УДК 004.05, 004.42

О.О. ЛЕТИЧЕВСЬКИЙ
Інститут кібернетики ім. В.М. Глушкова НАН України, Київ, Україна,
e-mail: *oleksandr.letychevskyi@litsoft.com.ua.*

В.С. ПЕСЧАНЕНКО
Херсонський державний університет, Херсон, Україна,
e-mail: *volodymyr.peschanenko@litsoft.com.ua.*

Я.В. ГРИНЮК
Інститут кібернетики ім. В.М. Глушкова НАН України, Київ, Україна,
e-mail: *yaroslav.hryniuk@gmail.com*.

ТЕХНІКА НЕЧІТКОГО ТЕСТУВАННЯ ТА ЇЇ ВИКОРИСТАННЯ В ЗАДАЧАХ КІБЕРБЕЗПЕКИ

Анотація. Розглянуто технологію нечіткого тестування, яка полягає у тестуванні програмних систем з поданням критичних або неочікуваних вхідних даних. Наведено огляд поточного стану проблеми та представлено основні системи нечіткого тестування. Проаналізовано підхід до технології нечіткого тестування з використанням алгебричних методів, зокрема символьного моделювання. Розглянуто алгоритм «легкої ваги», який розроблено для скорочення часу генерації тестів. Алгоритм реалізовано в середовищі системи інсерційного моделювання та апробовано в тестуванні давно відомих версій систем, розроблених в ОС Linux.

Ключові слова: нечітке тестування, вразливості в програмному забезпеченні, символьне моделювання, алгебра поведінок, інсерційні моделі.

ВСТУП

Термін «нечітке тестування» є українським аналогом терміну «fuzzing» або «fuzz testing», що активно використовується в останні десять років у процесі розроблення програмного та апаратного забезпечення. Нечітке тестування на відміну від традиційних видів полягає у створенні тестів, які без визначення цілі тестування можуть викликати збій в програмному або апаратному забезпеченні.

Якщо в традиційному тестуванні ціллю тесту є перевірка функціональності системи та відповідність між реальною поведінкою та очікуваними результатами роботи системи із заданими вхідними даними, то в техніці нечіткого тестування як вхідні дані розглядають критичні або неочікувані дані, що можуть призвести до падіння системи або її непередбаченої роботи. Подання на вхід некоректних даних — це метод визначення стійкості системи до шкідливого втручання зловмисників або захищеності від збоїв зовнішніх систем, з якими вона інтегрована.

Впродовж останніх двох десятиліть створено клас систем, що реалізує техніку нечіткого тестування, а саме системи fuzzers. Ці системи розроблялись як для тестування, так і для визначення вразливостей коду. Великих успіхів досягнуто у виявленні вразливостей нульового дня, тобто таких, які ще невідомі користувачам чи розробникам програмного забезпечення. Об'єктом нечіткого тестування може бути як окремий виконуваний файл, так і операційна система, керувальний пристрій, мережевий маршрутизатор, мобільний телефон, медичний пристрій тощо.

ОГЛЯД СУЧАСНИХ СИТЕМ НЕЧІТКОГО ТЕСТУВАННЯ

Система нечіткого тестування зазвичай містить такі складові.

— Компонента створення тестових наборів, що являє собою інтелектуальну систему, яка зможе згенерувати тестові випадки, що викликають збій або непередбачену поведінку в програмному або апаратному забезпеченні.

— Компонента виконання тестів, що інтегрована в єдину систему генерації тестових випадків та виконання тестів з можливістю запуску великої їхньої кількості.

— Компонента спостереження, відтворення та візуалізації поведінки системи під час збою, що відбувся.

За методом генерації системи нечіткого тестування поділяються на такі різновиди.

— Випадкова генерація, коли тестові сценарії генеруються за допомогою генератора випадкових чи псевдовипадкових даних. Такі системи мінімально ефективні, оскільки вхідні дані, які вони генерують для цільового програмного забезпечення, завідомо неправильні.

— Шаблонна генерація, що враховує семантику вхідних даних. Такі системи більш ефективні, ніж випадкові, але мають деякі важливі недоліки. Зокрема, ефективність нечіткості залежить від якості шаблону, який під час тестування, наприклад, протоколів має враховувати контрольну суму ідентифікатора сеансів, а це обмежує можливість тестових випадків проникнути в ціль. Якщо використовувані шаблони не охоплюють певного функціонального сценарію, то відповідної частини тестованого коду не буде досягнуто, а приховані вразливості залишаться невиявленими.

— Модельна генерація, що враховує можливі сценарії, семантику поведінки системи та формати даних. Така генерація може бути побудована на базі моделі вимог до системи або моделі самої системи.

Системи нечіткого тестування можна класифікувати з огляду на те, яка інформація потрібна для їхньої роботи на етапі тестування програми. Такою інформацією може бути покриття коду, покриття потоку даних, інтенсивність використання процесора, специфікація формату вхідних даних для генератора тестових випадків.

Системи нечіткого тестування також можна розрізняти за відкритістю інформації систем, що тестуються. Системи «чорної скриньки»: під час тестування немає ніякої інформації про програму, що тестується, крім бінарного коду; «біла скринька» — коли відомий первинний код системи та інформація з етапів проєктування системи; «сіра скринька» — інформація про первинний код відома частково.

Системи чорної скриньки будуються з урахуванням випадкових змін коректних вхідних даних для генерації нових тестових випадків. Змінами вхідних даних можуть бути інверсія випадкових бітів, копіювання або видалення байтів тощо. Такі системи можуть використовувати специфікації для формування коректних вхідних даних. Теоретично такі системи менш практичні, ніж системи із категорії білої скриньки, але, як свідчать результати, складність сучасних програмних систем настільки велика, що швидкі та більш примітивні системи чорної скриньки також здатні знаходити помилки в поведінках та мають свою цінність.

Системи із категорії білої скриньки використовують додаткову інформацію про тестовану програму. Такою інформацією може бути первинний код, відслідковування потоку виконання програми та використання тестового середовища. На відміну від систем із категорії чорної скриньки вони здатні набагато краще генерувати нові тестові випадки завдяки наявності додаткової інформації. Наприклад, під час першого прогону система збирає обмеження для даних, які спостерігаються в процесі виконання, та на основі цього генерує тестові випадки. Такі системи здатні ефективно генерувати тестові випадки, які охоплюють велику кількість можливих сценаріїв виконання програми, але складність сучас-

ного програмного забезпечення настільки велика, що розв'язування символьних рівнянь може займати занадто великий проміжок часу.

Системи з категорії сірої скриньки збирають тільки часткову інформацію про виконання програми. Для цього використовують такі техніки, як інструментування коду. Використовуючи зібрану інформацію, генератор тестових випадків може продукувати кращі шляхи виконання тестованої програми, але на відміну від систем білої скриньки немає гарантії, що зібрана інформація покращить покриття.

Системи нечіткого тестування можна також класифікувати за типами тестованих програм: програм загального призначення, ядер операційних систем, компіляторів та інтерпретаторів, драйверів і вбудованих пристроїв та мережевих протоколів.

Однією із найбільш поширених програм нечіткого тестування є система AFL (American Fuzzy Lop), створена у компанії Google. Програма має відкритий код та багато відомих програмних систем і протоколів, таких як Firefox, Open SSL, PHP, bash, які були перевірені AFL із виявленням суттєвих вразливостей. Найбільш успішне застосування AFL здійснено для виявлення вразливостей нульового дня [1].

Основні переваги AFL — це простота у використанні у поєднанні з ефективністю, аналіз файлу, що тестується відносно покриття коду та створення вхідних даних з метою максимального покриття. Програма AFL відтворює попередні кроки, щоб знайти падіння програми. Основу AFL складає генератор тестових випадків, що ґрунтується на генетичному алгоритмі. На першому кроці AFL потрібно передати вибірку коректних вхідних даних, потім використовуються кроки генетичного алгоритму такі як відбір, мутації та розмноження для генерації наступного набору тестових випадків. Помилковою поведінкою вважається зависання програми або її аварійне завершення.

Отже, система AFL є достатньо простою та ефективною для знаходження вразливостей. Але разом із тим має такі недоліки.

— Під час процедури тестування AFL намагається охопити увесь потік керування програми. Але разом з тим багато досліджень стосовно обмежування потоку керування з метою тестування лише тих частин, де вразливість може бути активована.

— Система AFL працює тільки із програмами, які приймають параметри по стандартному каналу введення/виведення, та у файлах. Тобто якщо потрібно протестувати частину програмного продукту, необхідно створити так званий fuzzing harness або програму, яка викликає функцію, що цікавить, із параметрами, які передаються на стандартний канал введення/виведення.

Відома компанія Synopsys розробила інструмент нечіткого тестування — систему SFT (Synopsys Fuzz Testing) [2], яку використовують для розроблення апаратного забезпечення та протоколів взаємодії. Особливо це важливо у разі перевірки стійкості вразливих дій протоколів Інтернету речей, де кількість пристроїв досить велика та можливі конфлікти між ними, що викликані, наприклад, зміною версій прошивок пристроїв, недостатньою якістю протоколів взаємодії тощо. Компанія Synopsys розробила критерії визначення «зрілості» програми відповідно до результатів нечіткого тестування. Одним із критеріїв є час від початку тестування до виявлення першої вразливості. Рівень зрілості системи визначають також кількістю виявлень відповідно до ступеня «серйозності» (severity). Кількість використовуваних тестів системи SFT налічує мільярди одиниць, вони застосовуються також у верифікації автомобільних систем та медичних пристроїв, фінансових програм та кіберзахисті об'єктів критичної інфраструктури.

Система Peach [3] — це система нечіткого тестування із категорії сірої скриньки. Використовується для дослідження мережевих протоколів та драйверів. Система набула поширення завдяки достатньо широким можливостям для представлення специфікації вхідних даних — так званих Peach Pits. Унаслідок цього є можливість достатньо точно формулювати правила побудови вхідних даних.

Система Syzkaller [4] — це система із категорії сірої скриньки, спеціалізована на тестуванні ядер операційних систем. Вона складається із двох компонентів: менеджера віртуальних середовищ та, власне, системи тестування. Менеджер запускає віртуальну програму із ядром операційної системи, що тестується. Потім він запускає систему у віртуальному середовищі та зберігає результати роботи. Система генерує програми із випадковим набором системних викликів та запускає їх.

Система SAGE [5] — це перша система тестування із категорії білої скриньки. Створена у дослідницькому відділі компанії Microsoft і використовується для тестування програмних продуктів Microsoft, а саме парсерів для таких різних форматів даних як jpeg, pdf, mp3 тощо. Генератор тестових випадків базується на використанні символьних обчислень для збору обмежень та генерації нових випадків. Система працює з бінарним виконуваним файлом та використовує декілька технік для покращування швидкості та використання пам'яті.

Система Csmith [6] — це система із категорії чорної скриньки. По суті, це компілятор програм мовою програмування C, який може створювати тести з урахуванням заданих граматик відповідно до стандарту C99. Система використовується для тестування компіляторів. Система Csmith довела свою ефективність, виявивши до цього невідомі помилки як у відкритих компіляторах, так і комерційних.

АЛГЕБРИЧНИЙ ПІДХІД У НЕЧІТКОМУ ТЕСТУВАННІ

В Інституті кібернетики ім. В.М. Глушкова (Київ) розробляють підхід, який використовує модель бінарного коду тестованої системи для генерації тестових випадків. При цьому використовують алгебричний підхід та техніку символьного моделювання.

Процес нечіткого тестування — це генерація тестових випадків за деякою підказкою, яка сформована на основі аналізу моделі тестованої програми. Така техніка реалізує моделювання «легкої ваги» (lightweight) на відміну від повного символьного моделювання, що використовується у виявленні вразливостей у програних системах [7]. Завдяки цьому матимемо такі переваги:

1) пошук підказки, на основі якої будується тест, не застосовує пошуку у повному просторі всіх станів програми, який призводить до експоненціального вибуху;

2) абстрагування від точної семантики вразливостей до їхніх класів дає можливість виявити так звані вразливості нульового дня або ті, що були невідомі раніше.

Весь процес нечіткого тестування складають такі етапи.

— Створення алгебричної моделі. Бінарний код системи дизасемблюється та транслюється в мову алгебри поведінок.

— Визначення вразливостей, що виявляються. Розглядається множина ймовірних вразливостей, що можуть призвести до збою або неочікуваної поведінки, наприклад переповнення буфера, запис поза межами структур, помилки в математичних операціях. Шаблони вразливостей записуються у мові алгебри поведінок і являють собою узагальнений опис вразливості на рівні машинних команд.

— Виявлення потенційних до вразливостей точок у моделі. Алгоритми алгебричного зіставлення використовують для пошуку місць, де можливе спрацювання вразливостей. Для цього застосовують техніку «легковагового» (light-weight) символьного моделювання.

— Створення тесту. Тестовий випадок генерується як можливі вхідні дані впродовж шляху (траси), що веде від точки введення даних до точки вразливостей. Тут тест — це множина значень, що вводяться із зовнішнього середовища впродовж траси як такі, що можуть активізувати вразливість.

Розглянемо процес нечіткого тестування в деталях.

Створення алгебричної моделі. Модель бінарного коду представляють за допомогою рівнянь алгебри поведінок, яка містить мову дій [8]. Після дизасемблювання отримуємо код в одній із мов процесора, зокрема INTEL x86, який розглядаємо як приклад. Код складається із множини машинних інструкцій I, які розпаровуються на інструкції потоку керування та інструкції роботи із даними (скорочено — інструкцій даних).

Кожна інструкція даних — це трійка: $I = (M, D, S, A)$, де M — назва інструкції, D — компонента «ціль», S — компонента «джерело», A — компонента «додаткове джерело». Інструкція може містити або всі компоненти, або їхню частину, або бути порожньою. Інструкція потоку керування містить адресу сегменту коду або реалізує перехід по сегменту згідно з результатами інструкцій даних.

Кожна компонента являє собою елементи середовища процесора, у цьому випадку — швидка пам'ять або регістри та основна пам'ять. Кожне джерело має свій відповідний обсяг, що визначається кількістю байтів. У мові машинних інструкцій розглядають регістри однобайтові, величиною зі слово, подвійне слово та більших розмірів, що дорівнюють степені двійки. Пам'ять — це рядок певної кількості байтів *Length* (байт, слово, подвійне слово і т.д.), що записуються із певної адреси пам'яті *Addr*, позначимо його *Mem* (*Addr*, *Length*).

Кожна інструкція даних відображає операцію копіювання компоненти «джерело» в компоненту «ціль». Таке копіювання здійснюється за умови виконання арифметичних, логічних або інших операцій для відповідних даних.

Отже, для нечіткого тестування інструкції перетворюються в поведінкові специфікації. У роботі [9] описано процес перекладу асемблерної програми в специфікації алгебри поведінок для точного виявлення вразливості згідно з їхніми шаблонами. Оскільки нечітке тестування визначає генерацію великої кількості тестів, то в системі, що розглядається, генеруються лише підказки на місця, які можливо навантажувати непередбачуваними або критичними даними. Відповідно час на налаштування таких підказок має бути значно менший, ніж час на точне виявлення вразливостей. Таким чином, модель має бути підготовлена для швидкого виявлення потенційних місць, де спрацьовує вразливість.

Розглянемо приклад перетворення дизасембльованих інструкцій у мову алгебри поведінок. Маємо таку програму:

```
mov ebx, 0x100
mov ax, 0x10
mov cx, ax
mov ecx, ebx
mov eax, 0x3c
```

Цей фрагмент програми являє собою просте пересилання значень між регістрами. В алгебричному вигляді це безумовні копіювання порцій пам'яті за

допомогою операції копіювання байтів. Отримуємо послідовність дій, що мають як параметри адресу призначення та адресу джерела. Усі копіювання трансформуються на переписування регістра rax, що містить 8 байт або 64 біти — максимально можливий обсяг у цій архітектурі:

mov(0,ebx,0x100) = ((1) ->(" ")(rbx(0,4) = 0x0100;rip = 0x401005)),
mov(1,ax,0x10) = ((1) ->(" ")(rax(0,2) = 0x10;rip = 0x401009)),
mov(2,cx,ax) = ((1) ->(" ")(rcx(0,2) = rax(0,2);rip = 0x40100c)),
mov(3,ecx,ebx) = ((1) ->(" ")(rcx(0,4) = rbx(0,4);rip = 0x40100e)),
mov(4,eax,0x3c) = ((1) ->(" ")(rax(0,4) = 0x3c;rip = 0x401013)),

Регістр rip вказує на адресу наступної інструкції. Він формує поведінку, що визначає послідовність дій, кожна з яких має свій числовий номер:

B0 = mov(0).B401005,
B401005 = mov(1).B401009,
B401009 = mov(2).B40100c,
B40100c = mov(3).B40100e,
B40100e = mov(4).B401013,
B401013 = …

Визначення класу вразливостей, що виявляються. На відміну від точного формального визначення вразливостей, що потребує значного більшого часу та може супроводжуватись комбінаторним вибухом станів системи під час пошуку вразливої поведінки, клас вразливостей в нечіткому тестуванні можна визначити деякою формулою. Розглянемо відому вразливість «Переповнення буфера», що полягає в записі у пам'ять поза дозволеною зоною. Такими вразливостями є запис поза зоною масиву, поза зоною виділеної пам'яті та в загальних випадках можемо розглядати запис поза зоною стекової пам'яті.

Нехай *addr* — деяка адреса автоматичної (стекової) пам'яті, за якою відбувається запис відповідно до інструкції. Вразливість спрацьовує, якщо запис відбувається поза межами відповідної пам'яті, тобто умови спрацювання можна виразити нерівністю

$$addr >= BP \,||\, addr <= SP \,\&\,\& \, addr > SS.$$

Ця нерівність означає, що ми пишемо поза межами стекової пам'яті, що визначається межами *BP* (базова адреса стеку) та *SP* (вказівник стеку). Додатково визначимо межу стекового сегмента *SS*.

Виявлення потенційних точок спрацювання вразливостей. Ця процедура визначає досяжні точки спрацювання вразливостей від точок введення даних із зовнішнього середовища. У загальному випадку — це виявлення пар (I, V), де V — інструкція спрацювання вразливості, I — точка введення даних.

Точками введення даних слугують інструкції із системними викликами *syscall*. Кожен такий системний виклик може виконувати введення даних із командного рядка, файлу, клавіатури або сокета. Відповідні дані розміщуватимуться в пам'яті, адресу та розмір якої зафіксовано в регістрах загального призначення.

Точка спрацювання вразливості являє собою запис у пам'ять, яка знаходиться поза межами стеку або виділеної динамічної пам'яті, що може призвести до аварійного збою або можливості атаки із перехопленням керування.

Побудова тесту. Для кожної пари (I, V) можна довести, що інструкція V досяжна з точки введення даних I та умова спрацювання вразливості здійсненна.

Це можна довести символьним моделюванням, яке детально розглянуто в [10]. Якщо ввести предикат, що визначатиме пам'ять, яка була введена із зовнішнього середовища через $Dirty(x)$, то початковий стан після виконання інструкції I буде таким:

$$E0 = Forall\,(j : int)\,(0 <= j < N)\,Dirty(addr + j),$$

де $addr$ — адреса введених даних, N — їхній розмір. Вважатимемо також, що для всіх інших адрес x предикат $Dirty$ не є істинним, тобто $!Dirty(x)$.

Якщо семантика інструкцій представлена за допомогою мови дій в алгебрі поведінок, то в символьному моделюванні для переходу в інший стан середовища після виконання інструкції спочатку перевіряється здійсненність формули C & $E0$, де C — передумова, а $E0$ — поточний стан середовища. Якщо формула здійсненна, то можна перейти в інший стан $E1$ із виконанням функції pt предикатного перетворювача [11]:

$$E1 = pt(C \& E0, R),\ \text{де}\ R\ —\ \text{післяумова інструкції}.$$

Отже, якщо існує сценарій поведінки програми $E0, E1, \ldots$, що представляє послідовність виконання програми до інструкції V, то нехай EN — стан, який буде мати місце після виконання інструкції V. Тоді вразливість спрацьовує, якщо формула

$$EN\,(addr)\,\&\,\&\,(addr >= BP\,||\,addr <= SP\,\&\,\&\,addr > SS)\,\&\,\&\,Dirty(addr)$$

є здійсненною. Тоді оберненим символьним моделюванням можна отримати експлойт, тобто дані, які треба подати на вхід програми у місці системного виклику, щоб спрацювала вразливість. У той же час можна згенерувати тест, що відповідає спрацюванню вразливості. Цей спосіб може бути неефективним, якщо складність обчислень під час моделювання досить висока. До того ж велика кількість сценаріїв та інструкцій можуть призвести до довготривалого часу виконання, хоча такий спосіб гарантує точне виявлення вразливості.

Зауважимо, що в адресу, в яку записують дані в точці спрацювання вразливості, можуть потрапити дані, які були введені під час системного виклику. Отже, спрощенням розглядуваного способу виявлення вразливості є ігнорування інструкцій, що не залежать від даних, які впливають на адресу. Зокрема, спрощуються довгочасні операції символьних розв'язувачів.

У нечіткому тестуванні символьне моделювання може бути послаблене щодо здійсненності передумови інструкції та виконання предикатного перетворювача. Опишемо техніку спрощеного моделювання, що має відповідний термін в літературі, а саме техніку «легкої ваги» (light weight modelling).

Моделювання методом «легкої ваги». Цей метод має декілька модифікацій залежно від використання програм-розв'язувачів та машин доведень. Метод мінімально використовує такі ресурси, але водночас зменшує точність підказки для тесту. Тому для вибору способу нечіткого тестування треба мати на увазі баланс між часом виконання та точністю виявлень помилок та аварійних завершень програми.

Точкою введення будемо вважати точку введення даних за допомогою інструкції системного виклику *syscall*, що зчитує дані із файлу. Адреса зчитаних даних міститься в індексному регістрі SI, а розмір пам'яті — в регістрі DX.

Як точку вразливості розглядаємо запис у стекову пам'ять. Це може бути інструкція $mov\,(N\,[BP - x], Y)$, де N — розмір запису, BP — базова адреса стеку, Y — будь-яке джерело, з якого відбувається запис. У цій інструкції як джерело розглядаємо регістр загального призначення.

Створимо список елементів середовища, що містить пам'ять, яка може бути критичною. На початку списку розміщена множина адрес: $SI, SI+1, SI+2, \ldots$ $\ldots, SI+M$, де M — розмір пам'яті з регістру DX. Цей список назвемо списком «брудної» пам'яті.

Для моделювання «легкої ваги» від точки системного виклику ми розглядаємо тільки інструкції копіювання, де фігурує пам'ять зі списку. Водночас мають виконуватись правила:

1) якщо «брудна» пам'ять міститься у компоненті джерела в інструкції, а це, зазвичай, другий або третій параметр інстукції, то цю компоненту фіксуємо також у списку «брудної» пам'яті;

2) якщо «брудна» пам'ять міститься у компоненті призначення, а це, зазвичай, перший параметр інструкції, то її можна вилучити із списку «брудної» пам'яті;

3) якщо «брудна» пам'ять фігурує і у компоненті призначення, і в джерелі, то ніяких дій не передбачається.

У процедурі вилучення виникає проблема — як зіставляти пам'ять з моделі, якщо вона невідома, із пам'яттю в списку. Коли «брудною» пам'яттю є регістр, то ця проблема розв'язується однозначно. Якщо регістр є частиною іншого регістру, то треба розглянути та залишити або вилучити зі списку ту частину регістру, в якій ще зберігається пам'ять. Для цієї операції не передбачено застосування машин-розв'язувачів.

Інша справа із самою пам'яттю. Розглянемо два способи моделювання.

Перший спосіб є грубим. Для нього передбачається, що вся невідома пам'ять є «брудною», і в цьому разі не потрібно ніяких обчислень, але більшість тестів не спрацьовує.

Другий спосіб полягає у слідкуванні за величинами адрес із застосуванням «конколічних» обчислень, які мінімально використовують символьне моделювання, та в місцях, де відома семантика розміщення даних після виконання системних викликів, підставляють деяке довільне значення адреси пам'яті. Ця адреса може бути віртуальною, оскільки після спрацювання одного системного виклику пам'ять, що призначена для запису зчитаних даних, може не використовуватись для інших інструкцій. У цьому випадку відсоток невідомої (символьної) пам'яті, в яку записується або з якої вилучається «брудна» пам'ять, може бути доволі малим. Цей факт впливає на швидкість генерації підказок.

Для моделювання відрізку від однієї точки до іншої можуть бути задані за сценарієм декілька точок введення, що є природним для реактивних систем або у разі тестування інтерфейсу користувача. Кількість «брудної» пам'яті в списку збільшується.

ВИСНОВКИ

Перевагою запропонованого методу є те, що під час створення тестів ми розглядаємо лише ті сценарії поведінки, що призводять до вразливості, тим самим збільшуючи покриття множини станів програми, де може трапитись збій або знайтись вразливість. На відміну від програми AFL, що відслідковує повне покриття потоку керування, ми додаємо контроль за ймовірними сценаріями, що зумовлять знаходження вразливості. Отже, значно скорочується загальний простір пошуку.

Прототип програми нечіткого тестування було створено на основі системи інсерційного моделювання [12] та перевірено на прикладах застарілих версій програм у системі Linux, таких як Midnight Commander та текстовий редактор vi. Тести

генерувалися лише для тих пар інструкцій, які передавали дані із системного виклику до запису в критичну зону стеку, що спричинила його пошкодження. Застосовано «конколічний» спосіб моделювання. Для виявлення такої пари тест видавався в різних поданнях критичних значень — максимальних та мінімальних значень у байтах, рядках надвеликої довжини, яка розраховувалась згідно з формулою значень адрес, за якими здійснювався запис у зоні стеку.

СПИСОК ЛІТЕРАТУРИ

1. American Fuzzy Lop. URL: https://lcamtuf.blogspot.com/2014/10/bash-bug-how-we-finally-cracked.html.
2. Synopsis. URL: https://www.synopsys.com/software-integrity/security-testing/fuzz-testing.html.
3. Peach. URL: https://medium.com/csg-govtech/lifes-a-peach-fuzzer-how-to-build-and-use-gitlab-s-open-source-protocol-fuzzer-fd78c9caf05e.
4. Syzkaller. URL: https://github.com/google/syzkaller/blob/master/docs/research.md.
5. SAGE. URL: https://queue.acm.org/detail.cfm?id=2094081.
6. Csmith. URL: https://srg.doc.ic.ac.uk/files/papers/compilerbugs-oopsla-19.pdf.
7. Летичевський О.О., Гринюк Я.В., Яковлев В.М. Алгебраїчний підхід у формалізації вразливостей в бінарному коді. *Control Systems and Computers*. 2019. № 6. С. 5–20.
8. Letichevsky A. Algebra of behavior transformations and its applications. *Structural Theory of Automata, Semigroups, and Universal Algebra. NATO Science Series II. Mathematics, Physics and Chemistry*. Kudryavtsev V.B., Rosenberg I.G. (Eds.). 2005. Vol. 207. P. 241–272.
9. Letychevskyi O., Peschanenko V., Radchenko V., Hryniuk Y., Yakovlev V. Algebraic patterns of vulnerabilities in binary code. *Conference Proceedings of 2019 10th International Conference on Dependable Systems, Services and Technologies (DESSERT'2019)* (June 5–7, 2019, Leeds, United Kingdom). IEEE, 2019. P. 70–73.
10. Потиенко С.В. Методы прямого и обратного символьного моделирования систем, заданных базовыми протоколами. *Проблеми програмування*. 2008. № 4. С. 39–45.
11. Летичевский А.Ад., Летичевский А.А., Годлевский А.Б., Песчаненко В.С., Потиенко С.В. Свойства предикатного трансформера системы VRS. *Кибернетика и системный анализ*. 2010. № 4. С. 3–16.
12. Letichevsky A., Letychevskyi O., Peschanenko V. Insertion modeling and its applications. *Computer Science Journal of Moldova*. 2016. Vol. 24, Iss. 3. P. 357–370.

O.O. Letychevskyi, V.S. Peschanenko, Y.V. Hryniuk

FUZZING TECHNIQUE AND ITS USAGE IN CYBERSECURITY TASKS

Abstract. The paper considers the technology of fuzzy testing, which involves testing software systems with the operating of critical or unexpected input data. An overview of the current state of the problem is made and the main systems of fuzzy testing are presented. The approach to the technology of fuzzy testing with the use of algebraic methods, in particular symbolic modeling, is considered. The "light weight" algorithm, which is designed to reduce the generation time of tests, is considered. The algorithm is implemented in the environment of the insertion modeling system and applied in testing older versions of systems developed in Linux.

Keywords: fuzzing, vulnerabily of software, symbolic modeling, behavior algebra, insertion model.

Надійшла до редакції 22.09.2021