08	depositETH should revert with `ErrZeroAmount` when zero value.
WDE-09	depositETH should increase contract's divETH balance
WDE-10	depositETH should mint wdivETH to user.

Contracte WithdrawQueueEscrow	
WQE-01	requestWithdrawal should revert with `InvalidWithdrawalAmount` if ammount is not within the limits.
WQE-02	requestWithdrawal should increase WithdrawalQueueEscrow's DivaEtherToken balance.
WQE-03	requestWithdrawal should increase divaethSharesToBurn.
WQE-04	requestWithdrawal should create a new WithdrawalRequest.
WQE-05	claim should revert with `InvalidWithdrawalRequestId`given with withdrawalRequestId is invalid
WQE-06	claim should revert with `WithdrawalRequestAlreadyClaimed` if withdrawal request is already claimed.
WQE-07	claim should revert with `WithdrawalRequestNotAllocated` if withdrawal request has not been alocated.
WQE-08	claim should return corresponding ETH - fee to owner.

WQE-09	claim should reduce assignedWithdrawalEther.
WQE-10	claim should increase withdrawal request sharesClaimed.
WQE-11	`assignedWithdrawalEther` should be always greater than or equal to the contract balance.
WQE-12	The sum of the shares of all withdrawal requests should be equal to the total shares to burn.
WQE-13	No withdrawal request should have more shares withdrawn than requested.
WQE-14	availableEther should report WithdrawQueueEscrow balance - ETH allocated in bundles.
WQE-15	isWithdrawalRequestClaimable should return correct request data.
WQE-16	isWithdrawalRequestClaimable should return false if no valid requestId is given.
WQE-17	Request bundles with zero shares should not be created.

4.5. Vulnerabilitats detectables

A continuació es presenten una serie de vulnerabilitats interessants les quals son fàcilment detectades pel *setup* desenvolupat.

V1 - La conversió incorrecta dels actius a DivaEtherToken fa trenca la comptabilitat.

Aquesta vulnerabilitat s'havia trobat prèviament amb anàlisis manuals per l'auditor 0xJuancito, però s'ha introduït a posteriori per a testejar l'efectivitat de l *setup* de *fuzzing*. Aquesta vulnerabilitat radica en una funció de conversió que és usada per a relacionar el token divETH al llarg de tot el protocol amb les *shares*. Les *shares* representen la propietat d'una fracció de tot l'ETH del protocol. És un clar exemple d'un cas que fàcilment pot escapar al testeig unitari, ja que mentre el token no rebasegi, la funció retornarà el valor correcte, perquè les variables totalEther i totalShares tindran el mateix valor. Com es pot observar a la Figura 18, aquestes estan invertides en calcular el quocient del factor de conversió.

```
/// @notice Converts an amount of divETH into the corresponding amount of shares.
/// @param assets The amount of divETH to convert.
/// @return The equivalent amount of shares.
ftrace | funcSig
function convertToShares(uint256 assets) public view returns (uint256) {
    /// @dev Assumption: totalEther can not be 0 because we lock shares/ether on address(1)
    return assets.mulDiv(totalEther, totalShares);
}
```

Figura 18 Fragment de codi vulnerable V1.

A continuació es mostra la traça d'execució que el *fuzzer* mostra quan detecta la vulnerabilitat.

- CryticTester.rebaseDivaEtherToken(300)
- CryticTester.crytic_DET_14()

Com es pot observar a la taula de propietats, la propietat DET-14 comprova que el token divETH està relacionat 1 a 1 amb l'ETH total del protocol i utilitza aquesta funció per a realitzar la comprovació. Si aquesta propietat es testa després de realitzar un *rebasig*, el test fallarà.

V2 - Denegació de servei en la cola de retirades.

Aquesta vulnerabilitat ha estat trobada únicament pel *setup* de *fuzzing.*

Per retirar ETH del protocol, els usuaris han de seguir un procés de tres passos.

1. Crear una petició de retirada especificant la quantitat que volen retirar cridant la funció **requestWithdrawal**.
2. Esperar un màxim de 24 hores a que el protocol executi les peticions de retirada assignant ETH a cada una d'elles en paquets.
3. Retirar l'ETH assignat a la petició mitjançant la funció **claim**.

Aquesta vulnerabilitat radica l'existència de la possibilitat de crear paquets de cues de retirada amb zero *shares* assignades. Un paquet amb zero *shares* no constitueix un problema en si mateix, però en el moment de cridar la funció **claim**, el valor total de *shares* del paquet s'utilitza com a factor divisor, resulta en un error de divisió per zero i bloqueja completament la cua de retirades.

La figura 19 mostra el fragment de codi vulnerable. Com es pot observar, s'hi realitzen comprovacions per evitar la creació de paquets amb zero *shares*. No obstant, existeix la possibilitat que les comprovacions passin i, a causa d'una pèrdua de precisió en la conversió del valor *_availableEther* a *shares*, es creï el paquet amb zero *shares*.

La figura 20 mostra mateix fragment de codi amb la vulnerabilitat mitigada.

```
function _fulfillWithdrawalRequests() internal {
  // shares that are pending processing
  uint256 sharesToBeProcessed = divaethSharesToBurn - totalSharesProcessed;

  if(sharesToBeProcessed == 0) {
    return;
  }

  // amount of available ether to be used.
  uint256 _availableEther = availableEther();

  // @dev if contract has no ether this cant fullfill any request
  if(_availableEther == 0) {
    return;
  }

  // amount of shares that can be processed
  uint256 sharesProcessed = FixedPointMathLib.min(sharesToBeProcessed, DIVAETH.convertToShares(_availableEther));

  ...

}
```

Figura 19 Fragment de codi vulnerable V2.

```
function _fulfillWithdrawalRequests() internal {
  // shares that are pending processing
  uint256 sharesToBeProcessed = divaethSharesToBurn - totalSharesProcessed;

  // amount of available ether to be used.
  uint256 _availableEther = availableEther();

  // amount of shares that can be processed
  uint256 sharesProcessed = FixedPointMathLib.min(sharesToBeProcessed, DIVAETH.convertToShares(_availableEther));

  if(sharesProcessed == 0) {
    return;
  }

  ...

}
```

Figura 20 Fragment de codi no vulnerable V2.

A continuació es mostra la traça d'execució que el *fuzzer* mostra quan detecta la vulnerabilitat:

- CryticTester.rebaseDivaEtherToken(115792089237316195423570985008687907853269984665640564039424384007913129639936)
- CryticTester.crytic_WQE_07()
- CryticTester.fulfillAllWithdrawalRequests(3593795444981880677329949876123150460995344030590954811901228553938954469 4)
- CryticTester.claim(3600, 15199999999999999940)
- CryticTester.fulfillSomeWithdrawalRequests(0, 7200)

V3 - Retirada exitosa incorrecta.

Aquesta vulnerabilitat ha estat trobada únicament pel *setup* de *fuzzing.*

Aquesta vulnerabilitat radica en la possibilitat d'executar una retirada sense esperar el període de fins a 24 hores estipulat pel protocol. Encara que la retirada es pugui realitzar, aquesta retira zero ETH del protocol, no obstant això, és un comportament incorrecte.

Com es pot observar a la Figura 21, la funció **claim** comprova que la petició de retirada hagi estat executada, comprovant el total de *shares* que s'han assignat per a retirada. El fet que s'utilitzi l'operador < en lloc del <= permet que retirades no executades (zero shares assignades) puguin cridar la funció claim de manera exitosa sense haver d'esperar a ser executades. Aquestes retiraran 0 ETH.

```
function claim(uint256 _withdrawalRequestId) external whenNotPaused {

  ...

  if(
    withdrawBundles.length == 0 ||
    withdrawBundles[withdrawBundles.length - 1].totalShares < withdrawalRequestSharesStartAt
  ){
    revert WithdrawalRequestNotAllocated();
  }

  ...

}
```

Figura 21 Fragment de codi vulnerable V3.

```
function claim(uint256 _withdrawalRequestId) external whenNotPaused {

  ...

  if(
    withdrawBundles.length == 0 ||
    withdrawBundles[withdrawBundles.length - 1].totalShares <= withdrawalRequestSharesStartAt
  ){
    revert WithdrawalRequestNotAllocated();
  }

  ...

}
```

Figura 22 Fragment de codi no vulnerable V3.

A continuació es mostra la traça d'execució que el *fuzzer* mostra quan detecta la vulnerabilitat:

- CryticTester.fulfillAllWithdrawalRequests(101804427064202130427270638872314578094594721536484563280439044797607590301875)
- CryticTester.crytic_WQE_07()

5 Conclusions i treballs futurs

5.1. Síntesis de treball realitzat

Durant la realització d'aquest treball s'ha estudiat l'ecosistema d'eines i mètodes disponibles en l'àmbit del *fuzzing* de contractes intel·ligents d'Ethereum. Amb una millor perspectiva i coneixement del camp, es poden extreure les següents conclusions:

- Malgrat la poca maduresa de la tecnologia Ethereum, la seguretat ha estat un factor a tenir en compte des de les etapes inicials de l'ecosistema. Això ha fet que ja existeixin diverses eines al mercat que permeten la construcció de *setups* de *fuzzing* de manera professional i fiable per a contractes intel·ligents Solidity. No obstant això, la tecnologia és bastant nova i hi ha pocs experts que dominin les eines, fet que provoca una manca de documentació en el sector i l'absència d'estàndards establerts per a la construcció de *setups*. El *setup* construït en aquest treball intenta sintetitzar els diferents mètodes observats per a la construcció de *setups* efectius.
- Encara que les estratègies i nomenclatures siguin similars entre el *fuzzing* de programes convencionals i el *fuzzing* de contractes intel·ligents, propietats com l'estat perpetu o l'ABI fan que en el *fuzzing* de contractes tingui molt més sentit utilitzar tècniques específiques (*fuzzing* amb estat, generació d'inputs a partir de l'ABI) que són molt útils en aquest cas d'ús.
- El *fuzzing* és un mètode molt efectiu que hauria d'incloure's en el cicle de vida de qualsevol producte de qualitat basat en contractes intel·ligents.
- Els mètodes descrits en aquest treball permeten abstraure la descripció de les propietats globals d'un protocol i verificar-les utilitzant *fuzzing* de manera efectiva.
- Encara que el *fuzzing* estigui catalogat com una eina automàtica en la seva execució, sempre necessitarà una etapa inicial que inclogui la intervenció humana per a la construcció del setup.
- La ciberseguretat en contractes intel·ligents mai és suficient, i el *fuzzing* no suposa un reemplaçament d'altres tècniques com les revisions manuals o el testeig unitari, sinó un complement per assolir la màxima seguretat.

5.2. Avaluació dels objectius plantejat inicialment

Durant tot el treball s'ha seguit el pla proposat inicialment. No obstant això, algunes parts, com l'extensió del setup de fuzzing al protocol complet o la

integració de l'execució del setup en el cicle de vida del producte, s'han hagut d'eliminar de l'abast, ja que suposarien una inversió de temps massa gran per a l'extensió d'aquest treball.

5.3. Treballs futurs

El fuzzing de contractes intel·ligents és una tecnologia molt recent amb diverses àrees d'estudi. A continuació es proposen una sèrie de treballs futurs compatibles amb la finalització d'aquest projecte:

- Extendre el setup de fuzzing de Diva Staking a la totalitat del protocol.
- Estudiar eines de fuzzing de contractes intel·ligents en altres ecosistemes que utilitzen diferents llenguatges, com Solana, StarkNet entre altres.
- Explorar tècniques de testeig híbrides que combinin, a més del fuzzing, també l'execució simbòlica i la verificació formal.

6 Glossari

Ad Hoc: solucions o processos específics dissenyats per a una tasca o problema particular, sense considerar una aplicació més àmplia.

Bug Bounty: Un programa que recompensa els individus per trobar i reportar errors o vulnerabilitats en el programari.

CI (Continuous Integration): Pràctica que consisteix en incorporar canvis de codi o tests en un repositori compartit do codi de manera automàtica i periòdica.

Code Coverage: Mesura que indica el percentatge de codi font que ha estat cobert pel test.

Commit: En el control de versions, com Git, un commit és una instantània dels canvis en el codi font.

DeFi (Descentralized Finance): Un ecosistema financer construït sobre tecnologies blockchain que permet la prestació de serveis financers, com ara préstecs, assegurances i comerç, sense la necessitat d'intermediaris centrals.

Edge Cases: Condicions extremes o poc comunes d'ús en les quals un programa podria no funcionar correctament.

EVM (Ethereum Virtual Machine): L'entorn d'execució per a contractes intel·ligents en la xarxa Ethereum.

EVM Compatible: Fa referència a les blockchains que són capaces d'executar contractes intel·ligents utilitzant la mateixa màquina virtual que Ethereum, permetent la interoperabilitat de contractes i aplicacions.

Fee: Comissió que s'ha de pagar per la realització d'un servei específic.

Framework: Estructura de programari que proporciona una base reutilitzable per desenvolupar aplicacions, incloent eines, biblioteques i guies per facilitar el desenvolupament i manteniment del codi.

Fuzzing: Una tècnica de prova que automàticament injecta dades invàlides, inesperades o aleatòries en el sistema per trobar errors i vulnerabilitats.

Go: lenguatge de programació de codi obert creat per Google, dissenyat per ser eficient, simple i segur, especialment adequat per a sistemes distribuïts i aplicacions de xarxa.

Happy Paths: Seqüències de passos en proves de programari que representen els escenaris d'ús més comuns i sense errors, garantint que el sistema funciona correctament en condicions ideals.

Harness: Component de programari utilitzat en fuzzing per gestionar, controlar i monitoritzar l'entrada i execució de proves automatitzades, facilitant la detecció d'errors i vulnerabilitats.

Immunefi: Una plataforma de bug bounty enfocada a la seguretat de DeFi i contractes intel·ligents, que connecta projectes amb investigadors de seguretat per identificar i corregeir vulnerabilitats.

Open Source: Programari del qual el codi font és accessible públicament per a qualsevol persona per veure, modificar i distribuir.

Pool: Agrupació de liquiditat aportats per diversos usuaris.

Rekt: Argot utilitzat dins de la comunitat cripto per referir-se a una pèrdua significativa de fons, sovint a causa de fallades de seguretat o mala gestió d'inversions.

Rust: Llenguatge de programació segur i eficient, dissenyat per evitar errors de memòria i aconseguir alt rendiment.

Secuirty Review: Un examen detallat del codi i l'arquitectura d'un sistema per identificar vulnerabilitats de seguretat i millorar les defenses contra atacs.

Seed: En el context del fuzzing, es refereix al conjunt de dades d'entrada vàlides inicials utilitzades per a iniciar la campanya de fuzzing.

Setup: En el context de la informàtica es refereix a la configuració o l'equipament necessari per a una activitat específica.

Shares: Representacions de propietat.

Solidity: Un llenguatge de programació orientat a contractes per a la creació de contractes intel·ligents que s'executen en l'Ethereum Virtual Machine (EVM).

Stack: Regió de la memòria d'un programa que s'utilitza per emmagatzemar variables locals de funcions, els seus arguments i adreces de retorn.

Storage: Emmagatzematge persistent de dades.

SUT: Sistema o component específic que s'està provant durant el procés de proves de programari per verificar-ne el comportament i funcionalitat.

Testeig Dinàmic: El testeig dinàmic és un procés d'avaluació de programari que s'efectua mentre el programa està en execució, a diferència del testeig estàtic, que es realitza analitzant el codi sense executar-lo.

The Merge: Transició d'Ethereum del mecanisme de consens de Proof of Work (PoW) a Proof of Stake (PoS) per millorar l'eficiència energètica i la seguretat de la xarxa.

Timestamp: Marca temporal en contractes intel·ligents que registra l'hora exacta d'execució d'una transacció.

Trustless: Un sistema o procés en el qual les parts no necessiten confiar entre elles perquè el sistema funcioni de manera segura.

TVL (Total Value Locked): Una mètrica que mesura la quantitat total de valor bloquejat dins dels protocols DeFi. És un indicador de la salut i l'activitat d'un ecosistema DeFi.

Vyper: Un llenguatge de programació per a contractes intel·ligents que prioritza la simplicitat i la seguretat.

White Hat: Hacker ètic que utilitza les seves habilitats per trobar i corregir vulnerabilitats en sistemes amb el propòsit de millorar-ne la seguretat.

7 Bibliografia

[1] Nakamoto, Satoshi. «Bitcoin: A Peer-to-Peer Electronic Cash System». www.bitcoin.org [en línia]. 2008 [consultat 9 març 2024]. Disponible a: https://bitcoin.org/bitcoin.pdf

[2] M. Antonopoulos, Andreas. Dr. Wood, Gavin. «What Is Ethereum?». A: *Mastering Ethereum* [en línia]. 1ª ed. Estats Units d'Amèrica;O'Reilly; 2018 des. [consultat 9 març 2024]. 384 pags. Disponible a: https://github.com/ethereumbook/ethereumbook

[3] Jiang, Erya. Quin Bo. Wang Quin. Wang Zhipeng. Wu Qianhong. Weng Jian. «Decentralized Finance (DeFi): A Survey». ArXiv [en línia] 2023 nov. [consultat 9 març 2024]; arXiv:2308.05282v2 [cs.CR] Disponible a: https://arxiv.org/pdf/2308.05282v2.pdf

[4] Chainalysis. «The 2024 Crypto Crime Report». Estats Units d'Amèrica; 2024 febr. 112 pags. Disponible a: https://go.chainalysis.com/rs/503-FAP-074/images/The%202024%20Crypto%20Crime%20Report.pdf

[5] Halborn. «Breaking Down The Top 50 DeFi Hacks 2016-2022». Estats Units d'Amèrica; 33 pags. Disponible a: https://www.halborn.com/reports/top-hacks/halborn_top_50_defi_hacks_2016_2022.pdf

[6] EU Blockchain Observatory and Forum. «Ethereum Merge Trend Report» Comunitat Europea; 2023 abr. 17 pags Disponible a: https://www.eublockchainforum.eu/sites/default/files/reports/EUBOF3.0_Ethereum_Merge_Trend_Report_final.pdf

[7] Miller, Barton P. Fredriksen, Lars. So, Bryan. «An empirical study of the reliability of UNIX utilities». Communications of the ACM 33 (1990): pag 32 - pag 44.

[8] Böhme, Marcel. Pham, Van-Thuan. Roychoudhury, Abhik. «Coverage-based Greybox Fuzzing as Markov Chain». Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (2016): pag 1032 – pag 1043.

[9] Runerson, Per. «A Survey of Unit Testing Practices». IEEE Software (2006 jul. Vol 23, No. 4): pag 22-29.

[10] Nomic Foundation. Hardhat [programari]. Versió 2.22.4

[11] Paradigm. **Foundry** [programari]. Versió 0.2.0

[12] OpenZeppelin [en línia] [consultat 15 maig 2024]. Disponible a: https://www.openzeppelin.com/

[13] TrailOfBits [en línia] [consultat 15 maig 2024]. Disponible a: https://www.trailofbits.com/

[14] SpearBit [en línia] [consultat 15 maig 2024]. Disponible a: https://spearbit.com/

[15] Code4rena [en línia] [consultat 15 maig 2024]. Disponible a: https://code4rena.com/

[16] Cantina [en línia] [consultat 15 maig 2024]. Disponible a: https://cantina.xyz/

[17] Sherlock [en línia] [consultat 15 maig 2024]. Disponible a: https://www.sherlock.xyz/

[18] Immunefi [en línia] [consultat 15 maig 2024]. Disponible a: https://immunefi.com/

[19] OpenZeppelin. Code Inspector [programari SaaS].

[20] TrailOfBits. Slither [programari]. Versió 2.0.0

[21] Cyfrin. Aderyn [programari]. Versió 0.0.26

[22] TrailOfBits. Echidna [programari]. Versió 2.2.3

[23] TrailOfBits. Medusa [programari]. Versió 0.1.3

[24] Certora. Certora Prover [programari SaaS].

[25] a16zcrypto. Halmos [programari]. Versió 0.1.13

[26] Lessig, Lawrence. «Code is law». A: *Code and other laws of cyberspace* [en línia]. 1ª ed. Estats Units d'Amèrica;Basic Books ; 1999. 297 pags.

[27] Etherscan. Etherscan [programari SaaS].

[28] Vogelsteller Fabian. Buterin, Vitalik «ERC-20: Token Standard». Ethereum Improvement Proposals [en línia]. 2015 [consultat 15 maig 2024]. Disponible a: https://eips.ethereum.org/EIPS/eip-20

[29] Diva Staking [en línia] [consultat 22 maig 2024]. Disponible a: https://divastaking.com/

[30] Ethereum Staking [en línia] [consultat 11 juny 2024]. Disponible a: https://ethereum.org/en/staking/

[31] Edgington, Ben. «Slashing». A: *Upgrading Ethereum* [en línia]. 0.3ª ed. 2023 ag. [consultat 11 juny 2024]. 367 pags. Disponible a: https://eth2book.info/

[32] Boneh, Dan. Lynn, Ben. Shacham, Hovav. «Short Signatures from the Weil Pairing». Journal of Cryptology (2004): vol 17 pag 297 – pag 319.