УДК 004.415.53 **DOI: 10.31673/2412-9070.2023.025459**

А. П. САМОЙЛЕНКО, аспірант;
А. П. БОНДАРЧУК, доктор техн. наук, професор,
Державний університет телекомунікацій, Київ

ПОРІВНЯННЯ КОМБІНОВАНОГО АЛГОРИТМУ ФАЗИНГУ НА ОСНОВІ МУТАЦІЙНОГО АНАЛІЗУ З АЛГОРИТМОМ ФАЗИНГУ НА ОСНОВІ ПОКРИТТЯ КОДУ

Здійснено порівняльний аналіз між алгоритмом μ2 та алгоритмом Zest із використанням однакової кількості часу. Здобуті висновки підтверджують, що фазинг на основі мутаційного аналізу потребує більше часу на генерацію вхідних даних, які забезпечують повне покриття коду. У результаті дослідження з'ясовано, що за рівних умов комбінований алгоритм на основі мутаційного аналізу забезпечує менше загального покриття коду порівняно з алгоритмом на основі покриття коду.

Ключові слова: фазинг; мутаційне тестування; мутаційний аналіз; генерація тестів.

Вступ

Постановка проблеми. Потреба в якості програмного продукту стає все більш нагальною, особливо коли зростає кількість користувачів цього продукту. Ручне тестування може бути вкрай неефективним для відшукання таких помилок, що не були відтворені під час тестування, оскільки може витрачатися велика кількість ресурсів. Автоматизоване тестування частково розв'язує цю проблему, але для багатьох схожих методик для написання тестів і для вибору тестових даних все ще потрібно залучати людські ресурси. Цей підхід окрім уже згаданої витрати часу має ще один недолік — вплив людського фактора. Очевидно, що впливу людського фактора можна позбутись, скориставшись таким видом тестування, котрий дав би змогу якщо не відкинути людський фактор, то хоча б мінімізувати його вплив. Реалізувати зазначену методику можна, застосувавши фазинг.

Фазинг було запропоновано Бартоном Міллером і його колегами в 1990 році. У своїй праці «An Empirical Study of the Reliability of UNIX Utilities» вони описали метод автоматичної генерації випадкових вхідних даних для тестування програмного забезпечення. Це один із видів тестування, за якого програмний продукт перевіряється на наявність багів великою кількістю випадково згенерованих вхідних даних. Такий підхід має низку переваг, якщо порівнювати з такими методиками, як юніт-тестування або функціональне тестування, оскільки деякі тести можуть взагалі бути пропущені під час традиційного тестування або ж бути неповними. Також фазинг дає змогу економити час, оскільки внаслідок упровадження такої методики процес тестування може бути повністю автоматизовано.

Найпростіший фазер може генерувати абсолютно випадкові значення, але такий спосіб створення вхідних даних є низькоефективним. Тому найпопулярніші фазери, зокрема libFuzzer та AFL, використовують фазинг на основі покриття коду.

Проте зі статті [1] можна дійти висновку, що корисна інформація в процесі застосування такої техніки тестування на покритті може досить швидко вичерпатися, і хоч адекватне покриття є досить важливою умовою, але недостатньою для відшукання всіх помилок. Тому найбільш надійним варіантом є мутаційний аналіз, оскільки ця техніка здатна виявляти проблеми, які неможливо було б виявити покриттям.

У мутаційного аналізу і фазерів є схожі проблеми, оскільки їм обом властива збитковість, а у фазерів це ще й обчислювальні витрати. Отже, зменшення збитковості й обчислювальних витрат є нагальним напрямом дослідження.

У статті [1] зазначено, що дослідження фазинг-тестування з мутаційним аналізом є досить перспективним, бо, по-перше, немає таких інструментів, які б використовували ці дві методики одночасно, а по-друге, відсутні дослідження.

Аналіз останніх досліджень і публікацій. Першу публікацію, де запропоновано застосовувати мутаційний аналіз і визначено подальші напрями дослідження, було видано 2022 року [1]. У ній розглядалися такі нагальні проблеми, що потребують розв'язку:

- знайти оптимізації, які не покладаються на статичні тест-кейси;
- знайти засоби ефективного виявлення мутантів;
- визначити, в який спосіб виявляти надлишкові та еквівалентні мутанти під час порівняння фазерів;
- створити фреймворки для мутаційного аналізу задля оцінювання ефективності фазерів;
- поліпшити обізнаність у мутаційному аналізі дослідників фазингу.

Один із запропонованих напрямів дослідження було застосовано для оцінювання ефективності фазерів на основі покриття коду [2]. Інша публікація [3] розвиває ідею застосування фазингу з мутаційних аналізом і імплементує фреймворк, що

використовує мутаційний аналіз замість метрики покриття коду.

Мета дослідження. Метою дослідження є порівняння ефективності комбінованого алгоритму фазингу, який базується на мутаційному аналізі, з алгоритмом фазингу, що використовує покриття коду як підхід для генерації тестових вхідних даних.

Основна частина

JQF [4] є фреймворком для імплементації алгоритму фазингу на основі покриття коду на мові програмування Java. Основним алгоритмом, що використовується за замовчуванням, є Zest, який спрямовано на здійснення фазингу з огляду на структурну та семантичну валідність вхідних даних. Інакше кажучи, Zest прагне створювати вхідні дані, які задовольняють не лише структурні вимоги, а й семантичні властивості, забезпечуючи максимальне покриття коду програми. Основною метою Zest є виявлення складних семантичних помилок, які не можуть бути знайдені за допомогою традиційних інструментів фазингу, що зазвичай акцентуються на виявленні проблем у логіці оброблення помилок. Завдяки цим особливостям, JQF стає потужним інструментом для тестування програмного забезпечення, що дає змогу виявляти більш складні та більш глибокі дефекти, що лежать за межами звичайних засобів фазингу.

μ2 [6] — це імплементація фазингу із застосуванням мутаційного аналізу, яка додає аналіз мутацій до критеріїв вибору вхідних даних, беручи до уваги покриття коду. Застосування методу мутацій дає змогу створювати змінені версії вхідних даних, відомих як «мутанти», що відображають потенційні дефекти в програмному коді. Після цього μ2 виконує ці мутанти на тестовому наборі і вимірює покриття коду, яке забезпечується кожним мутантом. Отже, μ2 поєднує покриття коду і аналіз мутацій, що уможливлює вибір вхідних даних для фазингу з огляду не лише на структурні вимоги, а й на потенційні дефекти в програмному коді. Це збільшує ефективність фазингу та допомагає відшукувати більше вразливостей та помилок у програмах.

Відмінність між алгоритмом фазингу на основі покриття коду і алгоритмом, що використовується в μ2, можна побачити на рис. 1 і рис. 2.

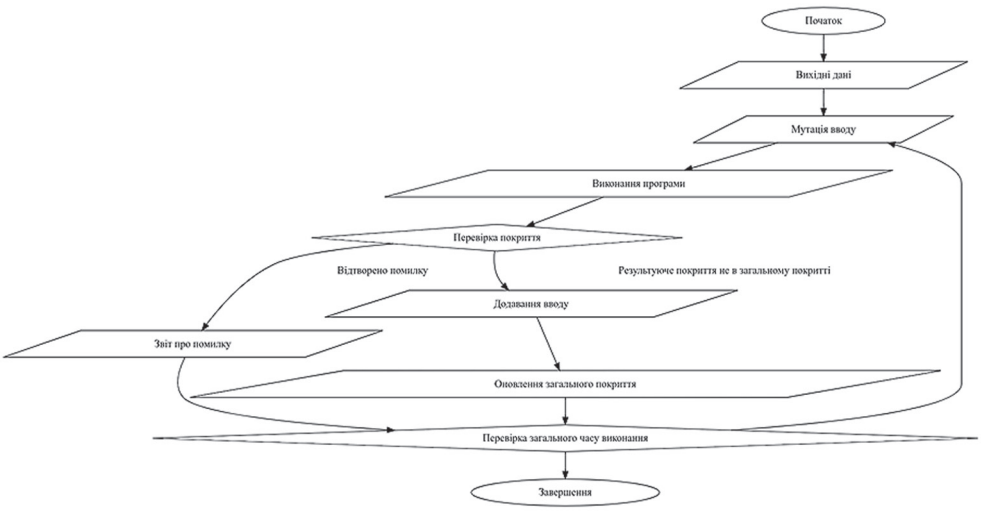

Рис. 1

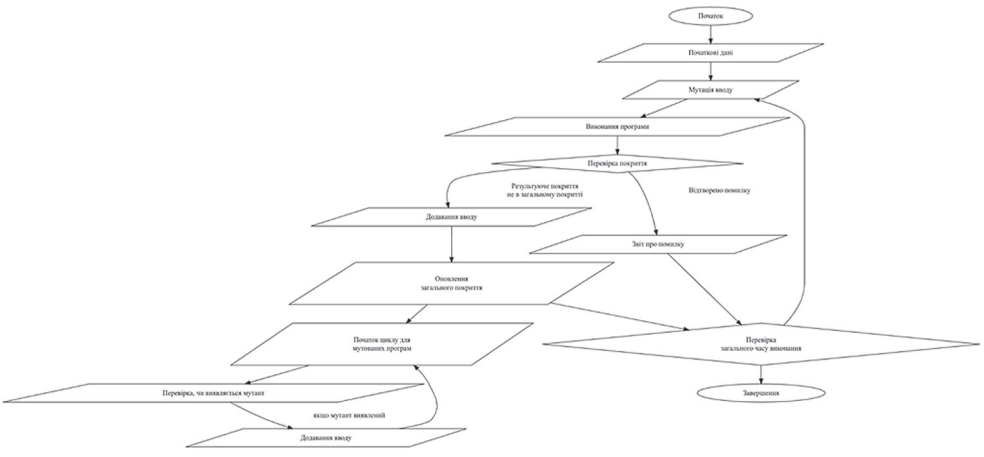

Рис. 2

Це дослідження спрямовано на порівняння ефективності двох підходів — μ2 та JQF (використовує алгоритм Zest). Залежність часу від рівня покриття коду, який досягається кожним із алгоритмів, унаочнює рис. 3. Попередні дослідження показали [6], що μ2 є більш ефективним порівняно з JQF, оскільки він регулярно відшукував більшу кількість мутантів за того самого обсягу вхідних даних.

Проте, як випливає з рис. 3, що за однакового часу виконання μ2 генерує менше тестових вхідних даних, які покривають код програми, порівняно з JQF. Це може бути пояснено тим, що JQF, заснований на алгоритмі Zest, здатний формувати семантично правильні дані, які сприяють значнішому покриттю коду. Тобто JQF генерує більшу кількість шляхів у програмі та забезпечує краще покриття.

Отже, результати підтверджують, що μ2, хоч і знаходить більше мутантів, але потребує більше часу на генерацію вхідних даних, які забезпечують повне покриття коду. Водночас JQF (Zest) показує якісніші результати, забезпечуючи вищий рівень покриття за той самий період часу.

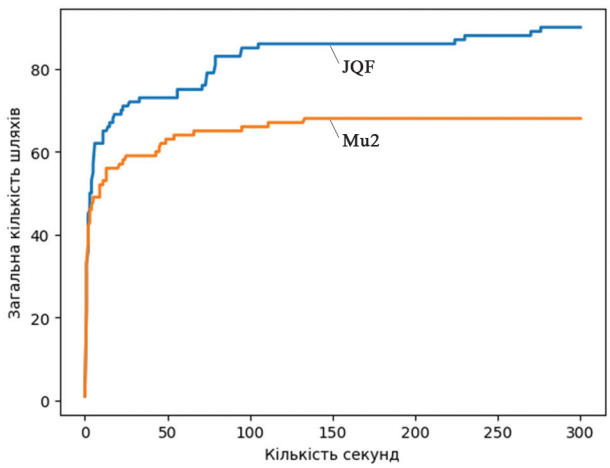

Рис. 3

Результати дослідження, в якому порівнювалася ефективність двох методів фазингу JQF і μ2 наведено в таблиці. Тут номер ітерації відповідає порядковому номеру кожного окремого експерименту. У другій і третій колонках наведено дані про загальну кількість шляхів, які було покрито впродовж п'яти хвилин завдяки генеруванню вхідних даних із використанням відповідно JQF і μ2.

Важливо зазначити, що переважна більшість загальних кількостей шляхів для μ2 менша, ніж для JQF. Це пояснює те, що μ2 потребує більше часу на генерацію вхідних даних, які забезпечують покриття коду. Отже, результати таблиці вказують на різницю в ефективності двох методів фазингу JQF і μ2 за однакового періоду часу.

Порівняння ефективності JQF і μ2-методів фазингу

Номер ітерації	JQF	Mu2
1	90	68
2	73	68
3	72	65
4	79	84
5	64	73
6	72	72
7	69	67
8	69	67
9	75	72

Висновки

Нині застосування мутаційного аналізу до фазингу потребує розв'язання низки проблем — створення й адаптація вже наявних підходів для зниження обчислювальної складності, привернення уваги з боку дослідників до цієї проблеми та створення відповідних засобів, що реалізують фазинг на основі мутаційного аналізу.

Вирішеннями для зниження обчислювальної складності може бути застосування паралельних обчислень під час мутаційного тестування, розроблення нових алгоритмів формування супермутантів, поєднання технік локалізації відмов, а особливу увагу варто приділити поєднанню фазингу з техніками локалізації відмов на основі мутацій.

Щоб знизити обчислювальну складність для оцінювання визначення повноти тестового набору, потрібно дослідити використання прогнозування як на основі машинного навчання, так і казуального моделювання.

Оскільки алгоритм роботи μ2 вираховує ефективність виявлення мутантів безпосередньо в процесі обчислення покриття коду, слід здійснити подальше дослідження, аби відшукати оптимальний баланс між покриттям шляхів і здатністю виявлення помилок. Наприклад, оцінювання ступеня загасання покриття коду [7–11] та можливість використання мутаційного аналізу для попередження цього загасання може бути перспективним напрямком для подальших досліджень.

Однак аналіз результатів також виявив, що за рівних умов μ2 забезпечує менше загального покриття коду порівняно з Zest. Це може бути пов'язано з тим, що μ2 зосереджується на виявленні помилок на глибшому семантичному рівні, тоді як Zest акцентується на підвищенні покриття шляхів, що сприяє збільшенню кількості покритих сценаріїв.

Список використаної літератури

1. ***Gopinath R., Görz Ph., Groce A.*** *Mutation analysis: Answering the fuzzing challenge. arXiv preprint // arXiv:2201.11303, 2022.*

2. **Systematic** *Assessment of Fuzzers using Mutation Analysis / Ph. Goerz, B. Mathis, K. Hassler [et al.] // 2023. Usenix Security.*

3. **Guiding** *Greybox Fuzzing with Mutation Testing / V. Vikram, I. Laybourn, Ao Li [et al.] // ISSTA 2023248.*

4. **Padhye R., Lemieux C., Sen K.** *JQF: Coverage-guided property-based testing in Java // Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis. 2019. P. 398–401.*

5. **Semantic** *fuzzing with zest / R. Padhye [et al.] // Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis. 2019. P. 329–340.*

6. **Laybourn I.** *μ2: using mutation analysis to guide mutation-based fuzzing // Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings. 2022. P. 331–333.*

7. **Reachable** *Coverage: Estimating Saturation in Fuzzing / D. Liyanage [et al.] // Proceedings of the 45th IEEE/ACM International Conference on Software Engineering (ICSE'23), 17-19 May 2023, Australia. 2023.*

8. **Investigating** *Coverage Guided Fuzzing with Mutation Testing / R. Qian [et al.] // Proceedings of the 13th Asia-Pacific Symposium on Internetware. 2022. P. 272–281.*

9. **Guidelines** *for Coverage-Based Comparisons of Non-Adequate Test Suites / M. Gligoric [et al.] // Space. 2014, 6.1,350: 1,142.*

10. **Hemmati H.** *How effective are code coverage criteria? // IEEE International Conference on Software Quality, Reliability and Security. 2015. P. 151–156.*

11. **Can this** *fault be detected: A study on fault detection via automated test generation / P. Ma [et al.] // Journal of Systems and Software. 2020. 170: 110769.*

A. P. Samoilenko, A. P. Bondarchuk

COMPARISON OF THE COMBINED FUZZING ALGORITHM BASED ON MUTATION ANALYSIS WITH THE FUZZING ALGORITHM BASED ON CODE COVERAGE

This article presents a comparative analysis between the μ2 algorithm and the Zest algorithm using the same amount of time rather than inputs and comparison by metrics other than mutation analysis. The Zest algorithm, embedded in the JQF framework, prioritizes comprehensive code coverage by generating inputs that satisfy both structural and semantic requirements. In contrast, μ2 is an approach that expands the scope of coverage-guided fuzzing through the integration of mutation analysis, which, in turn, enables the generation of higher-quality test cases.

The obtained conclusions confirm that mutation-based fuzzing requires more time for generating input data that ensures complete code coverage. The research has revealed that under equal conditions, the combined algorithm based on mutation analysis provides less overall code coverage compared to the code coverage-based algorithm.

Furthermore, it is noteworthy that the approach of fuzzing guided by mutation analysis, such as the one exemplified by the μ2 algorithm, is relatively less explored in the existing body of research. In reality, the shortage of publications dedicated to the topic of fuzzing based on mutation analysis is evident from the limited references in fuzzing review papers. This gap in awareness might contribute to the restrained popularity of mutation analysis within the security research community and in software testing. Additionally, this article aims to enhance the visibility of mutation analysis among fuzzing researchers.

Keywords: fuzz testing; mutation testing; mutation analysis; test generation.