ĐẠI HỌC QUỐC GIA TP. HỒ CHÍ MINH

TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN

KHOA MẠNG MÁY TÍNH VÀ TRUYỀN THÔNG

BÁO CÁO ĐỒ ÁN LẬP TRÌNH AN TOÀN & KHAI THÁC LỖ HỔNG PHẦN MỀM

-&-

ĐỀ TÀI:
CROSSFUZZ - CROSS-CONTRACT FUZZING FOR SMART CONTRACT VULNERABILITY DETECTION.

Giảng viên hướng dẫn: **Phan Thế Duy**

Thực hiện bởi Nhóm 2, gồm:

• **LẠI QUAN THIÊN**	**22521385**	**Trưởng nhóm**
• **MAI NGUYỄN NAM PHƯƠNG**	**22521164**	**Thành viên**
• **HỒ DIỆP HUY**	**22520541**	**Thành viên**
• **ĐẶNG ĐỨC TÀI**	**22521270**	**Thành viên**

Lớp: **NT521.P12.ANTT**

TP. Hồ Chí Minh, tháng 12 năm 2024

LỜI MỞ ĐẦU

Trong bối cảnh công nghệ blockchain không ngừng phát triển, Ethereum đã nổi lên như một nền tảng tiên phong, thúc đẩy sự bùng nổ của các ứng dụng phi tập trung (**dApps**) và tài chính phi tập trung (**DeFi**). Tuy nhiên, song song với sự tăng trưởng này là những thách thức về bảo mật, đặc biệt là các lỗ hổng trong **hợp đồng thông minh** (smart contracts). Những lỗ hổng này không chỉ gây thiệt hại lớn về tài chính mà còn đe dọa đến uy tín và niềm tin của người dùng đối với các ứng dụng blockchain.

Với mục tiêu cải thiện tính an toàn và độ tin cậy của các hợp đồng thông minh, công nghệ **fuzzing** đã trở thành một giải pháp tiềm năng, đặc biệt là trong việc phát hiện các lỗi bảo mật khó nhận biết. Trong đó, **CrossFuzz**, một công cụ kiểm thử tự động, được thiết kế để phát hiện lỗ hổng bằng cách phân tích luồng giao dịch phức tạp giữa các hợp đồng thông minh liên kết.

Tuy nhiên, việc triển khai và áp dụng CrossFuzz trên nền tảng Ethereum đòi hỏi người nghiên cứu phải am hiểu sâu về cơ chế hoạt động của blockchain, cấu trúc hợp đồng thông minh và các kỹ thuật kiểm thử bảo mật hiện đại. Nhằm khai thác tối đa tiềm năng của công cụ này, chúng em đã lựa chọn đề tài **"CrossFuzz: Cross-contract fuzzing for smart contract vulnerability detection"** để đi sâu vào các khía cạnh lý thuyết và thực tiễn.

Đề tài tập trung vào việc tìm hiểu các khái niệm nền tảng về blockchain và Ethereum, cơ chế hoạt động của CrossFuzz, đồng thời triển khai các thí nghiệm minh họa, bao gồm kiểm thử tự động và phân tích kết quả để phát hiện lỗ hổng. Qua đó, chúng em hy vọng đề tài này sẽ không chỉ góp phần hỗ trợ cộng đồng blockchain nâng cao khả năng bảo mật mà còn trang bị thêm kiến thức và kỹ năng thực tiễn cho các nhà phát triển và nghiên cứu trong lĩnh vực công nghệ này.

Chúng em tin rằng, trong thời đại số hóa hiện nay, việc đảm bảo an toàn cho các ứng dụng blockchain là yếu tố then chốt để thúc đẩy sự phát triển bền vững của công nghệ này. Hy vọng rằng, kết quả nghiên cứu từ đề tài sẽ mang lại giá trị thiết thực và tạo nền tảng cho các nghiên cứu chuyên sâu hơn trong tương lai.

LỜI CẢM ƠN

Trước tiên, chúng em xin gửi lời cảm ơn sâu sắc đến thầy Phan Thế Duy - giảng viên hướng dẫn môn Lập trình an toàn & Khai thác lỗ hổng phần mềm, đã tận tình chỉ dẫn và chia sẻ những kinh nghiệm quý báu trong suốt quá trình thực hiện đề tài. Những ý kiến đóng góp và sự hướng dẫn của thầy đã giúp chúng em rất nhiều trong việc định hướng và hoàn thiện nội dung đồ án.

Chúng em cũng xin gửi lời cảm ơn chân thành đến các thành viên trong nhóm, những người đã không ngừng nỗ lực, phối hợp và đóng góp ý kiến để hoàn thành đề tài một cách tốt nhất. Dù gặp nhiều khó khăn và giới hạn về thời gian, nhưng nhờ sự đoàn kết và tinh thần trách nhiệm, chúng em đã vượt qua và đạt được mục tiêu đề ra.

Cuối cùng, chúng em xin bày tỏ lòng biết ơn đến những người đã hỗ trợ và động viên chúng em trong suốt quá trình thực hiện đề tài. Do giới hạn về kiến thức và thời gian, không thể tránh khỏi những thiếu sót trong đồ án, chúng em rất mong nhận được sự góp ý từ thầy và các bạn để hoàn thiện hơn.

TP. Hồ Chí Minh, tháng 12 năm 2024

Nhóm 2, lớp NT521.P12.ANTT

MỤC LỤC

CHƯƠNG 1. TỔNG QUAN ĐỀ TÀI

1.1. Xu hướng

Trong những năm gần đây, công nghệ blockchain đã trở thành một trong những công nghệ cốt lõi của thời đại số hóa, với các ứng dụng không chỉ giới hạn trong tài chính mà còn mở rộng sang các lĩnh vực như chăm sóc sức khỏe, điện toán đám mây và Internet of Things (IoT). Ethereum, một trong những nền tảng blockchain phổ biến nhất, hỗ trợ triển khai các ứng dụng phi tập trung (dApps) và hợp đồng thông minh.

Tuy nhiên, sự gia tăng trong việc sử dụng hợp đồng thông minh cũng kéo theo những mối lo ngại về bảo mật. Các lỗ hổng trong hợp đồng thông minh đã dẫn đến những sự cố nghiêm trọng, điển hình là vụ tấn công DAO năm 2016, nơi 60 triệu Ether bị đánh cắp do lỗ hổng reentrancy. Theo báo cáo từ SlowMist, tính đến tháng 7 năm 2023, đã có hơn 1109 sự cố bảo mật xảy ra, gây tổn thất trên 30 tỷ USD. Điều này nhấn mạnh tầm quan trọng của việc phát triển các phương pháp phát hiện lỗ hổng nhằm đảm bảo an toàn cho hợp đồng thông minh.

Một trong những phương pháp kiểm thử hiệu quả nhất hiện nay là fuzz testing, đặc biệt khi ứng dụng vào việc phát hiện các lỗ hổng giao dịch phức tạp giữa các hợp đồng liên kết (cross-contract vulnerabilities). Phương pháp này cho phép kiểm thử trên quy mô lớn, giúp giảm thiểu các lỗi logic và tăng cường bảo mật.

1.2. Phạm vi đề tài

1.2.1. Etherum là gì? Blockchain là gì?

Blockchain là một **cơ sở dữ liệu phân tán** hoạt động như một sổ cái kỹ thuật số, ghi lại các giao dịch một cách an toàn, minh bạch và bất biến. Các đặc điểm chính của blockchain bao gồm:

- Phi tập trung: Không có cơ quan trung ương nào kiểm soát blockchain. Thay vào đó, các bản sao dữ liệu được lưu trữ trên nhiều nút (nodes) trong mạng.

- Bất biến: Một khi dữ liệu được ghi vào blockchain, nó không thể bị sửa đổi hoặc xóa mà không có sự đồng thuận của toàn bộ mạng.

- Minh bạch: Tất cả các giao dịch trên blockchain đều có thể được kiểm chứng công khai.

Blockchain là nền tảng cốt lõi cho nhiều ứng dụng, trong đó nổi bật nhất là tiền mã hóa (cryptocurrency) như Bitcoin và Ethereum.

Ethereum là một nền tảng blockchain phi tập trung ra mắt năm 2015, nổi bật với khả năng thực thi các hợp đồng thông minh và hỗ trợ ứng dụng phi tập trung (dApps). Hai thành phần chính của Ethereum:

- **Ethereum Virtual Machine (EVM):** Môi trường thực thi các hợp đồng thông minh, đảm bảo tính nhất quán và bảo mật trên toàn mạng.

- **Ether (ETH):** Đồng tiền mã hóa chính của Ethereum, dùng để thanh toán phí giao dịch và thực thi hợp đồng.

1.2.2. Hợp đồng thông minh trong thời đại số

Hợp đồng thông minh là các đoạn mã máy tính chạy trên blockchain, tự động thực thi các điều khoản hợp đồng khi các điều kiện được đáp ứng. Hợp đồng thông minh có các đặc điểm:

- **Tự động:** Loại bỏ sự phụ thuộc vào bên trung gian.

- **Minh bạch:** Tất cả các bên tham gia đều có thể kiểm tra mã hợp đồng.

- **An toàn:** Dữ liệu trong hợp đồng không thể bị sửa đổi.

Vai trò và ứng dụng trong nhiều lĩnh vực:

- **Tài chính phi tập trung (DeFi):** Quản lý tài sản, cho vay, vay, và giao dịch phi tập trung.

- **Quản lý chuỗi cung ứng:** Theo dõi và đảm bảo tính minh bạch trong chuỗi cung ứng.

- **Sở hữu trí tuệ:** Quản lý bản quyền và phân phối tự động tiền bản quyền.

- **Bảo hiểm:** Tự động xử lý và chi trả bảo hiểm dựa trên các sự kiện được xác minh.

Tuy nhiên, hợp đông thông minh vẫn đối mặt với những thách thức như:

- **Bảo mật:** Các lỗi trong mã có thể dẫn đến thiệt hại lớn.

- **Khả năng mở rộng:** Ethereum và các blockchain khác cần cải thiện hiệu suất để đáp ứng nhu cầu ngày càng tăng..

- **Pháp lý:** Quy định pháp luật về hợp đồng thông minh vẫn chưa hoàn thiện

1.2.3. Kiểm thử bảo mật với CrossFuzz

Các phương pháp kiểm thử bảo mật truyền thống, như phân tích tĩnh hay thực thi biểu tượng, có hiệu quả trong việc phát hiện lỗ hổng đơn lẻ nhưng thường bỏ qua các tương tác phức tạp giữa các hợp đồng. Điều này dẫn đến tỷ lệ phát hiện sai cao và bỏ sót nhiều lỗ hổng nghiêm trọng.

CrossFuzz được phát triển để khắc phục những hạn chế này thông qua fuzz testing, một kỹ thuật kiểm thử dựa trên việc tạo dữ liệu ngẫu nhiên. CrossFuzz tập trung vào:

+ Khởi tạo hợp đồng chính xác: Giải quyết vấn đề giá trị mặc định trong constructor để tránh lỗi khi gọi các hàm liên hợp đồng.

+ Tối ưu hóa thứ tự giao dịch: Sử dụng phân tích luồng dữ liệu liên hợp đồng để xác định và ưu tiên các giao dịch quan trọng.

+ Hiệu quả vượt trội: Tăng độ bao phủ mã bytecode và phát hiện nhiều lỗ hổng hơn so với những công cụ kiểm thử khác.

CrossFuzz không chỉ nâng cao hiệu quả kiểm thử mà còn mở ra hướng nghiên cứu mới trong việc phát hiện các lỗ hổng bảo mật phức tạp, góp phần đảm bảo an toàn cho các hệ thống hợp đồng thông minh

CHƯƠNG 2. CHI TIẾT ĐỀ TÀI

2.1. Động lực nghiên cứu của CrossFuzz

CrossFuzz được phát triển nhằm giải quyết các hạn chế trong việc phát hiện lỗ hổng bảo mật trong các hợp đồng thông minh, đặc biệt là các lỗ hổng bảo mật liên quan đến tương tác giữa nhiều hợp đồng (cross-contract vulnerabilities). Các lý do chính thúc đẩy phát triển CrossFuzz sẽ được trình bày ở phần dưới đây.

2.1.1. Sự gia tăng của cross-contract vulnerabilities và các thiệt hại về kinh tế

Trong những năm gần đây, hợp đồng thông minh Ethereum đã được sử dụng rộng rãi trong nhiều lĩnh vực khác nhau, chẳng hạn như tài chính, chăm sóc sức khỏe, điện toán đám mây và Internet of Things. Tuy nhiên, các lỗ hổng bảo mật tiềm ẩn tồn tại trong hợp đồng thông minh đã dẫn đến nhiều sự cố bảo mật, gây thiệt hại về kinh tế và rủi ro bảo mật

Ví dụ cụ thể như từ năm 2020 đến 2022, theo báo cáo từ SlowMist thì số lượng lỗ hổng cross-contract đã gia tăng đáng kể, gây thiệt hại đến 81,2 triệu USD

Ngoài ra do hợp đồng thông minh không thể sửa chữa hoặc cập nhật sau khi triển khai, việc phát hiện và phòng ngừa các lỗ hổng bảo mật là nhu cầu cấp bách

2.1.2. Hạn chế của các phương pháp kiểm thử hiện tại

a. Các phương pháp phân tích tĩnh

Các công cụ như Clairvoyance và SmartDagger sử dụng phân tích tĩnh để xây dựng biểu đồ luồng điều khiển (CFG) và luồng dữ liệu (DFG). Tuy nhiên, chúng vẫn còn tồn đọng một số vấn đề như:

- Tỷ lệ false-positive (dương tính giả) cao
- Không thể tạo các trường hợp kiểm tra để xác minh lỗ hổng
- Bỏ qua tác động của các trình tự giao dịch lên trạng thái hợp đồng, dẫn đến khả năng phát hiện không đầy đủ

b. Các phương pháp Fuzzing hiện có

3 công cụ Fuzzing phổ biến trong lĩnh vực kiểm thử hợp đồng ở hiện tại là sFuzz, xFuzz và ConFuzziuss vẫn tồn đọng những hạn chế như sau:

- Không thiết lập tham số đúng cho constructor: Khi triển khai hợp đồng, các công cụ này thường sử dụng giá trị mặc định (e.g., 0 cho kiểu uint hoặc 0x0 cho kiểu address) thay vì các giá trị thích hợp. Điều này làm mất khả năng khởi tạo đúng các hợp đồng phụ thuộc, dẫn đến lỗi trong việc kiểm tra các giao dịch liên quan

- Bỏ sót các chức năng quan trọng: Các công cụ này sử dụng mô hình học máy để giảm số lượng trình tự giao dịch cần thực hiện, nhưng điều này có thể khiến các chức năng quan trọng chứa lỗ hổng bị bỏ qua

- Hiệu quả bao phủ mã thấp: ví dụ như trong bài báo thì khi kiểm tra một tập hợp 396 hợp đồng, xFuzz chỉ đạt được 69,34% bao phủ mã bytecode, trong khi một số lỗ hổng tiềm năng không được kiểm tra

2.2. Nền tảng lý thuyết

2.2.1. Hợp đồng thông minh Ethereum

Hợp đồng thông minh Ethereum là các chương trình máy tính chạy trên nền tảng chuỗi khối Ethereum, được viết bằng ngôn ngữ lập trình Solidity. Sau khi biên dịch thành mã byte bằng công cụ như **solc**, chúng được thực thi bởi Máy ảo Ethereum (EVM). Mỗi hợp đồng thông minh chứa các **biến trạng thái** và **hàm**, trong đó các hàm được thực thi thông qua các giao dịch. Giao dịch bao gồm thông tin như địa chỉ người gửi, chữ ký hàm, tham số và lượng ether gửi kèm

2.2.2. Kiểm thử Fuzzing và ConFuzzius

Kiểm thử fuzzing là phương pháp tạo các test case với dữ liệu ngẫu nhiên để kiểm tra chương trình. Các test case này được gửi đến chương trình cần kiểm tra, và dựa trên các điều kiện đường dẫn được thu thập, kiểm thử xác định liệu có lỗ hổng bảo mật hoặc hành vi bất thường nào bị kích hoạt. Sau đó, thuật toán di truyền sẽ tạo ra các test case mới để tiếp tục kiểm tra cho đến khi đạt điều kiện dừng (ví dụ: thời gian kiểm thử hoặc phạm vi mã đã bao phủ)

ConFuzzius là công cụ cải tiến fuzz testing bằng cách kết hợp với giải ràng buộc (constraint-solving) để giải quyết các nhánh điều kiện nghiêm ngặt khó kiểm tra. Các tính năng chính bao gồm:

- Xử lý phụ thuộc Read-After-Write (RAW) giữa các biến trạng thái, tối ưu hóa chiến lược chọn seed

- Kết hợp các giao dịch thỏa mãn điều kiện RAW để tạo test case con hiệu quả hơn

- Cải thiện khả năng bao phủ mã, đặc biệt ở các nhánh khó tiếp cận

ConFuzzius vẫn còn những hạn chế tồn động bên trong chính nó, là một trong những nền tảng thúc đẩy sự phát triển của CrossFuzz:

- Không hỗ trợ phát hiện lỗ hổng trong các tình huống hợp đồng giao tiếp chéo (cross-contract scenarios)

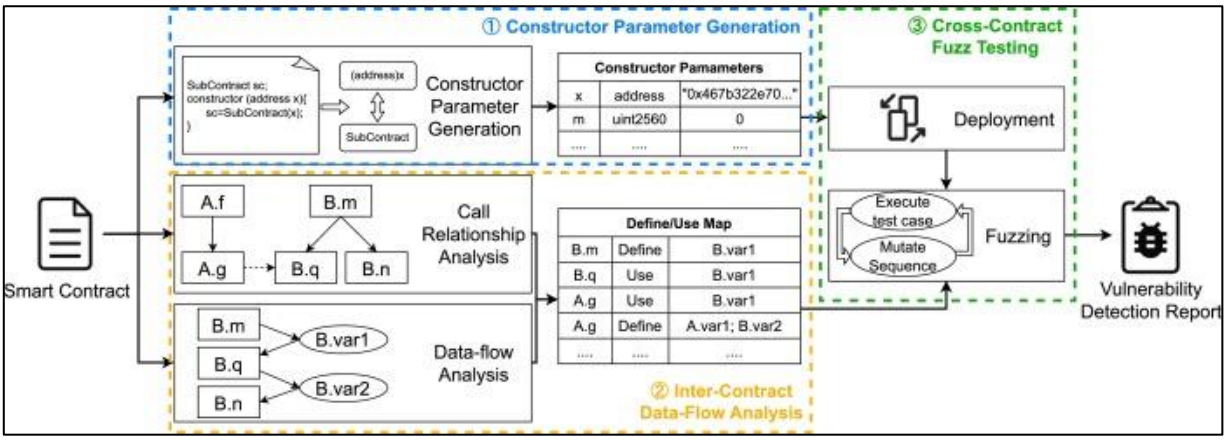
- Sử dụng giá trị mặc định cho các tham số của hàm khởi tạo khi triển khai hợp đồng, dẫn đến bỏ qua ảnh hưởng của các cuộc gọi giữa các hợp đồng đến luồng dữ liệu

2.3. Phương pháp tiếp cận của CrossFuzz

CrossFuzz là một phương pháp kiểm thử fuzzing để phát hiện lỗ hổng bảo mật giữa các hợp đồng thông minh, như thể hiện trong Hình 1.

Đầu tiên, CrossFuzz theo dõi các đường truyền dữ liệu của các tham số constructor, phân tích mối quan hệ giữa các tham số kiểu địa chỉ và các hợp đồng phụ thuộc, sau đó tạo các tham số cho constructor. Tiếp theo, CrossFuzz phân tích ICDF để hướng dẫn việc biến đổi các chuỗi giao dịch. Cuối cùng, CrossFuzz triển khai hợp đồng thông minh bằng các tham số constructor và thực hiện kiểm thử fuzzing.

Trong quá trình kiểm thử fuzzing, CrossFuzz thực thi các test case và giám sát việc thực thi các giao dịch, sau đó biến đổi chuỗi giao dịch dựa trên thông tin dòng dữ liệu. Khi thời gian kiểm thử đạt đến giới hạn đã định, CrossFuzz kết thúc và xuất báo cáo về các chuỗi giao dịch kích hoạt lỗ hổng. Các báo cáo này có thể giúp các nhà phát triển phân tích và xác định vị trí các lỗ hổng trong mã nguồn.



Hình 1: The framework of CrossFuzz

Trong phần sau đây, sẽ trình bày chi tiết ba phần quan trọng của CrossFuzz, tức là tạo tham số Constructor, phân tích luồng dữ liệu giữa các hợp đồng (ICDF) và Cross-contract fuzz testing.

2.3.1. Tạo tham số Constructor

Phần này sẽ mô tả quá trình tạo các tham số constructor, bao gồm việc theo dõi đường truyền dữ liệu của các tham số kiểu địa chỉ trong constructor. Các tham số constructor được tạo ra giúp constructor khởi tạo chính xác các hợp đồng phụ thuộc, điều này giúp vượt qua các hạn chế của các phương pháp fuzzing hiện tại.

Các hợp đồng thông minh tự động thực thi constructor của chúng trong quá trình triển khai, cho phép các nhà phát triển gán giá trị cho các biến trạng thái và khởi tạo trạng thái hợp đồng. Vì chỉ có người tạo hợp đồng mới có thể gọi constructor, họ thường thiết lập các thông tin quan trọng như chủ sở hữu hợp đồng, số tiền ban đầu, thời gian dịch vụ kết thúc, và địa chỉ của các hợp đồng phụ thuộc.

Kỹ thuật Phân tích Dòng Thông Tin (Information-Flow Analysis - IFA) đảm bảo an ninh thông tin bằng cách phân tích tính hợp pháp của việc truyền dữ liệu. Dựa trên IFA, quy trình phân tích “taint” coi dữ liệu đầu vào từ bên ngoài là nguồn "taint" và theo dõi các đường truyền dữ liệu của chúng để phân tích xem hệ thống phần mềm có an toàn hay không. Giống như các kỹ thuật phân tích "taint", CrossFuzz theo dõi các đường truyền dữ liệu của các tham số kiểu địa chỉ trong constructor, xác định xem tham số đó có được sử dụng để khởi tạo hợp đồng phụ thuộc hay không, và thiết lập một ánh xạ giữa các tham số kiểu địa chỉ và các hợp đồng phụ thuộc. Khi triển khai hợp đồng đang thử nghiệm, CrossFuzz điền các địa chỉ của các hợp đồng phụ thuộc vào các tham số constructor để khởi tạo chính xác các hợp đồng phụ

thuộc. Đối với các tham số kiểu địa chỉ không tìm thấy hợp đồng phụ thuộc tương ứng, CrossFuzz điền địa chỉ của người tạo hợp đồng vào. Các tham số constructor khác sẽ được điền các giá trị mặc định, giống như các phương pháp fuzzing truyền thống.

2.3.2. Phân tích dòng dữ liệu giữa các hợp đồng (ICDF)

Để tối ưu hóa chiến lược biến đổi chuỗi giao dịch của CrossFuzz, nhóm tác giả thực hiện phân tích dòng dữ liệu giữa các hợp đồng (Inter-contract Data Flow - ICDF) trước khi tiến hành kiểm thử fuzzing. Quá trình này không chỉ xem xét các thao tác định nghĩa và sử dụng các biến trạng thái trong các hợp đồng riêng lẻ, mà còn xem xét các mối quan hệ gọi hàm giữa các hợp đồng khác nhau.

Các hợp đồng thông minh thay đổi các biến trạng thái của hợp đồng thông qua các cuộc gọi hàm (tức là các giao dịch) để thay đổi trạng thái của hợp đồng. Nếu giá trị của các biến trạng thái bị thay đổi bởi các giao dịch, việc thực thi các giao dịch tiếp theo sẽ bị ảnh hưởng. Hơn nữa, số lượng kết hợp chuỗi giao dịch có thể tăng theo cấp số mũ do các cuộc gọi hàm giữa các hợp đồng, vì số lượng các hàm sẽ tăng lên.

Các phương pháp kiểm thử fuzzing cho hợp đồng thông minh thường tạo ra các chuỗi giao dịch dựa trên phân tích dòng dữ liệu, có nghĩa là các giao dịch chứa các thao tác thay đổi biến trạng thái sẽ được thực thi trước, sau đó các giao dịch chứa thao tác đọc biến trạng thái sẽ được thực thi sau. Dựa trên ConFuzzius, CrossFuzz phân tích ICDF để thu được thông tin về định nghĩa và cách sử dụng các biến trạng thái bởi các hàm. Thông tin này giúp tối ưu hóa chiến lược biến đổi chuỗi giao dịch, nâng cao hiệu quả phát hiện lỗ hổng.

2.3.3. Cross-contract fuzz testing

CrossFuzz thực hiện kiểm thử fuzz đa hợp đồng bằng cách tạo giao dịch gọi hàm từ hợp đồng cần kiểm thử và các hợp đồng phụ thuộc, rồi gửi chúng đến Máy ảo Ethereum (EVM) để phát hiện lỗ hổng bảo mật. Quá trình gồm bốn bước: triển khai hợp đồng, tạo trường hợp thử nghiệm, thực thi thử nghiệm, và đột biến thử nghiệm.

Đầu tiên, các hợp đồng phụ thuộc được triển khai trên Máy ảo Ethereum (EVM), và CrossFuzz lưu lại các địa chỉ của chúng để sử dụng trong quá trình thử nghiệm.. Tiếp theo, nó triển khai hợp đồng cần kiểm thử với các tham số khởi tạo được tạo ở phần 4.1.

Tiếp theo, CrossFuzz sử dụng ABI (Application Binary Interface - Giao diện nhị phân ứng dụng) của các hợp đồng thông minh để lấy thông tin về các hàm, kiểu tham số và kiểu

trả về. Từ thông tin này, CrossFuzz tạo ra các trường hợp thử nghiệm ban đầu với các chuỗi giao dịch chỉ chứa một lần gọi hàm cho mỗi hàm trong hợp đồng hoặc các hợp đồng phụ thuộc. Phương pháp này đảm bảo mỗi hàm được gọi ít nhất một lần, giúp kiểm thử toàn diện hơn so với các phương pháp khác như ConFuzzius.

Sau đó, CrossFuzz mã hóa các chuỗi giao dịch thành dữ liệu hệ thập lục phân rồi gửi đến EVM để thực thi. Trong quá trình thực thi, nó giám sát các thay đổi trong ngăn xếp, lượng gas tiêu thụ, môi trường khối, và các thông tin khác để phát hiện lỗ hổng bảo mật.

CrossFuzz sử dụng phương pháp phản hồi phạm vi mã nhằm tăng cường phạm vi kiểm thử cho các hợp đồng thông minh. Những nhánh mã chưa được khám phá sẽ được ưu tiên đột biến để sinh ra các trường hợp thử nghiệm mới. Dựa theo độ chi tiết, đột biến trường hợp thử nghiệm có thể được phân thành hai loại: cấp độ chuỗi và cấp độ giao dịch. Cấp độ chuỗi thay đổi thứ tự hoặc số lượng giao dịch, trong khi cấp độ giao dịch thay đổi tham số trong các giao dịch.

Các giao dịch bị kết thúc bởi lệnh REVERT được coi là các giao dịch ngoại lệ. Các giao dịch ngoại lệ này có thể làm giảm khả năng bao phủ mã và cản trở việc thực thi các giao dịch tiếp theo, từ đó làm giảm hiệu quả của kiểm thử.

Mục tiêu là giải quyết giao dịch ngoại lệ. Khi các giao dịch này được thực thi lại, có thể các điều kiện nhánh mới sẽ được đáp ứng, giúp khám phá thêm mã chưa được kiểm thử. Để giải quyết vấn đề của các giao dịch ngoại lệ, CrossFuzz tối ưu hóa chiến lược đột biến chuỗi giao dịch bằng cách tính điểm ưu tiên dựa trên ICDF. Các hàm có điểm ưu tiên cao nhất được thêm vào chuỗi giao dịch để bổ sung thông tin biến trạng thái cần thiết cho các giao dịch ngoại lệ ban đầu.

Lưu ý rằng CrossFuzz không phân biệt nguyên nhân của các ngoại lệ trong giao dịch. Do đó, CrossFuzz thực hiện đột biến chuỗi giao dịch cho bất kỳ giao dịch nào thực thi lệnh REVERT. Để tính điểm ưu tiên, tác giả gán điểm tích cực một cách heuristic nếu biến trạng thái được định nghĩa bởi hàm và điểm tiêu cực nếu biến trạng thái được sử dụng bởi hàm. Với một chuỗi giao dịch T và một giao dịch ngoại lệ exp, điểm ưu tiên của một hàm f gồm ba phần như thể hiện trong phương trình (1):

$$S(f, exp) = S_{Def}(f, exp) + S_{Provide}(f, exp) - S_{Use}(f, exp) \quad (1)$$

S_{Def}(f, exp) là số lượng biến trạng thái được định nghĩa bởi hàm f. Nhóm tác giả xem xét rằng việc thay đổi lặp đi lặp lại một biến trạng thái cụ thể có thể dẫn đến các chuỗi giao dịch quá dài, làm giảm hiệu quả của kiểm thử fuzz. Do đó, S_{Def}(f, exp) chỉ xem xét số lượng biến trạng thái chỉ được định nghĩa bởi hàm f thay vì các hàm khác trong chuỗi giao dịch. Như thể hiện trong phương trình (2), S_{Def}(f, exp) được tính bằng cách trừ tập hợp các biến trạng thái được định nghĩa bởi các giao dịch trước giao dịch ngoại lệ exp (ký hiệu là V_{PreDef}(exp)) khỏi tập hợp các biến trạng thái được định nghĩa bởi hàm f (ký hiệu là V_{Def}(f)):

$$S_{Def}(f, exp) = |V_{Def}(f) + V_{PreDef}(exp)| \quad (2)$$

Tương tự như S_{Def}(f, exp), $S_{Provide}$(f, exp) dùng để đếm số lượng biến trạng thái mà hàm f có thể cung cấp cho exp. Sự khác biệt là, so với S_{Def}(f, exp), $S_{Provide}$(f, exp) quan tâm nhiều hơn đến các biến trạng thái bổ sung mà hàm f cung cấp cho chuỗi giao dịch. Như thể hiện trong phương trình (3), $S_{Provide}$(f, exp) đầu tiên lấy giao của V_{Def}(f) và các biến trạng thái được sử dụng bởi exp (ký hiệu là VUse(exp)), sau đó trừ V_{PreDef}(exp) từ kết quả này:

$$S_{Provide}(f, exp) = |V_{Def}(f) \cap V_{Use}(exp) - V_{PreDef}(exp)| \quad (3)$$

S_{Use}(f, exp) được sử dụng để tính số lượng biến trạng thái chỉ được sử dụng bởi hàm f chứ không phải các biến trạng thái cũng được sử dụng bởi exp. Tác giải coi SUse(f, exp) là một tác động phụ, và giá trị này càng lớn, khả năng kích hoạt ngoại lệ khi thực thi hàm f càng cao. Như thể hiện trong phương trình (4), S_{Use}(f, exp) được tính bằng cách trừ V_{Use}(f) khỏi V_{Use}(exp):

$$S_{Use}(f, exp) = | V_{Use}(f) - V_{Use}(exp) | \quad (4)$$

Lưu ý rằng V_{Def}(f) và V_{Use}(f) có thể được lấy từ ICDF của hàm f, trong khi V_{Use}(exp) và V_{PreDef}(exp) có thể được lấy thông qua phân tích động của việc thực thi giao dịch. Cụ thể, CrossFuzz sử dụng giao diện do ConFuzzius cung cấp để phân tích các thay đổi trong ngăn xếp trong quá trình thực thi exp hoặc các giao dịch khác. Phân tích này cho phép CrossFuzz thu thập các thao tác dữ liệu (tức là định nghĩa hoặc sử dụng) của các biến trạng thái trong các giao dịch cụ thể, từ đó xác định V_{Use}(exp) và V_{PreDef}(exp).

CrossFuzz thực thi các chuỗi giao dịch đã được đột biến trên EVM, thu thập thông tin để kiểm tra xem có phát hiện lỗ hổng bảo mật nào không. Các chuỗi tiếp tục được đột biến và thử nghiệm cho đến khi kết thúc thời gian kiểm thử, cuối cùng xuất ra báo cáo về các lỗ hổng được phát hiện.

2.4. Thiết kế và đánh giá thí nghiệm

Phần này trình bày chi tiết về thiết kế và đánh giá hiệu quả của CrossFuzz, một phương pháp phát hiện lỗ hổng trong các hợp đồng thông minh (smart contract) trên Ethereum. CrossFuzz tập trung vào việc khai thác các lỗ hổng liên hợp đồng (cross-contract vulnerabilities) dựa trên fuzz testing.

Để đánh giá khả năng bao phủ mã và phát hiện lỗ hổng của CrossFuzz, tác giả tập trung vào làm rõ 3 Research Questions (câu hỏi nghiên cứu):

- RQ1: Liệu chiến lược đột biến chuỗi giao dịch có cải thiện khả năng bao phủ mã không? Nếu có, xác suất tối ưu để kích hoạt chiến lược này là bao nhiêu?

- RQ2: Liệu CrossFuzz có thể đạt được khả năng bao phủ mã cao hơn và phát hiện lỗ hổng tốt hơn so với các phương pháp fuzz testing hiện đại không?

- RQ3: Các tham số được tạo cho hàm khởi tạo có cải thiện khả năng bao phủ mã và phát hiện lỗ hổng không?

2.4.1. Thiết kế thí nghiệm

Để đánh giá hiệu quả của CrossFuzz, các nhà nghiên cứu đã xây dựng một quy trình thực nghiệm chi tiết, bao gồm các khía cạnh sau:

a. Tập dữ liệu:

Do các nghiên cứu trước đó về phát hiện lỗ hổng đa hợp đồng như Clairvoyance và SmartDagger chưa công khai tập dữ liệu của họ, và tập dữ liệu của xFuzz lại thiếu nhãn lỗ hổng, nhóm nghiên cứu đã tự xây dựng tập dữ liệu của riêng mình. Họ thu thập ngẫu nhiên 500 mã nguồn hợp đồng thông minh từ EtherScan, một nền tảng phân tích cho Ethereum. Sau đó, họ sử dụng công cụ phân tích tĩnh Slither để loại bỏ các hợp đồng không có lời gọi hàm ngoài (external function call), giữ lại 396 hợp đồng để tiến hành thử nghiệm. Các hợp đồng

được phân loại theo ba kích thước dựa trên số dòng mã: dưới 100 dòng (<100), từ 100 đến dưới 500 dòng (<500) và từ 500 dòng trở lên (>=500).

b. Chỉ số đánh giá:

Hai chỉ số đánh giá chính được sử dụng là: Độ bao phủ mã bytecode và số lượng lỗ hổng được phát hiện. Độ bao phủ mã bytecode tính toán dựa trên công thức: độ_bao_phủ_bytecode = số_lệnh_được_thực_thi / tổng_số_lệnh. Số lượng lỗ hổng được phát hiện chỉ được so sánh giữa CrossFuzz và ConFuzzius, hai công cụ có cùng quy tắc phát hiện. Cả hai công cụ đều có khả năng phát hiện 11 loại lỗ hổng bảo mật. Ngoài ra, độ lệch chuẩn cũng được tính toán và hiển thị trên biểu đồ để minh họa ý nghĩa thống kê.

c. Hệ thống cơ sở (Baselines):

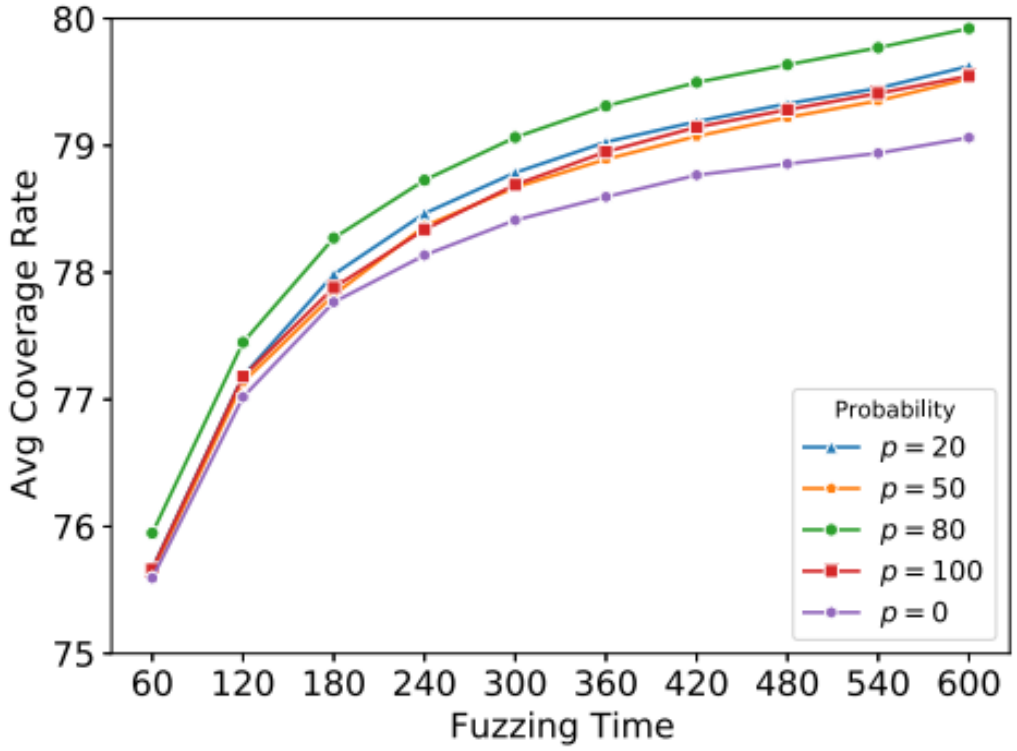
CrossFuzz được so sánh với ba fuzzer mã nguồn mở khác là: sFuzz, ConFuzzius, và xFuzz. sFuzz được phát triển dựa trên ContractFuzzer và cho kết quả tốt hơn về phát hiện lỗ hổng. ConFuzzius kết hợp kỹ thuật thực thi biểu tượng với fuzz testing, có khả năng bao phủ mã tốt. xFuzz được tối ưu hóa cho việc phát hiện lỗ hổng đa hợp đồng, cũng dựa trên sFuzz và ContractFuzzer.

d. Thiết lập tham số:

Tất cả các fuzzer đều sử dụng tham số mặc định. Phiên bản EVM được đặt là 'byzantium', ConFuzzius được kích hoạt mô-đun phân tích phụ thuộc dữ liệu và mô-đun giải ràng buộc. Thời gian kiểm thử tối đa cho mỗi hợp đồng thông minh được đặt là 10 phút, và mỗi hợp đồng được kiểm tra 5 lần, với kết quả cuối cùng là trung bình của 5 lần kiểm tra. Các thử nghiệm được thực hiện trên máy Ubuntu 20.04 LTS với bộ xử lý Intel Core i7-12700 và 32 GB RAM.

2.4.2. Đánh giá

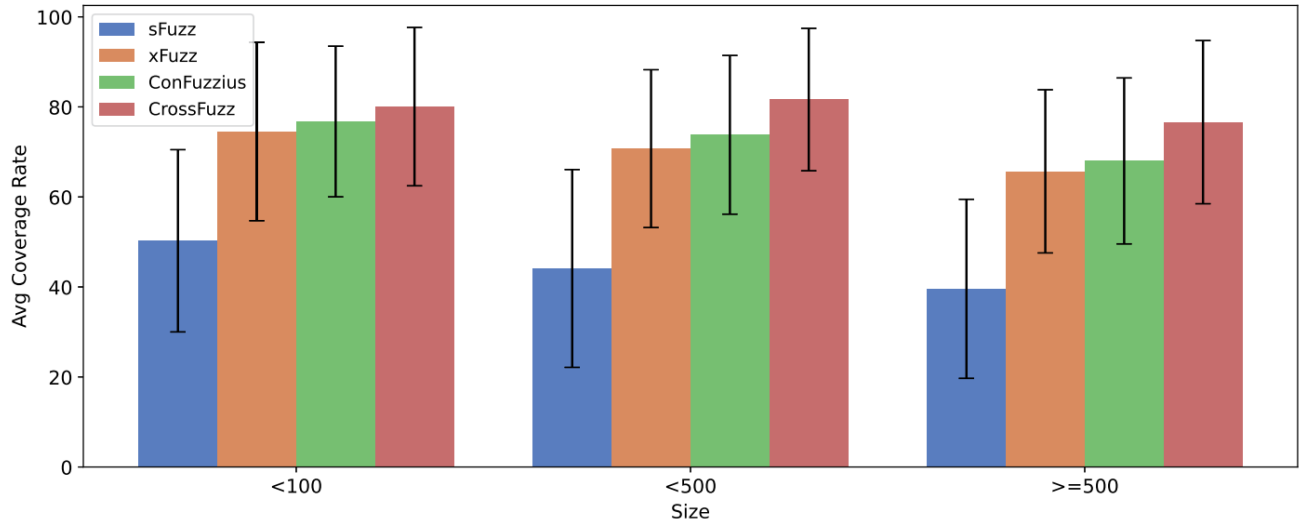
a. Kết quả cho RQ1:



Hình 2: Comparison of enabling the transaction sequence mutation strategy with different activation probabilities.

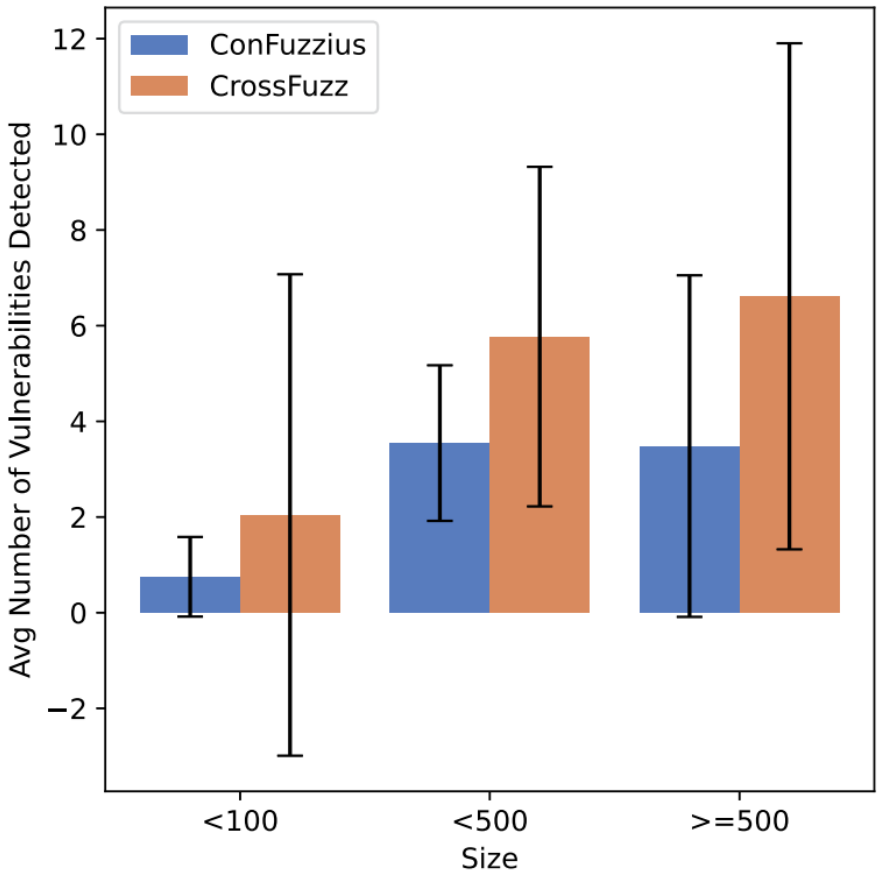
Để đánh giá ảnh hưởng của chiến lược đột biến chuỗi giao dịch, các nhà nghiên cứu đã thử nghiệm CrossFuzz với các xác suất kích hoạt chiến lược này khác nhau (0%, 20%, 50%, 80% và 100%). Kết quả cho thấy việc kích hoạt chiến lược đột biến chuỗi giao dịch dựa trên ICDF giúp cải thiện độ bao phủ mã của CrossFuzz, với xác suất tối ưu là 80%.

b. Kết quả cho RQ2:



Hình 3: Comparison of different fuzzing tools in terms of coverage.

CrossFuzz cho thấy hiệu quả vượt trội so với các fuzzer khác về độ bao phủ mã và khả năng phát hiện lỗ hổng. Độ bao phủ mã bytecode trung bình của CrossFuzz (79.92%) cao hơn ConFuzzius (72.11%), xFuzz (69.34%) và sFuzz (43.03%). Về số lượng lỗ hổng được phát hiện, CrossFuzz phát hiện được trung bình 6.11 (±5.08) lỗ hổng cho mỗi hợp đồng, cao hơn 1.82 lần so với ConFuzzius (3.36 ±3.52). CrossFuzz đạt hiệu quả cao hơn ở các hợp đồng có kích thước lớn do khả năng bao phủ nhiều mã hơn, đặc biệt là các hàm cốt lõi phức tạp thường dễ chứa lỗ hổng.



Hình 4: Comparison of different fuzzing tools in terms of number of vulnerabilities detected.

Vulnerability Type	Number of vulnerabilities detected	
	ConFuzzius	CrossFuzz
Arbitrary Memory Access	20	38
Assertion Failure	1712	2189
Integer Overflow	771	955
Reentrancy	15	672
Transaction Order Dependency	229	944
Block Dependency	3477	4537
Unhandled Exceptions	423	2756
Unsafe Delegate Call	0	0
Leaking Ether	15	1
Locking Ether	0	0
Unprotected Self-Destruct	0	0
Sum	6662	12092

Hình 5: Comparison of the number of vulnerabilities detected by ConFuzzius and CrossFuzz

c. Kết quả cho RQ3:

Size	Statements in Constructor		Methods	Avg Coverage	Avg Number of Vulnerabilities Detected
	# Total	# Contract Conversions			
<100	13	0	$\text{CrossFuzz}_{\text{zero}}$	79.62 (±21.69)	1.88 (±1.62)
			$\text{CrossFuzz}_{\text{random}}$	79.69 (±21.75)	1.92 (±1.60)
			CrossFuzz	80.06 (±21.88)	2.04 (±1.62)
<500	726	68	$\text{CrossFuzz}_{\text{zero}}$	77.49 (±18.31)	4.88 (±4.57)
			$\text{CrossFuzz}_{\text{random}}$	80.09 (±17.99)	5.42 (±4.80)
			CrossFuzz	81.63 (±18.07)	5.77 (±4.97)
>=500	332	62	$\text{CrossFuzz}_{\text{zero}}$	71.11 (±18.89)	5.29 (±4.83)
			$\text{CrossFuzz}_{\text{random}}$	73.56 (±18.22)	6.19 (±5.14)
			CrossFuzz	77.01 (±76.61)	6.61 (±5.19)
Sum	1071	130	$\text{CrossFuzz}_{\text{zero}}$	75.58 (±18.98)	4.83 (±4.60)
			$\text{CrossFuzz}_{\text{random}}$	77.97 (±18.56)	5.46 (±4.88)
			CrossFuzz	79.92 (±18.34)	5.82 (±5.08)

Hình 6: Comparison of CrossFuzz with different contractor parameter generation strategies

CrossFuzz được thử nghiệm với ba chiến lược tạo tham số constructor: sử dụng ***CrossFuzzrandom*** và ***CrossFuzzzero***(địa chỉ ngẫu nhiên, địa chỉ zero) và theo dõi đường dẫn lan truyền dữ liệu. Kết quả cho thấy việc theo dõi đường dẫn lan truyền dữ liệu để tạo tham số constructor giúp CrossFuzz đạt được độ bao phủ mã và số lượng lỗ hổng phát hiện cao hơn so với hai chiến lược còn lại. Đặc biệt, CrossFuzz cho thấy hiệu quả rõ rệt ở các hợp đồng có kích thước lớn (>= 500 dòng) và có nhiều câu lệnh chuyển đổi hợp đồng trong constructor.

2.4.3. Thảo luận

Phần này phân tích các mối đe dọa đến tính hợp lệ của nghiên cứu từ ba khía cạnh: tính hợp lệ nội bộ, tính hợp lệ bên ngoài và tính hợp lệ cấu trúc, đồng thời đề xuất các cải tiến trong tương lai cho CrossFuzz.

Các mối đe dọa đến tính hợp lệ:

- Tính hợp lệ nội bộ:

+ Mối đe dọa: Liệu việc triển khai CrossFuzz có chính xác hay không.

+ Giải pháp: CrossFuzz được triển khai dựa trên các công cụ mã nguồn mở, đảm bảo tính chính xác và minh bạch. Mô-đun phân tích tĩnh được xây dựng dựa trên Slither, một công cụ phân tích tĩnh cho hợp đồng thông minh được cập nhật thường xuyên và có độ tin cậy cao. Mô-đun fuzz testing được phát triển dựa trên framework ConFuzzius, với các tính năng mở rộng như triển khai đa hợp đồng, cài đặt tham số constructor và chiến lược đột biến chuỗi giao dịch.

- Tính hợp lệ bên ngoài:

+ Mối đe dọa: Liệu kết quả thử nghiệm có thể được khái quát hóa hay không, dựa trên việc tập dữ liệu có đại diện hay không.

+ Giải pháp: Do các nghiên cứu trước đây về phát hiện lỗ hổng đa hợp đồng chưa công khai tập dữ liệu, nhóm nghiên cứu đã lựa chọn ngẫu nhiên 500 hợp đồng thông minh từ EtherScan, một nền tảng phân tích cho Ethereum, để xây dựng tập dữ liệu. Việc so sánh CrossFuzz với sFuzz, xFuzz và ConFuzzius, bao gồm cả xFuzz là công cụ duy nhất tập trung vào lỗ hổng đa hợp đồng, giúp tăng cường tính khái quát của kết quả. Nhóm nghiên cứu đã công khai CrossFuzz như một công cụ mã nguồn mở, tạo điều kiện cho các nghiên cứu tiếp theo sử dụng và đánh giá.

- Tính hợp lệ cấu trúc:

+ Mối đe dọa: Liệu các chỉ số đánh giá được sử dụng có thể đánh giá toàn diện hiệu quả của CrossFuzz hay không.

+ Giải pháp: Nghiên cứu sử dụng hai chỉ số đánh giá phổ biến trong fuzz testing hợp đồng thông minh: độ bao phủ bytecode và số lượng lỗ hổng được phát hiện. Ngoài ra, nghiên cứu cũng đánh giá hiệu quả thời gian của CrossFuzz, bổ sung cho việc đánh giá hiệu quả.

CHƯƠNG 3. KỊCH BẢN VÀ KẾT QUẢ TRIỂN KHAI

3.1. So sánh giữa CrossFuzz và ConFuzzius

Trong kịch bản này, nhóm sẽ tiến hành so sánh độ bao phủ mã nguồn (code coverage) và nhánh (branch coverage) trong cùng 1 hợp đồng giữa ConFuzzius và phiên bản cải tiến của nó – CrossFuzz. Các thông số đều được để mặc định theo hướng dẫn của nhóm nghiêm cứu, số lần kiểm thử là 5.

Hợp đồng được dùng để kiểm thử là hợp đồng T.sol, trong đó hợp đồng E là hợp đồng chính dùng để kiểm thử:

```
pragma solidity 0.4.26;


contract Sub {
    mapping(address => uint) public balances;

    function addBalances(address _addr, uint _amount) public {
        balances[_addr] += _amount;
    }

    function checkBalance(address _addr) public view returns (uint) {
        return balances[_addr];
    }
}

contract Child {
    uint inner;
}

contract E is Child {
    Sub sub;
    uint count;
    bool flag;

    function E(Sub _sub) public {
        sub = Sub(_sub);
    }

    function setSub(Sub _sub) public {
        sub = Sub(_sub);
    }

    function setCount(uint _count) public {
        count = _count;
    }
```

```
    modifier minBalance {
        require(sub.checkBalance(msg.sender) >= 1 ether);
        _;
    }

    function addBalance(address _addr, uint _amount) public {
        sub.addBalances(_addr, _amount);
        count += 1;
    }

    function getFlag() public returns (bool) {
        return flag;
    }

    function setFlag(bool _flag) public {
        flag = _flag;
    }

    function getInner() public returns (uint) {
        return inner;
    }

    function withdraw(address _addr, uint _amount) public minBalance {
        if (count > 50) {
            revert();
        } else {
            count += 2;
        }
        _addr.transfer(_amount);
    }

}
```

Tiến hành thực hiện kiểm thử trên công cụ ConFuzzius:

```
wanthinnn@ThinLinux:~/Documents/NT521/project/ConFuzzius$ python3 fuzzer/main.py -s examples/T.sol -c E --solc v0.4.26 --evm byzantium -t 10
```

Kết quả:

```
2024-12-29 14:38:45,849 - Analysis - INFO - ------------------------------------------------------
2024-12-29 14:38:45,849 - Analysis - INFO - Number of generations:       44
2024-12-29 14:38:45,849 - Analysis - INFO - Number of transactions:      2538 (684 unique)
2024-12-29 14:38:45,849 - Analysis - INFO - Transactions per second:     253
2024-12-29 14:38:45,849 - Analysis - INFO - Total code coverage:         84.24% (449/533)
2024-12-29 14:38:45,849 - Analysis - INFO - Total branch coverage:       82.61% (38/46)
2024-12-29 14:38:45,849 - Analysis - INFO - Total execution time:        10.02 seconds
2024-12-29 14:38:45,850 - Analysis - INFO - Total memory consumption:    67.64 MB
wanthinnn@ThinLinux:~/Documents/NT521/project/ConFuzzius$
```

Tiếp theo, nhóm sẽ tiến hành thực hiện kiểm thử tương tự trên CrossFuzz:

```
(myenv) wanthinnn@ThinLinux:~/Documents/NT521/project/CrossFuzz$ python3 CrossFuzz.py --help
Usage: python CrossFuzz.py <file_path> <contract_name> <solc_version> <max_trans_length> <fuzz_time> <res_saved_path> <solc_path> <c
onstructor_params_path> <trans_duplication>

Arguments:
  file_path             Path to the Solidity file to be fuzzed
  contract_name         Name of the contract to be fuzzed
  solc_version          Supported Solidity compiler version (0.4.24, 0.4.26, 0.6.12, 0.8.4)
  max_trans_length      Maximum transaction length (e.g., 10)
  fuzz_time             Fuzzing duration in seconds (e.g., 60)
  res_saved_path        Path to save the result JSON (e.g., ./result.json)
  solc_path             Path to the Solidity compiler (solc)
  constructor_params_path Path to constructor parameters (e.g., 'auto' or './examples/p.json')
  trans_duplication     Duplicate transactions: 0 (no), 1 (yes)

(myenv) wanthinnn@ThinLinux:~/Documents/NT521/project/CrossFuzz$ python3 CrossFuzz.py examples/T.sol E 0.4.26 10 10 ./res.json myenv
/bin/solc auto 1
Running fuzzing for contract E in file examples/T.sol...
```

Kết quả:

```
INFO:Analysis:----------------------------------------------------------
INFO:Analysis:Number of generations:     14
INFO:Analysis:Number of transactions:    3249 (227 unique)
INFO:Analysis:Transactions per second:   302
INFO:Analysis:Total code coverage:       96.46% (682/707)
INFO:Analysis:Total branch coverage:     91.30% (42/46)
INFO:Analysis:Total execution time:      10.74 seconds
INFO:Analysis:Total memory consumption:          96.14 MB
(myenv) wanthinnn@ThinLinux:~/Documents/NT521/project/CrossFuzz$
```

Thực hiện tương tự những bước trên 5 lần cho cả hai công cụ, ta có nhận xét như sau: Về lỗ hỏng, thì cả hai công đều phát hiện được 2 lỗ hỏng là Integer overflow detected và Unchecked return value detected. Tuy nhiên, về độ bao phủ mã nguồn và nhánh thì CrossFuzz tỏ ra vượt trội hơn với kết quả bao phủ đều trên 90%, so với ConFuzzius thì chỉ loanh quanh 80%. Ngoài ra, CrossFuzz sẽ tiêu thụ bộ nhớ cao hơn tầm 10-15MB.

Qua đó có thể nhận thấy rằng, CrossFuzz mang lại khả năng phát hiện lỗi tốt hơn trong các tình huống yêu cầu độ bao phủ mã nguồn và nhánh cao, mặc dù phải đánh đổi một chút về hiệu suất bộ nhớ. Với độ bao phủ trên 90%, CrossFuzz phù hợp cho các hợp đồng phức tạp cần kiểm thử toàn diện, đặc biệt khi mục tiêu là giảm thiểu các lỗi ẩn sâu trong logic của hợp đồng.

Ngược lại, ConFuzzius có thể là lựa chọn tốt hơn trong các trường hợp yêu cầu tối ưu tài nguyên hệ thống, không cần kiểm chéo hợp đồng, do tiêu thụ ít bộ nhớ hơn và vẫn phát hiện được các lỗ hổng cơ bản như *Integer overflow* hay *Unchecked return value*. Tuy nhiên, với độ bao phủ mã nguồn chỉ đạt khoảng 80%, ConFuzzius có thể bỏ sót một số trường hợp đặc biệt hoặc nhánh logic khó tiếp cận.

3.2. So sánh giữa CrossFuzz và xFuzz

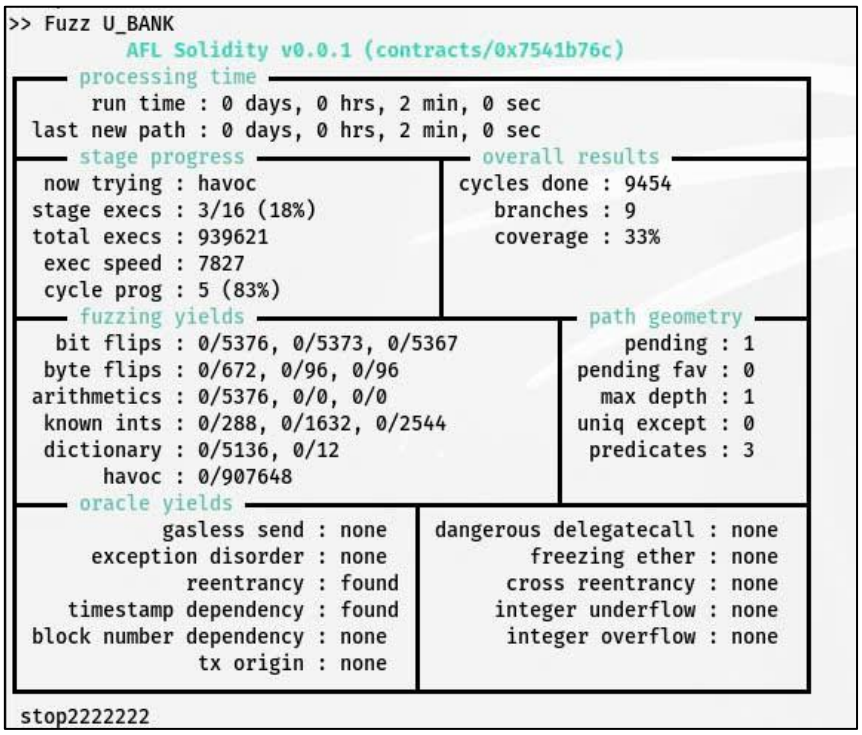
Trong kịch bản này, nhóm sẽ tiến hành so sánh độ bao phủ mã nguồn (code coverage) và nhánh (branch coverage) giữa xFuzz và CrossFuzz trên cùng một tập hợp các hợp đồng thông minh. Các thông số cấu hình của hai công cụ đều được giữ ở mức mặc định theo hướng dẫn của nhóm nghiên cứu, và số lần kiểm thử cho mỗi công cụ là 5 lần.

Tập hợp kiểm thử bao gồm 40 hợp đồng thông minh được thiết kế có sự phụ thuộc lẫn nhau, tức là các hợp đồng này có các lời gọi chéo (cross-contract calls) trong mã nguồn. Điều này nhằm mô phỏng các kịch bản thực tế phức tạp, nơi các hợp đồng phải tương tác với nhau để thực hiện các chức năng liên quan.

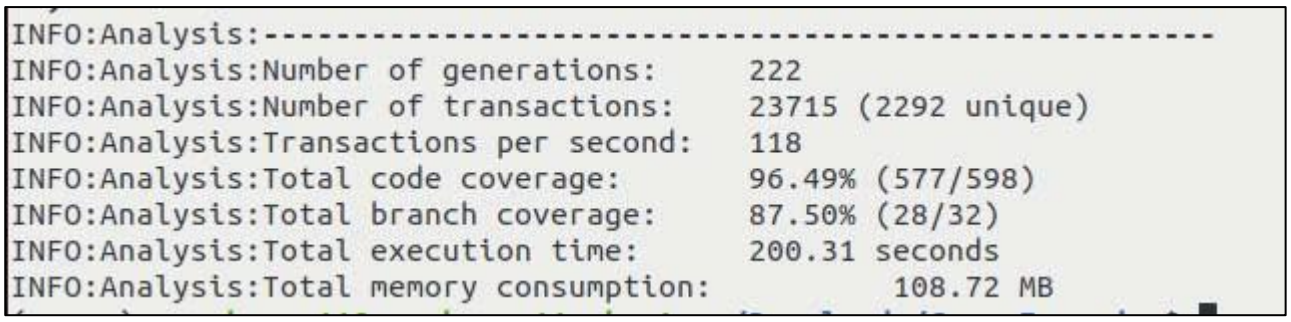
Việc sử dụng các hợp đồng phụ thuộc lẫn nhau giúp đánh giá khả năng của các công cụ trong việc xử lý các kịch bản đa hợp đồng và phát hiện lỗi trên các nhánh logic có tính liên kết cao.

Kết quả kiểm thử được trình bày dưới đây như sau:

- xFuzz:



- CrossFuzz:



Thực hiện tương tự những bước trên 5 lần cho cả hai công cụ, ta có nhận xét như sau: Về lỗ hỏng, thì cả hai công đều phát hiện được 2 lỗ hỏng là Reentrancy và Timestamp Dependency. Tuy nhiên, CrossFuzz tỏ ra vượt trội hơn khi phát hiện được cả những lỗ hỏng: Block dependency và Unchecked return value. Về độ bao phủ mã nguồn và bao phủ nhánh thì CrossFuzz tỏ ra vượt trội hơn với kết quả bao phủ đều trên 80%, so với xFuzz thì chỉ loanh quanh 30% đến 40%.

Qua đó có thể nhận thấy rằng, CrossFuzz không chỉ có khả năng phát hiện lỗ hỏng tốt hơn, mà còn có hiệu quả vượt trội hơn trong việc bao phủ mã nguồn và nhánh. Điều này cho thấy CrossFuzz có khả năng khai thác sâu hơn các nhánh logic và các tương tác phức tạp giữa các hợp đồng thông minh.

Trong khi đó, xFuzz mặc dù khả năng phát hiện được các lỗ hổng cũng không kém cạnh, nhưng khả năng bao phủ mã nguồn và nhánh còn hạn chế, dẫn đến việc bỏ sót các lỗ hổng tiềm tàng có tính phức tạp cao hơn.

Từ kết quả này, ta có thể khẳng định rằng CrossFuzz là một công cụ ưu việt hơn trong việc kiểm thử bảo mật các hợp đồng thông minh phức tạp, đặc biệt là trong các hệ thống có sự phụ thuộc và tương tác chéo giữa các hợp đồng.

3.3. Thử nghiệm kiểm thử CrossFuzz với những hợp đồng nhóm tự biên soạn

Trong kịch bản này, nhóm đã xây dựng một bộ gồm bốn file hợp đồng thông minh đơn giản (.sol), trong đó có hai hợp đồng được thiết kế an toàn và hai hợp đồng cố tình tích hợp các lỗ hổng bảo mật.

Các hợp đồng này được thiết kế với sự phụ thuộc và tương tác lẫn nhau, nhằm tái hiện các kịch bản phức tạp thường thấy trong hệ sinh thái blockchain. Mục tiêu là tạo ra một môi trường kiểm thử đa dạng và thực tế để đánh giá hiệu quả của các công cụ kiểm thử bảo mật.

Tên các hợp đồng lần lượt là **contract-1.sol, contract-2.sol, contract-3.sol và contract-4.sol**. Các thông số kiểm thử đều được để mặc định theo gợi ý của nhóm tác giả. Dưới đây là chi tiết kết quả kiểm thử.

3.3.1. Kiểm thử hợp đồng contract-1.sol và contract-2.sol

a. Hợp đồng contract-1.sol

Hợp đồng gồm:

- hợp đồng **Sub**: Quản lý số dư (balances) với các hàm addBalances và checkBalance.
- hợp đồng **E**: Kế thừa Child, tương tác với Sub, cung cấp các chức năng thêm số dư (addBalance), rút tiền (withdraw), và quản lý trạng thái (count, flag).

```
pragma solidity 0.4.26;


contract Sub {
    mapping(address => uint) public balances;

    function addBalances(address _addr, uint _amount) public {
        balances[_addr] += _amount;
    }

    function checkBalance(address _addr) public view returns (uint) {
        return balances[_addr];
    }
}

contract Child {
    uint inner;
}

contract E is Child {
    Sub sub;
    uint count;
    bool flag;
```

```
    function E(Sub _sub) public {
        sub = Sub(_sub);
    }

    function setSub(Sub _sub) public {
        sub = Sub(_sub);
    }

    function setCount(uint _count) public {
        count = _count;
    }

    modifier minBalance {
        require(sub.checkBalance(msg.sender) >= 1 ether);
        _;
    }

    function addBalance(address _addr, uint _amount) public {
        sub.addBalances(_addr, _amount);
        count += 1;
    }

    function getFlag() public returns (bool) {
        return flag;
    }

    function setFlag(bool _flag) public {
        flag = _flag;
    }

    function getInner() public returns (uint) {
        return inner;
    }

    function withdraw(address _addr, uint _amount) public minBalance {
        if (count > 50) {
            revert();
        } else {
            count += 2;
        }
        _addr.transfer(_amount);
    }

}
```

Tiến hành thực hiện kiểm thử:

```
(myenv) wanthinnn@ThinLinux:~/Documents/NT521/project/CrossFuzz$ python3 CrossFuzz.py demo
2024-12-29 15:27:38.504 | DEBUG    | comp:analysis_depend_contract:48 - Dependency contract found by analyzing written state variables: Sub
2024-12-29 15:27:38.505 | DEBUG    | comp:analysis_depend_contract:56 - Dependency contract found by analyzing function parameters: Sub
2024-12-29 15:27:38.505 | DEBUG    | comp:analysis_depend_contract:63 - Dependency contract found by analyzing written variables (both local and state): Sub
2024-12-29 15:27:38.505 | DEBUG    | comp:analysis_depend_contract:56 - Dependency contract found by analyzing function parameters: Sub
2024-12-29 15:27:38.505 | DEBUG    | comp:analysis_depend_contract:63 - Dependency contract found by analyzing written variables (both local and state): Sub
2024-12-29 15:27:38.506 | INFO     | comp:analysis_depend_contract:83 - Dependent contracts: {'Sub'}, total contracts: 3, contracts to deploy: 1
2024-12-29 15:27:38.506 | DEBUG    | comp:analysis_main_contract_constructor:157 - Constructor parameters: ['_sub contract Sub']
python3 fuzzer/main.py -s ./examples/contract-1.sol -c E --solc v0.4.26 --evm byzantium -t 10 --result fuzzer/result/res.json --cross-contract 1 --open-trans-comp 1 --depend-co
ntracts Sub --constructor-args _sub contract Sub --constraint-solving 1 --max-individual-length 10 --solc-path-cross /home/wanthinnn/Documents/NT521/project/CrossFuzz/myenv/bin
/solc --p-open-cross 80 --cross-init-mode 1 --trans-mode 1 --duplication 0
INFO:Main    :Contract ./examples/contract-1.sol has already been analyzed: fuzzer/result/res.json
INFO:Main    :The original test output file fuzzer/result/res.json has been deleted
INFO:Main    :Initializing seed to 0.004570123716796659
INFO:Fuzzer  :Fuzzing contract E
INFO:Fuzzer  :Dependent contract Sub deployed at        0x2c5e8a3b3aad9df32339409534e64dfcabcd3a65, created by 0xcafebabecafebabecafebabecafebabecafebabe
INFO:EVM:encode_values: ['0x2c5e8a3b3aad9df32339409534e64dfcabcd3a65']
INFO:Fuzzer  :主Contract deployed at 0x1c70b9d03f2387195cac4999476f9910bb887994
INFO:Detector:----------------------------------------------------
INFO:Detector:        !!! Unchecked return value detected !!!
INFO:Detector:----------------------------------------------------
INFO:Detector:SWC-ID:   104
INFO:Detector:Severity: Medium
INFO:Detector:----------------------------------------------------
INFO:Detector:Source code line:
INFO:Detector:----------------------------------------------------
INFO:Detector:./examples/contract-1.sol:43:1
INFO:Detector:sub.addBalances(_addr, _amount)
INFO:Detector:----------------------------------------------------
INFO:Detector:Transaction sequence:
INFO:Detector:----------------------------------------------------
INFO:Detector:Transaction - addBalance(address,uint256):
INFO:Detector:----------------------------------------------------
INFO:Detector:From:      0x2c5e8a3b3aad9df32339409534e64dfcabcd3a65
INFO:Detector:To:        0x1c70b9d03f2387195cac4999476f9910bb887994
INFO:Detector:Value:     0 Wei
INFO:Detector:Gas Limit: 4500000
INFO:Detector:Input:     0x21e5383a0000000000000000000000deadbeef
INFO:Detector:           deadbeefdeadbeefdeadbeefdeadbeefffffffff
INFO:Detector:           ffffffffffffffffffffffffffffffffffffffff
INFO:Detector:           fffffffffe
INFO:Detector:----------------------------------------------------
```

```
INFO:Detector:----------------------------------------------------
INFO:Detector:         !!! Integer overflow detected !!!
INFO:Detector:----------------------------------------------------
INFO:Detector:SWC-ID:   101
INFO:Detector:Severity: High
INFO:Detector:----------------------------------------------------
INFO:Detector:Source code line:
INFO:Detector:----------------------------------------------------
INFO:Detector:./examples/contract-1.sol:44:1
INFO:Detector:count += 1
INFO:Detector:----------------------------------------------------
INFO:Detector:Transaction sequence:
INFO:Detector:----------------------------------------------------
INFO:Detector:Transaction 1 - addBalance(address,uint256):
INFO:Detector:----------------------------------------------------
INFO:Detector:From:      0x2c5e8a3b3aad9df32339409534e64dfcabcd3a65
INFO:Detector:To:        0x1c70b9d03f2387195cac4999476f9910bb887994
INFO:Detector:Value:     0 Wei
INFO:Detector:Gas Limit: 4500000
INFO:Detector:Input:     0x21e5383a0000000000000000000002c5e8a3b
INFO:Detector:           3aad9df32339409534e64dfcabcd3a65ffffffff
INFO:Detector:           ffffffffffffffffffffffffffffffffffffffff
INFO:Detector:           fffffffffff
INFO:Detector:----------------------------------------------------
INFO:Detector:Transaction 2:
INFO:Detector:----------------------------------------------------
INFO:Detector:From:      0x2c5e8a3b3aad9df32339409534e64dfcabcd3a65
INFO:Detector:To:        0x2c5e8a3b3aad9df32339409534e64dfcabcd3a65
INFO:Detector:Value:     0 Wei
INFO:Detector:Gas Limit: 4500000
INFO:Detector:Input:     0x99c6d5f3000000000000000000000deadbeef
INFO:Detector:           deadbeefdeadbeefdeadbeefdeadbeefffffffff
INFO:Detector:           ffffffffffffffffffffffffffffffffffffffff
INFO:Detector:           fffffffffe
INFO:Detector:----------------------------------------------------
INFO:Detector:Transaction 3:
INFO:Detector:----------------------------------------------------
INFO:Detector:From:      0xdeadbeefdeadbeefdeadbeefdeadbeefdeadbeef
INFO:Detector:To:        0x2c5e8a3b3aad9df32339409534e64dfcabcd3a65
INFO:Detector:Value:     1 Wei
INFO:Detector:Gas Limit: 4500000
INFO:Detector:Input:     0x99c6d5f3000000000000000000000deadbeef
INFO:Detector:           deadbeefdeadbeefdeadbeefdeadbeef0000000000
INFO:Detector:           0000000000000000000000000000000000000000
INFO:Detector:           000000000001
INFO:Detector:----------------------------------------------------
INFO:Detector:Transaction 4 - setCount(uint256):
INFO:Detector:----------------------------------------------------
INFO:Detector:From:      0x2c5e8a3b3aad9df32339409534e64dfcabcd3a65
```

```
INFO:Analysis:----------------------------------------------------------
INFO:Analysis:Number of generations:     13
INFO:Analysis:Number of transactions:    2676 (211 unique)
INFO:Analysis:Transactions per second:   266
INFO:Analysis:Total code coverage:       97.45% (689/707)
INFO:Analysis:Total branch coverage:     93.48% (43/46)
INFO:Analysis:Total execution time:      10.04 seconds
INFO:Analysis:Total memory consumption:          95.97 MB
(myenv) wanthinnn@ThinLinux:~/Documents/NT521/project/CrossFuzz$
```

Nhận xét: Ta có thể thấy rằng, hợp đồng này tồn tại 2 lỗ hổng là Integer overflow và Unchecked return value, đây là 2 trong số 11 lỗ hổng phổ biến mà nhóm tác giả đã đề cập trước đó, ngoài ra, độ bao phủ mã nguồn và nhánh cũng khá cao, trên 90%.

Điều này cho thấy hợp đồng này, mặc dù đã được xây dựng với một số biện pháp bảo vệ như modifier minBalance, vẫn chưa đảm bảo an toàn tuyệt đối trước các lỗi cơ bản. Các lỗ hổng như Integer overflow và Unchecked return value là những lỗi có thể dẫn đến rủi ro nghiêm trọng trong môi trường thực tế nếu không được xử lý kịp thời.

Đặc biệt, độ bao phủ mã nguồn và nhánh đạt trên 90% cho thấy rằng công cụ kiểm thử đã hoạt động hiệu quả trong việc khám phá hầu hết các đường đi của chương trình. Tuy nhiên, độ bao phủ cao không đồng nghĩa với việc hợp đồng không còn lỗ hổng nào, bởi những vấn đề phức tạp hơn có thể không được phát hiện chỉ qua các bài kiểm thử cơ bản.

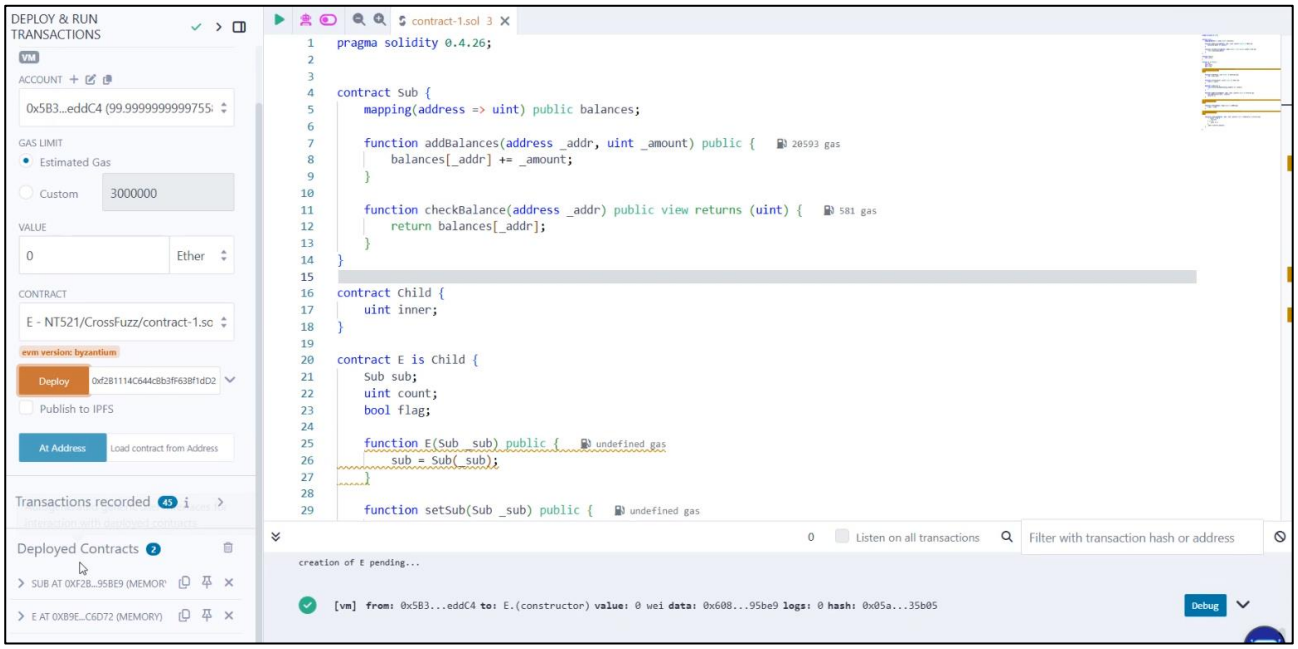
Do đó, nhóm khuyến nghị (sẽ được đề cập chi tiết ở *phần b*)

- Sử dụng thư viện an toàn như SafeMath để tránh lỗi Integer overflow.

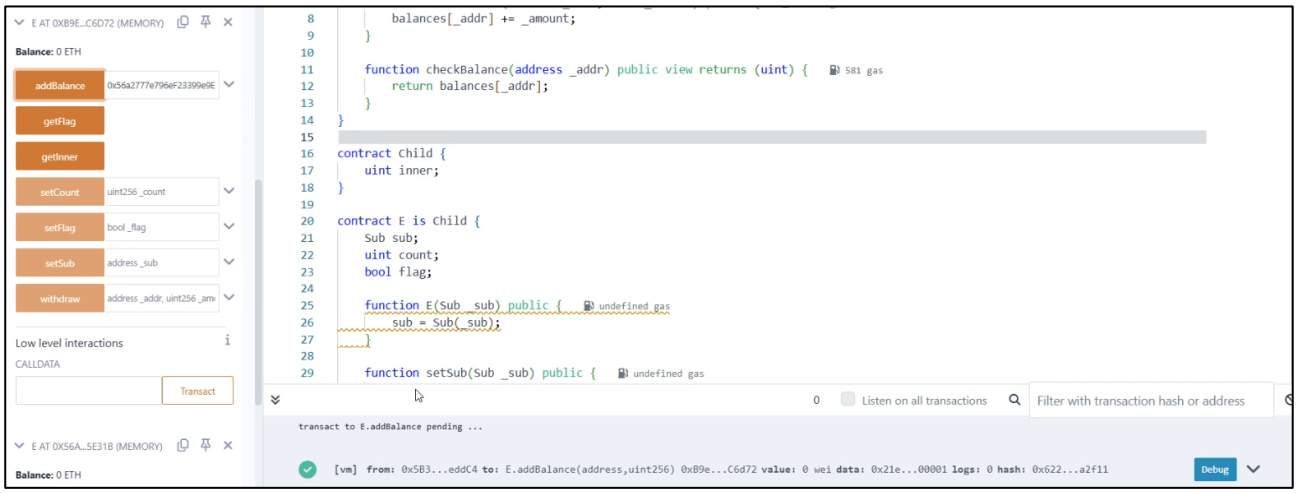
- Kiểm tra cẩn thận giá trị trả về của các hàm gọi hợp đồng khác nhằm loại bỏ lỗi Unchecked return value.

Thử nghiệm tính an toàn của hợp đồng: ta tiến hành triển khai hợp đồng trên IDE https://remix.ethereum.org

- Triển khai các hợp đồng E và Sub, trong đó E phụ thuộc vào Sub:

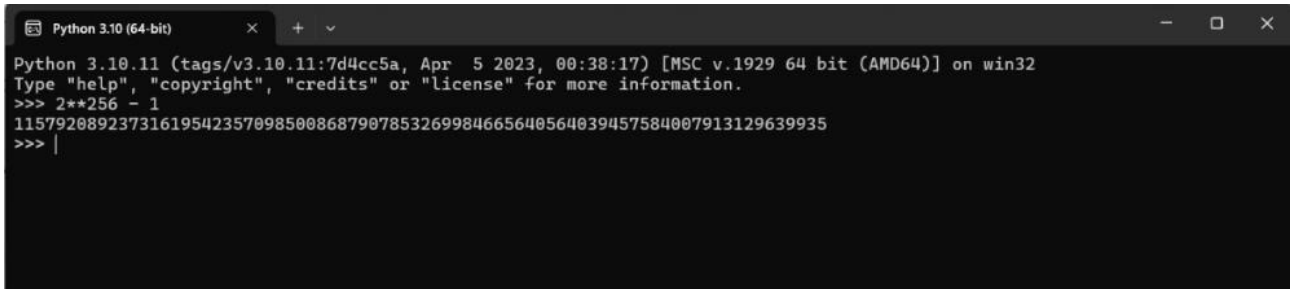


- Hợp đồng E1 sẽ thực hiện addBalance có ***giá trị 1*** cho hợp đồng E2:

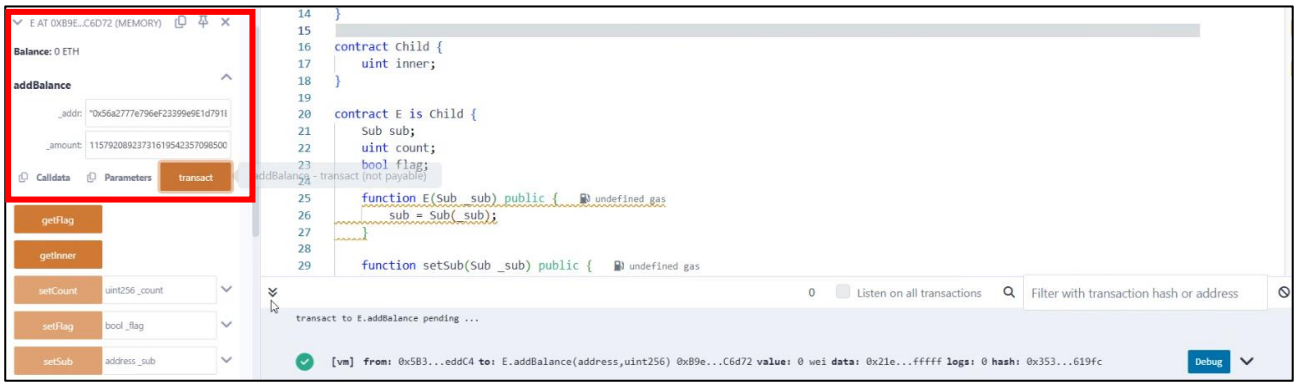


- Tiến hành kiểm tra số dư của hợp đồng E1 từ hợp đồng Sub, ta thấy lúc này số dư của E1 là 1. Chứng tỏ kết quả giao dịch đã thành công.

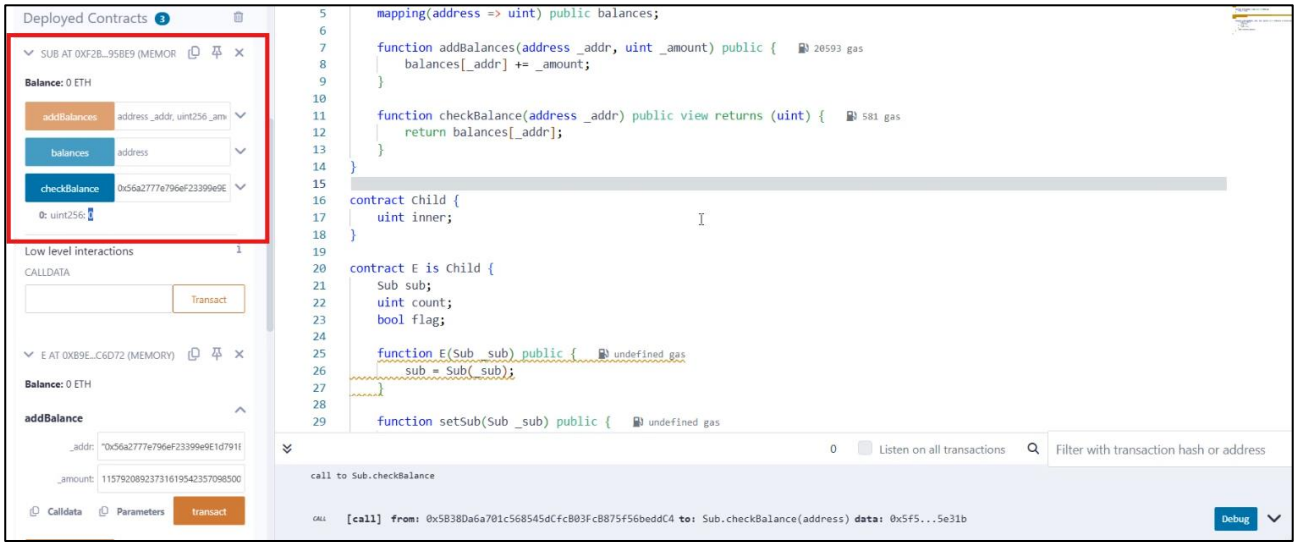
- Tiếp đến, ta tiến hành kiểm tra tràn số học trong giao dịch mới. Ta sử dụng Python để tính toán giá trị $2^{256} - 1$



- Thực hiện giao dịch lại thành công.



- Tiến hành kiểm tra lại số dư của hợp đồng E1 từ hợp đông Sub, ta thấy lúc này số dư của E1 là 0. Chứng tỏ đã xảy ra tràn số học khi ta thực hiện phép tính $(2^{256} – 1) + 1$



b. Hợp đồng contract-2.sol

Hợp đồng **contract-2.sol** là phiên bản cải tiến của hợp đồng ban đầu nhằm đảm bảo tính an toàn và loại bỏ hoàn toàn các lỗ hổng đã tồn tại.

```
pragma solidity ^0.4.26;

/**
 * @title SafeMath Library
 * @dev Math operations with safety checks that throw on error
 */
library SafeMath {
    function add(uint256 a, uint256 b) internal pure returns (uint256) {
        uint256 c = a + b;
        require(c >= a, "SafeMath: addition overflow");
        return c;
    }

    function sub(uint256 a, uint256 b) internal pure returns (uint256) {
        require(b <= a, "SafeMath: subtraction overflow");
        uint256 c = a - b;
        return c;
    }
}

contract Sub {
    using SafeMath for uint256;
    mapping(address => uint256) private balances;

    /**
     * @dev Add balance for a specific address
     * @param _addr Address to add balance to
     * @param _amount Amount to add
     * @return success True if the operation was successful
     */
    function addBalances(address _addr, uint256 _amount) public returns (bool) {
        require(_addr != address(0), "Invalid address");
        balances[_addr] = balances[_addr].add(_amount);
        return true;
    }

    /**
     * @dev Check balance of a specific address
     * @param _addr Address to check balance for
     * @return Balance of the address
     */
    function checkBalance(address _addr) public view returns (uint256) {
        return balances[_addr];
    }
}
```

```
contract Child {
    uint256 private inner;
}

contract E is Child {
    using SafeMath for uint256;

    Sub private sub;
    uint256 private count;
    bool private flag;

    event BalanceAdditionResult(address indexed addr, uint256 amount, bool
success);

    /**
     * @dev Constructor to set the Sub contract
     * @param _sub Address of the Sub contract
     */
    function E(Sub _sub) public {
        require(address(_sub) != address(0), "Invalid Sub contract address");
        sub = _sub;
    }

    /**
     * @dev Set a new Sub contract
     * @param _sub Address of the new Sub contract
     */
    function setSub(Sub _sub) public {
        require(address(_sub) != address(0), "Invalid Sub contract address");
        sub = _sub;
    }

    /**
     * @dev Add balance to the Sub contract
     * @param _addr Address to add balance to
     * @param _amount Amount to add
     */
    function addBalance(address _addr, uint256 _amount) public {
        require(_addr != address(0), "Invalid address");
        require(_amount > 0, "Amount must be greater than zero");

        // Kiểm tra giá trị trả về từ addBalances
        bool success = sub.addBalances(_addr, _amount);
        require(success, "Balance addition failed");

        count = count.add(1);

        emit BalanceAdditionResult(_addr, _amount, success);
    }
```

```
}
```

Những điểm nổi bật trong hợp đồng này bao gồm:

- Sử dụng thư viện SafeMath: Các phép toán cộng và trừ đều sử dụng SafeMath, đảm bảo kiểm tra tràn số (Integer overflow/underflow), giúp loại bỏ lỗ hổng phổ biến này.

- Cải thiện tính bảo mật: Mọi tham số quan trọng như địa chỉ (address) hoặc giá trị (uint256) đều được kiểm tra hợp lệ trước khi sử dụng. Ví dụ:

+ Kiểm tra địa chỉ không phải là address(0).

+ Yêu cầu giá trị _amount > 0 trước khi thêm số dư.

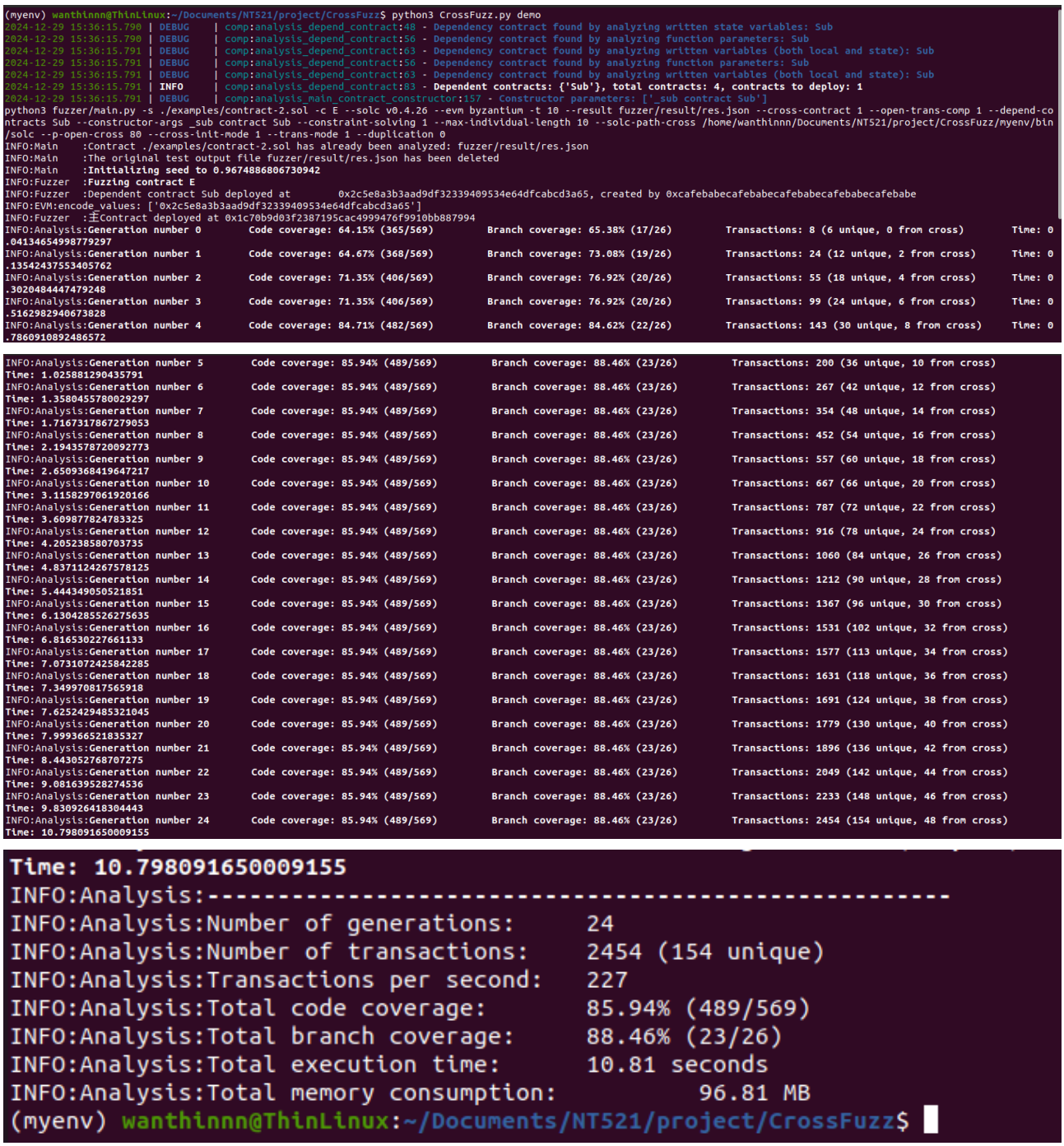
- Kiểm tra giá trị trả về từ các hàm quan trọng: Hàm addBalances của hợp đồng Sub trả về giá trị bool, và giá trị này được kiểm tra để đảm bảo rằng thao tác thêm số dư đã thành công. Điều này loại bỏ lỗ hổng Unchecked return value.

- Đóng gói dữ liệu an toàn: Các biến như inner, count, và flag trong hợp đồng Child và E đều được khai báo là private, ngăn chặn truy cập không mong muốn từ bên ngoài.

- Thêm sự kiện (event): Sự kiện BalanceAdditionResult được sử dụng để ghi lại kết quả thêm số dư, tăng khả năng theo dõi và giám sát hoạt động của hợp đồng.

- Kiến trúc hợp đồng chéo (Cross-Contract): Hợp đồng E phụ thuộc vào hợp đồng Sub để xử lý dữ liệu số dư, minh họa sự tương tác chặt chẽ giữa các hợp đồng thông minh.

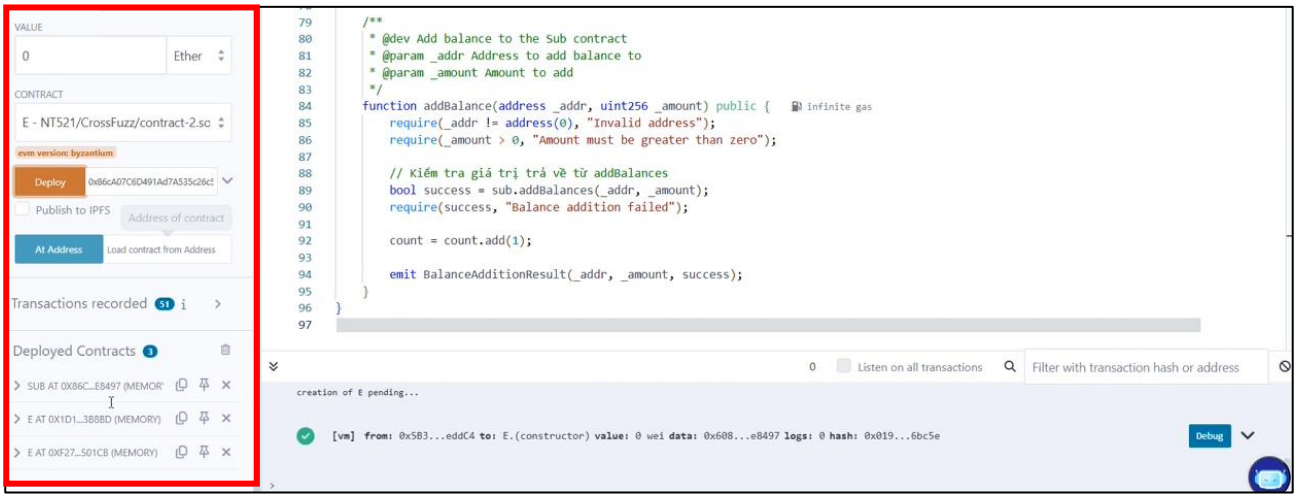
Tiến hành kiểm thử:



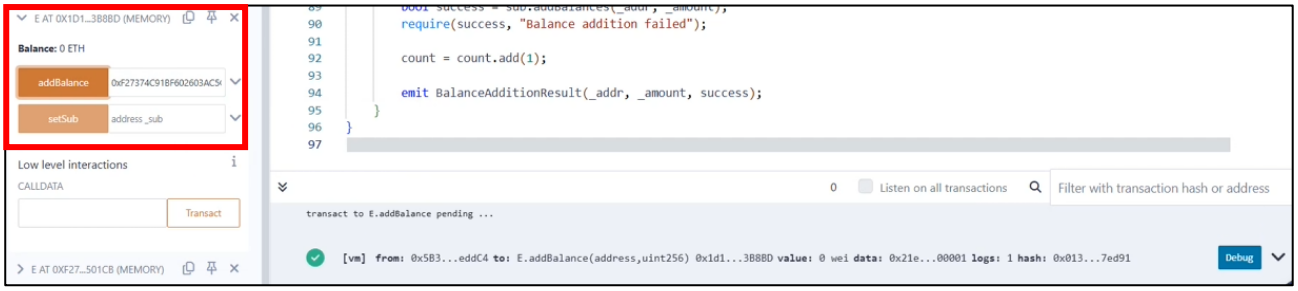
Nhận xét: các lỗ hổng ở phần trên đã được khắc phục, công cụ CrossFuzz không phát hiện thêm lỗi gì, tuy nhiên, độ bao phủ mã nguồn và nhánh có phần giảm nhẹ so với trước đó.

Thử nghiệm tính an toàn của hợp đồng: ta tiến hành triển khai hợp đồng trên IDE https://remix.ethereum.org

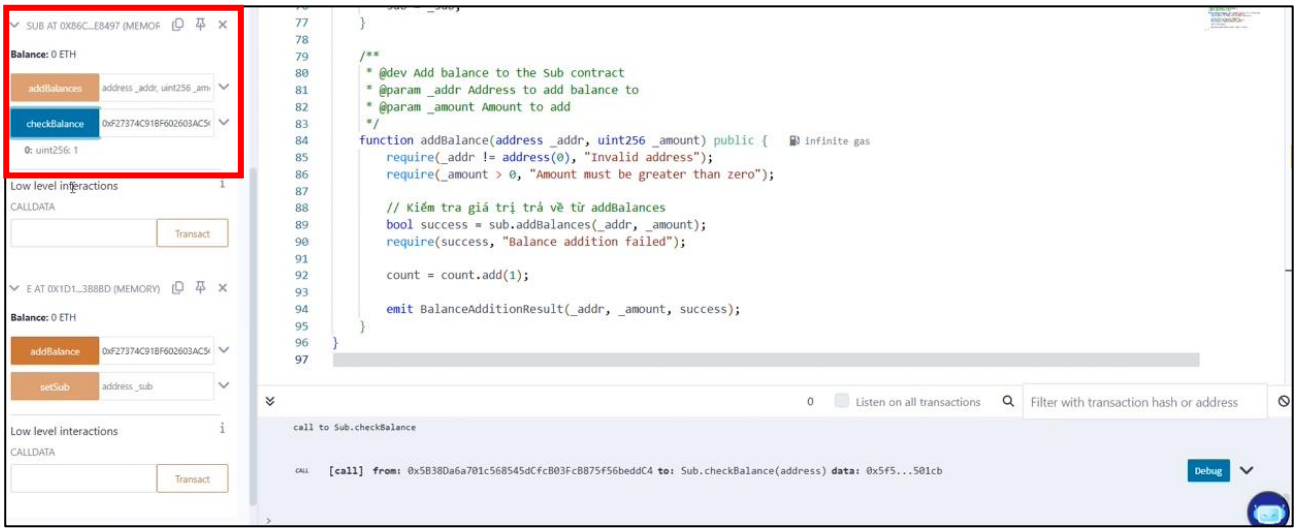
- Triển khai các hợp đồng E và Sub, trong đó E phụ thuộc vào Sub:



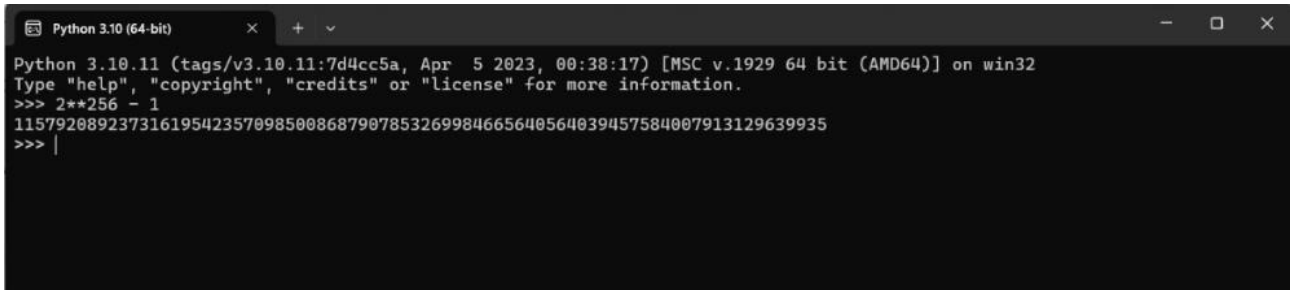
- Hợp đồng E1 sẽ thực hiện addBalance có ***giá trị 1*** cho hợp đồng E2:



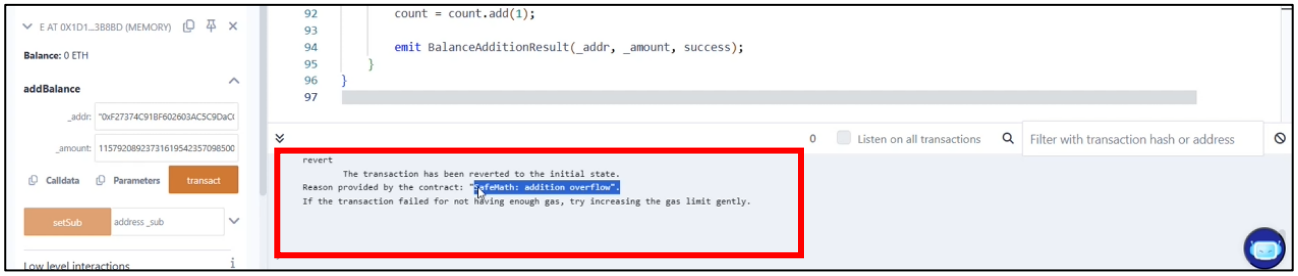
- Tiến hành kiểm tra số dư của hợp đồng E1 từ hợp đồng Sub, ta thấy lúc này số dư của E1 là 1. Chứng tỏ kết quả giao dịch đã thành công.



- Tiếp đến, ta tiến hành kiểm tra tràn số học trong giao dịch mới. Ta sử dụng Python để tính toán giá trị $2^{256} - 1$



- Thực hiện lại giao dịch, kết quả báo về không thành công, hệ thống báo lỗi tràn số học.



- Thực hiện 1 giao dịch khác với giá trị 1000000 => vẫn thành công.

- Tiến hành kiểm tra lại số dư của hợp đồng E1 từ hợp đồng Sub, giá trị mới là 1000001:

3.3.1. Kiểm thử hợp đồng contract-3.sol và contract-4.sol

a. Hợp đồng contract-3.sol

Hợp đồng này bao gồm hai hợp đồng chính: Manager và Worker.

- Hợp đồng Manager: Quản lý và phân công các tác vụ cho người lao động. Gồm các hàm

+ addTask: Thêm tác vụ cho người lao động. Chỉ chủ sở hữu mới có thể gọi hàm này.

+ getTaskCount: Lấy số lượng tác vụ của một người lao động.

+ getTaskByIndex: Lấy tác vụ tại chỉ số cụ thể trong danh sách của người lao động.

- Hợp đồng Worker: Đại diện cho người lao động và tương tác với hợp đồng Manager để thực hiện tác vụ. Gồm các hàm

+ performTask: Một hàm đơn giản để giả lập việc thực hiện tác vụ.

+ exploitAddTask: Lỗ hổng được cố tình thiết kế cho hợp đồng, cho phép người lao động tự thêm tác vụ vào danh sách của mình mà không cần sự cho phép của chủ sở hữu.

```
    pragma solidity 0.4.26;

contract Manager {
    mapping(address => string[]) public tasks;
    address public owner;

    constructor() public {
        owner = msg.sender;
    }

    // Thêm tác vụ cho một người dùng (Chỉ chủ sở hữu mới được phép gọi)
    function addTask(address _worker, string _task) public {
        require(msg.sender == owner, "Only owner can add tasks");
        tasks[_worker].push(_task);
    }

    // Lấy số lượng tác vụ của một người dùng
    function getTaskCount(address _worker) public view returns (uint256) {
        return tasks[_worker].length;
    }

    // Lấy tác vụ tại một chỉ số cụ thể
    function getTaskByIndex(address _worker, uint256 _index) public view returns
(string) {
        // Kiểm tra xem chỉ số có hợp lệ không
        require(_index < tasks[_worker].length, "Index out of bounds");
        return tasks[_worker][_index];
```

```
    }
}


contract Worker {
    Manager public manager;

    constructor(address _managerAddress) public {
        manager = Manager(_managerAddress);
    }

    // Thực hiện một tác vụ (giả định logic thực hiện)
    function performTask(string _task) public pure returns (string) {
        return _task;
    }

    // Lỗ hổng: Worker có thể tự thêm tác vụ cho mình bằng cách gọi hàm này
    function exploitAddTask(string _task) public {
        manager.addTask(address(this), _task); // Gọi trực tiếp addTask mà không bị
kiểm tra quyền
    }
}
```

Tiến hành kiểm thử:

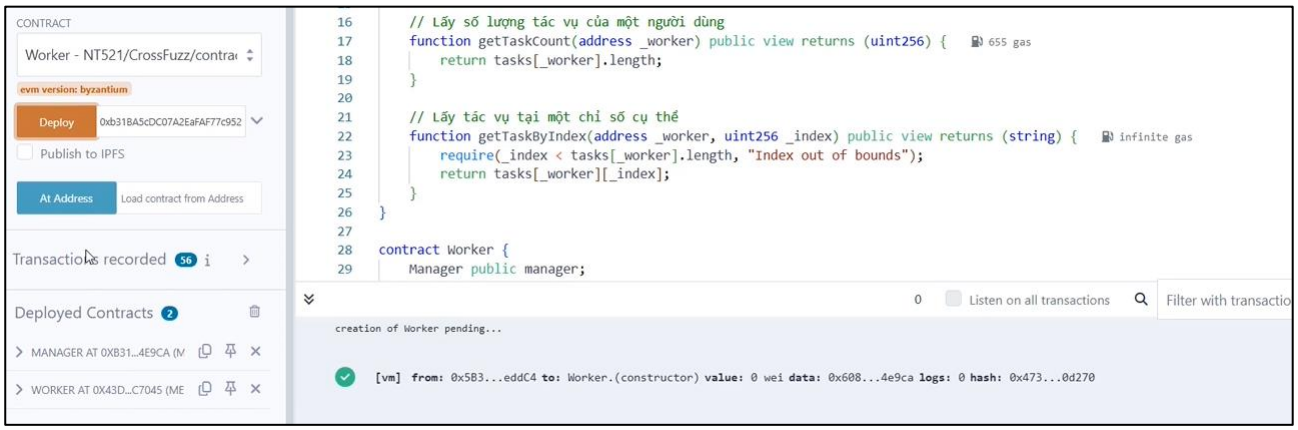
```
(myenv) wanthinnn@ThinLinux:~/Documents/NT521/project/CrossFuzz$ python3 CrossFuzz.py demo
2024-12-29 16:15:35.096 | DEBUG    | comp:analysis_depend_contract:48 - Dependency contract found by analyzing written state variables: Manager
2024-12-29 16:15:35.097 | DEBUG    | comp:analysis_depend_contract:63 - Dependency contract found by analyzing written variables (both local and state): Manager
2024-12-29 16:15:35.097 | INFO     | comp:analysis_depend_contract:83 - Dependent contracts: {'Manager'}, total contracts: 2, contracts to deploy: 1
2024-12-29 16:15:35.097 | DEBUG    | comp:analysis_main_contract_constructor:157 - Constructor parameters: ['_managerAddress address Manager']
python3 fuzzer/main.py -s ./examples/contract-3.sol -c Worker --solc v0.4.26 --evm byzantium -t 10 --result fuzzer/result/res.json --cross-contract 1 --open-trans-comp 1 --depe
nd-contracts Manager --constructor-args _managerAddress address Manager --constraint-solving 1 --max-individual-length 10 --solc-path-cross /home/wanthinnn/Documents/NT521/proj
ect/CrossFuzz/myenv/bin/solc --p-open-cross 80 --cross-init-mode 1 --trans-mode 1 --duplication 0
INFO:Main    :Contract ./examples/contract-3.sol has already been analyzed: fuzzer/result/res.json
INFO:Main    :The original test output file fuzzer/result/res.json has been deleted
INFO:Main    :Initializing seed to 0.25310831875043593
INFO:Fuzzer  :Fuzzing contract Worker
INFO:Fuzzer  :Dependent contract Manager deployed at    0x2c5e8a3b3aad9df32339409534e64dfcabcd3a65, created by 0xcafebabecafebabecafebabecafebabecafebabe
INFO:EVM:encode_values: ['0x2c5e8a3b3aad9df32339409534e64dfcabcd3a65']
INFO:Fuzzer  :Contract deployed at 0x1c70b9d03f2387195cac4999476f9910bb887994
INFO:Analysis:Generation number 0       Code coverage: 93.99% (516/549)         Branch coverage: 84.62% (22/26)         Transactions: 10 (8 unique, 0 from cross)       Time: 0
.03715658187866211
INFO:Analysis:Generation number 1       Code coverage: 97.63% (536/549)         Branch coverage: 92.31% (24/26)         Transactions: 27 (16 unique, 2 from cross)      Time: 0
.11557674407958984
INFO:Detector:-------------------------------------------------------
INFO:Detector:        !!! Unchecked return value detected !!!
INFO:Detector:-------------------------------------------------------
INFO:Detector:SWC-ID:    104
INFO:Detector:Severity: Medium
INFO:Detector:-------------------------------------------------------
INFO:Detector:Transaction sequence:
INFO:Detector:-------------------------------------------------------
INFO:Detector:Transaction - exploitAddTask(string):
INFO:Detector:-------------------------------------------------------
INFO:Detector:From:      0x2c5e8a3b3aad9df32339409534e64dfcabcd3a65
INFO:Detector:To:        0x1c70b9d03f2387195cac4999476f9910bb887994
INFO:Detector:Value:     0 Wei
INFO:Detector:Gas Limit: 4500000
INFO:Detector:Input:     0x2f5f93f700000000000000000000000000000000
INFO:Detector:           0000000000000000000000000000000020000000000
INFO:Detector:           0000000000000000000000000000000000000000000
INFO:Detector:           0000000000000000000000000000000000000000000
INFO:Detector:           00000000000000000000000000000000
INFO:Detector:-------------------------------------------------------
```

```
INFO:Analysis:-------------------------------------------------------
INFO:Analysis:Number of generations:     41
INFO:Analysis:Number of transactions:    1750 (348 unique)
INFO:Analysis:Transactions per second:   171
INFO:Analysis:Total code coverage:       97.63% (536/549)
INFO:Analysis:Total branch coverage:     92.31% (24/26)
INFO:Analysis:Total execution time:      10.25 seconds
INFO:Analysis:Total memory consumption:          100.14 MB
```

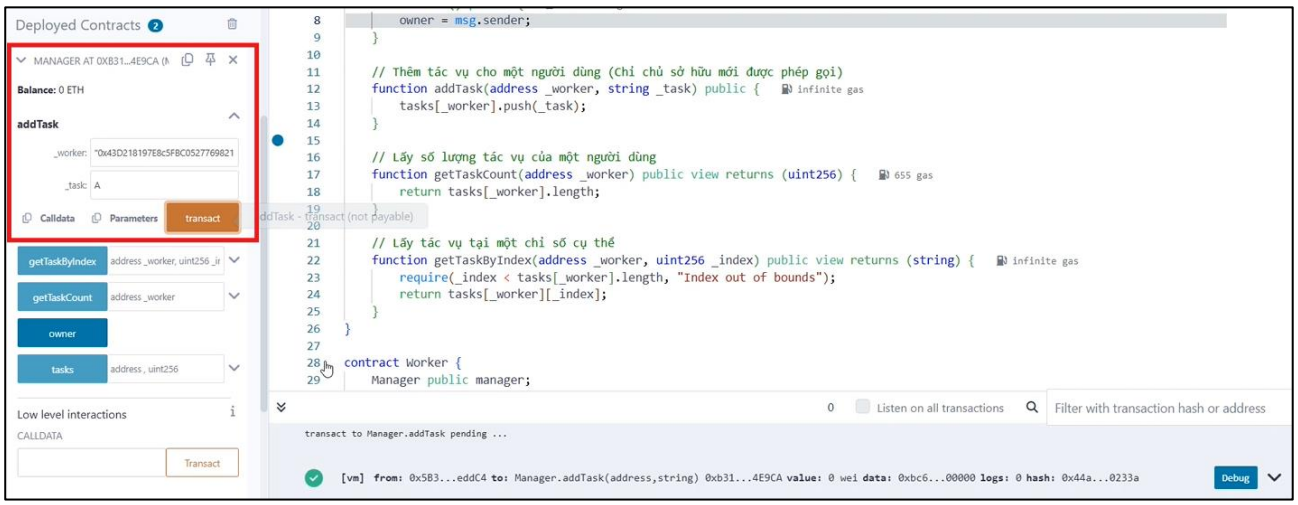
Nhận xét: Kết quả kiểm thử khi sử dụng công cụ CrossFuzz cho thấy quá trình fuzzing được thực hiện liên tục qua nhiều thế hệ, với mức độ bao phủ mã nguồn ngày càng cao. Sau mỗi thế hệ, cả độ bao phủ mã nguồn và độ bao phủ nhánh đều tăng đáng kể. Ban đầu, mã nguồn có khoảng 93.99% độ bao phủ mã nguồn và 84.62% độ bao phủ nhánh, nhưng sau 34 thế hệ, độ bao phủ mã nguồn duy trì ở mức 97.63% và độ bao phủ nhánh đạt 92.31%. Các giao dịch được thực hiện trong quá trình fuzzing bao gồm cả các giao dịch từ hợp đồng phụ, đồng thời các giao dịch này cũng trở nên đa dạng hơn qua mỗi thế hệ. Trong quá trình kiểm thử, công cụ đã phát hiện ra một lỗ hổng nghiêm trọng với mức độ nguy hiểm trung bình, liên quan đến việc không kiểm tra giá trị trả về trong một giao dịch Unchecked return value (SWC-ID: 104). Lỗi này có thể tiềm ẩn nguy cơ gây ra các vấn đề bảo mật trong ứng dụng. Mặc dù độ bao phủ mã và nhánh đạt được rất cao, công cụ cũng đã chỉ ra rằng một số vấn đề vẫn có thể tồn tại trong mã nguồn và cần được khắc phục.

Thử nghiệm tính an toàn của hợp đồng: ta tiến hành triển khai hợp đồng trên IDE https://remix.ethereum.org

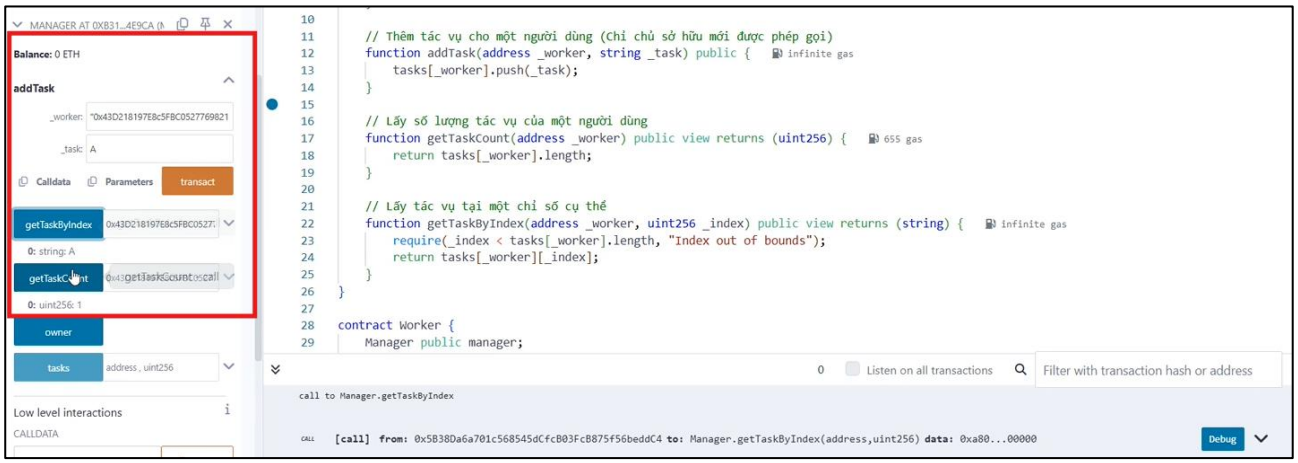
- Triển khai các hợp đồng Manager và Worker, trong đó Worker phụ thuộc vào Sub:



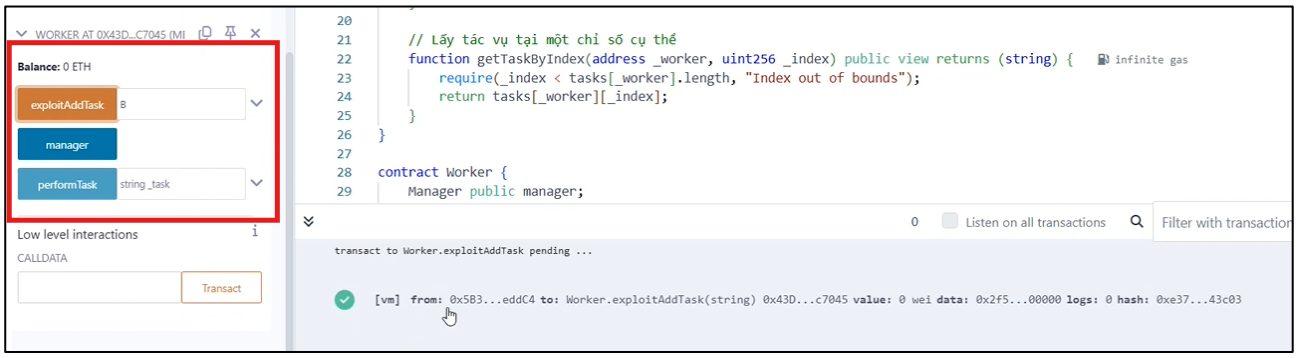
- Manager thực hiện giao Task “A” cho Worker:



- Phía Manager kiểm tra số task và tên task của Worker:



- Bên phía Worker tự giao task cho mình (vi phạm hợp đồng). Kết quả vẫn thành công, như vậy CrossFuzz đã phát hiện đúng lỗ hổng "Unchecked return value", khiến cho mọi Worker thuộc hợp đồng này đều có thể tự giao task cho mình.



- Bên phía Manager thực hiện check số task:

a. Hợp đồng contract-4.sol

Hợp đồng thông minh này là bản cải tiến của hợp đồng contract-3.sol trước đó, với hai hợp đồng riêng biệt: **Manager** và **Worker**. Hàm addTask của hợp đồng **Manager** đã được bảo vệ tốt bằng cách kiểm tra xem người gọi có phải là chủ sở hữu hay không.

```
pragma solidity 0.4.26;

contract Manager {
    mapping(address => string[]) public tasks;
    address public owner;

    constructor() public {
        owner = msg.sender;
    }

    // Thêm tác vụ cho một người dùng (Chỉ chủ sở hữu mới được phép gọi)
    function addTask(address _worker, string _task) public {
        require(msg.sender == owner, "Only owner can add tasks");
        tasks[_worker].push(_task);
    }

    // Lấy số lượng tác vụ của một người dùng
    function getTaskCount(address _worker) public view returns (uint256) {
        return tasks[_worker].length;
    }

    // Lấy tác vụ tại một chỉ số cụ thể
    function getTaskByIndex(address _worker, uint256 _index) public view returns
(string) {
        // Kiểm tra xem chỉ số có hợp lệ không
        require(_index < tasks[_worker].length, "Index out of bounds");
        return tasks[_worker][_index];
    }
}


contract Worker {
    Manager public manager;

    constructor(address _managerAddress) public {
        manager = Manager(_managerAddress);
    }

    // Thực hiện một tác vụ (giả định logic thực hiện)
    function performTask(string _task) public pure returns (string) {
        return _task;
    }
```

```
    // Lỗ hổng: Worker có thể tự thêm tác vụ cho mình bằng cách gọi hàm này
    function exploitAddTask(string _task) public {
        // So sánh địa chỉ hợp đồng manager với msg.sender
        require(msg.sender == address(manager), "Only owner can add tasks");
    }
}
```

Tiến hành kiểm thử:

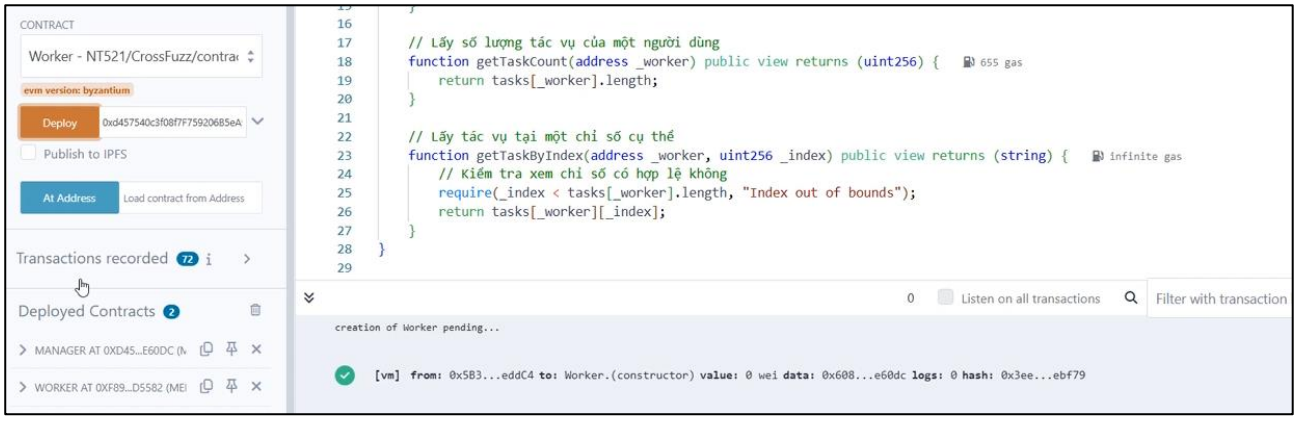
```
(myenv) wanthinnn@ThinLinux:~/Documents/NT521/project/CrossFuzz$ python3 CrossFuzz.py demo
2024-12-29 16:51:39.482 | DEBUG    | comp:analysis_depend_contract:48 - Dependency contract found by analyzing written state variables: Manager
2024-12-29 16:51:39.483 | DEBUG    | comp:analysis_depend_contract:63 - Dependency contract found by analyzing written variables (both local and state): Manager
2024-12-29 16:51:39.483 | INFO     | comp:analysis_depend_contract:83 - Dependent contracts: {'Manager'}, total contracts: 2, contracts to deploy: 1
2024-12-29 16:51:39.483 | DEBUG    | comp:analysis_main_contract_constructor:157 - Constructor parameters: ['_managerAddress address Manager']
python3 fuzzer/main.py -s ./examples/contract-4.sol -c Worker --solc v0.4.26 --evm byzantium -t 10 --result fuzzer/result/res.json --cross-contract 1 --open-trans-comp 1 --depe
nd-contracts Manager --constructor-args _managerAddress address Manager --constraint-solving 1 --max-individual-length 10 --solc-path-cross /home/wanthinnn/Documents/NT521/proj
ect/CrossFuzz/myenv/bin/solc --p-open-cross 80 --cross-init-mode 1 --trans-mode 1 --duplication 0
INFO:Main    :Contract ./examples/contract-4.sol has already been analyzed: fuzzer/result/res.json
INFO:Main    :The original test output file fuzzer/result/res.json has been deleted
INFO:Main    :Initializing seed to 0.5111581839312517
INFO:Fuzzer  :Fuzzing contract Worker
INFO:Fuzzer  :Dependent contract Manager deployed at    0x2c5e8a3b3aad9df32339409534e64dfcabcd3a65, created by 0xcafebabecafebabecafebabecafebabecafebabe
INFO:EVM:encode_values: ['0x2c5e8a3b3aad9df32339409534e64dfcabcd3a65']
INFO:Fuzzer  :主Contract deployed at 0x1c70b9d03f2387195cac4999476f9910bb887994
```

```
INFO:Analysis:Generation number 78       Code coverage: 99.77% (437/438)        Branch coverage: 100.00% (20/20)
Time: 10.065424919128418
INFO:Analysis:-----------------------------------------------------
INFO:Analysis:Number of generations:     78
INFO:Analysis:Number of transactions:    3367 (658 unique)
INFO:Analysis:Transactions per second:   335
INFO:Analysis:Total code coverage:       99.77% (437/438)
INFO:Analysis:Total branch coverage:     100.00% (20/20)
INFO:Analysis:Total execution time:      10.07 seconds
INFO:Analysis:Total memory consumption:          86.41 MB
(myenv) wanthinnn@ThinLinux:~/Documents/NT521/project/CrossFuzz$
```

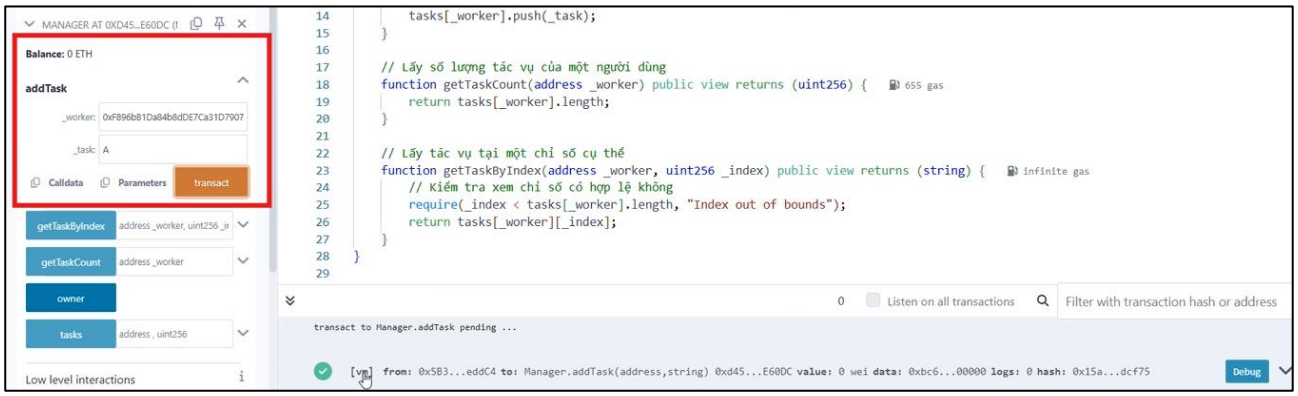
Nhận xét: Kết quả cho thấy công cụ đã đạt hiệu suất cao với mức độ bao phủ mã 99.77% (437/438) và bao phủ nhánh 100% (20/20). Đồng thời công cụ không phát hiện ra được bất cứ lỗ hổng nào trong 11 lỗ hổng phổ biến mà nhóm tác giả đề cập trước đó.

Thử nghiệm tính an toàn của hợp đồng: ta tiến hành triển khai hợp đồng trên IDE https://remix.ethereum.org

- Triển khai các hợp đồng Manager và Worker, trong đó Worker phụ thuộc vào Sub:

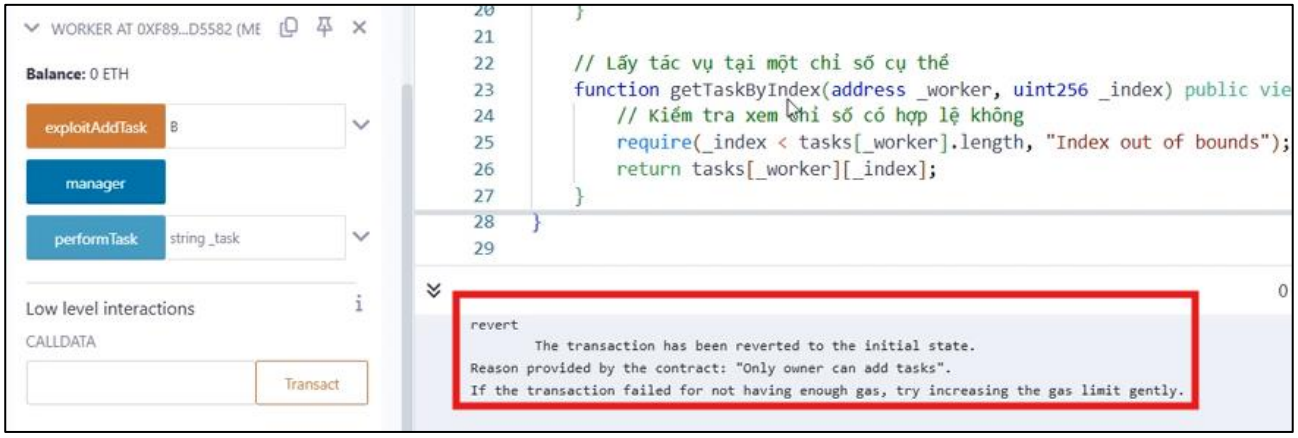


- Manager thực hiện giao Task “A” cho Worker:



- Phía Manager kiểm tra số task và tên task của Worker:

- Bên phía Worker tự giao task cho mình (vi phạm hợp đồng). Kết quả lần này không thành công. Như vậy, hợp đồng này đã hoàn toàn khắc phục được lỗ hỏng tồn tại trong hợp đồng contract-3.sol, và không có phát hiện được bất kỳ lỗ hỏng nào trong 11 lỗ hỏng phổ biến.



CHƯƠNG 4. KẾT LUẬN VÀ HƯỚNG PHÁT TRIỂN

4.1. Kết luận

Tóm lại, CrossFuzz là một công cụ kiểm thử fuzz được thiết kế đặc biệt để phát hiện lỗ hổng bảo mật **liên hợp đồng** trong các hợp đồng thông minh. CrossFuzz giải quyết hai thách thức chính: tạo tham số constructor và đột biến chuỗi giao dịch, nhằm cải thiện độ bao phủ mã và hiệu quả phát hiện lỗ hổng. CrossFuzz hoạt động bằng cách:

- **Theo dõi đường dẫn lan truyền dữ liệu** của các tham số kiểu địa chỉ trong tham số constructor để tạo ra các tham số phù hợp.

- **Phân tích mối quan hệ gọi hàm** giữa các hợp đồng để xác định định nghĩa và cách sử dụng biến trạng thái của mỗi hàm.

- **Thực hiện kiểm thử fuzz liên hợp đồng** bằng cách tạo và thực thi các chuỗi giao dịch.

- **Tối ưu hóa chiến lược đột biến** chuỗi giao dịch dựa trên ICDF (Inverse Cumulative Distribution Function) để tăng khả năng khám phá các đường dẫn thực thi khác nhau.

Các thí nghiệm trong bài báo cho thấy CrossFuzz vượt trội hơn so với các công cụ kiểm thử fuzz hiện có như sFuzz, ConFuzzius và xFuzz. Cụ thể, CrossFuzz:

- **Cải thiện độ bao phủ mã bytecode thêm 10,58%** so với xFuzz, một công cụ fuzzing tập trung vào lỗ hổng liên hợp đồng.

- **Phát hiện nhiều lỗ hổng bảo mật hơn 1,82 lần** so với ConFuzzius, một công cụ kiểm thử fuzz kết hợp kỹ thuật thực thi biểu tượng.

4.2. Hướng phát triển trong tương lai

Nhóm tác giả đã đưa ra một số cải tiến trong tương lai như sau

- **Mở rộng khả năng áp dụng** bằng cách hỗ trợ kiểm thử các trường hợp phức tạp hơn như hợp đồng gọi hợp đồng khác thông qua địa chỉ.

- **Nâng cao hiệu suất** bằng cách tối ưu hóa thuật toán tạo chuỗi giao dịch và xử lý các trường hợp đặc biệt.

- **Cải thiện khả năng phát hiện lỗ hổng** bằng cách tinh chỉnh cơ chế phân tích và xác định lỗ hổng.

Tính tới thời điểm hiện tại, những cải tiến trên vẫn chưa được thực hiện, và phiên bản CrossFuzz được công khai trên Github vẫn là phiên bản mới nhất.

4.3. Lời kết

Báo cáo này nhóm chúng em đã hoàn thành việc tìm hiểu và trình bày chi tiết về CrossFuzz, một công cụ kiểm thử fuzz tiên tiến dành cho hợp đồng thông minh, với khả năng giải quyết hiệu quả các thách thức trong việc phát hiện lỗ hổng bảo mật liên hợp đồng. Qua các thí nghiệm, CrossFuzz đã chứng minh được sự vượt trội so với các công cụ hiện có, cả về độ bao phủ mã bytecode và khả năng phát hiện lỗ hổng bảo mật.

Những kết quả này không chỉ khẳng định vai trò của CrossFuzz trong việc nâng cao chất lượng và tính an toàn của các hợp đồng thông minh mà còn mở ra tiềm năng phát triển các công cụ kiểm thử tương tự trong tương lai.

Nhóm hy vọng rằng báo cáo này sẽ đóng góp tích cực vào cộng đồng nghiên cứu về kiểm thử hợp đồng thông minh và thúc đẩy sự phát triển của các giải pháp bảo mật trong lĩnh vực blockchain.

Cuối lời, chúng em xin gửi lời cảm ơn sâu sắc đến thầy Phan Thế Duy, người đã đồng hành cùng chúng em trong suốt quá trình thực hiện đồ án. Thầy không chỉ truyền đạt kiến thức mà còn luôn sẵn lòng giải đáp mọi khó khăn, giúp chúng em hoàn thành công việc một cách tốt nhất. Những bài học và sự tận tụy của thầy là nguồn động lực lớn lao cho chúng em trên con đường học tập và phát triển. Kính chúc thầy luôn mạnh khỏe, hạnh phúc và tiếp tục lan tỏa đam mê tri thức đến nhiều thế hệ sinh viên và luôn luôn đạt được những thành công trong sứ mệnh cao cả của mình.

---------- HẾT ----------

DANH MỤC TÀI LIỆU THAM KHẢO

1. Cisco. Cisco Catalyst 2960-S and 2960 Series Switches with LAN Lite Software

2. Cisco. ISR4321/K9 Datasheet

3. The Art of Service - Kernel Based Virtual Machine Publishing. Kernel Based Virtual Machine A Complete Guide - 2021 Edition

4. Khaleel Ahmad, Ahamed Shareef. Hands-On Kernel-Based Virtual Machine (KVM)

5. Cong Li (2017). Kernel-based Virtual Machine