УДК 004.056.5:004.9

[1,2]Н.О.Маслова, к.т.н., доц.
[1]Ю.І. Шоломинський, студент
[1]Львівський державний університетет
безпеки життєдіяльності, Львів, Україна,
[2]Донецький національний
технічний університет, Дрогобич, Україна
[1]masgpp2@gmail.com
[1]nataliia.maslova@donntu.edu.ua
[2]solompaber@gmail.com

Інтеграція комп'ютерної логіки у DevSecOps для перевірки безпеки

У статті розглянуто підхід до інтеграції комп'ютерної логіки у процес DevSecOps як засобу підвищення ефективності автоматизованої перевірки безпеки. Проведено аналіз сучасних методів застосування формальних моделей, символьного виконання та SMT-розв'язувачів, зокрема Z3, у CI/CD-конвеєрах для перевірки політик безпеки, логічної узгодженості компонентів і виявлення вразливостей. Запропоновано модель включення логічного аналізатора у пайплайн розробки з поетапним контролем політик доступу за допомогою Open Policy Agent (OPA) та Rego. Наведено приклад практичної реалізації інтеграції у вигляді правильного та помилкового сценаріїв застосування логічних умов у DevSecOps середовищі. Окремо розглянуто переваги поєднання формальних методів і гнучких DevOps-практик, що забезпечують проактивне виявлення помилок безпеки ще на етапі розробки. Результати дослідження можуть бути використані для створення інтелектуальних систем безпеки, здатних до логічного аналізу конфігурацій, транзакцій та політик доступу в автоматизованому режимі.

Ключові слова: DevSecOps, комп'ютерна логіка, policy-as-code, SMT-розв'язувач, Z3, формальні методи, CI/CD-пайплайн, перевірка безпеки

DOI: 10.31474/1996-1588-2025-2-41-140-149

Вступ

У сучасних веб-додатках, а особливо у фінансових або корпоративних системах, критичне значення має правильна реалізація політик доступу (policy-as-code). Одним із ефективних підходів є інтеграція формальної перевірки логіки доступу у CI/CD пайплайн (CI/CD pipeline**)** – автоматизованої послідовності етапів (кроків, процесів), через які проходить програмне забезпечення від моменту написання коду до його розгортання (deployment) у робочому середовищі DevSecOps.

Актуальність інтеграції комп'ютерної логіки в DevSecOps. Сучасні системи розробки програмного забезпечення характеризуються високим рівнем автоматизації, проте саме автоматизація часто стає джерелом ризиків безпеки. Помилки у політиках доступу, некоректні перевірки прав користувачів або неправильна інтеграція модулів можуть призвести до критичних вразливостей.

DevSecOps підходить до безпеки як до невід'ємної частини життєвого циклу розробки, але класичні методи тестування не завжди здатні виявити логічні суперечності у правилах доступу чи взаємодії компонентів. Тут свою роль відіграє комп'ютерна логіка, яка дозволяє формалізувати правила безпеки у вигляді булевих виразів і перевіряти їх істинність автоматично.

Інтеграція логічних перевірок у DevSecOps-пайплайн (ланцюг автоматизованих процесів інтеграції, тестування та доставки) створює додатковий рівень контролю, що забезпечує:

- раннє виявлення логічних конфліктів у політиках доступу;
- підвищення довіри до автоматизованих процесів розгортання;
- зниження ризику людських помилок у конфігураціях безпеки.

Інтеграція методів комп'ютерної логіки у DevSecOps-процеси дає змогу формалізувати політики безпеки як набір логічних тверджень і перевіряти їх істинність за допомогою алгоритмів автоматичної верифікації. Це створює новий рівень прозорості та передбачуваності, де безпека стає не лише практикою, а й формально доведеною властивістю системи.

Таким чином, використання комп'ютерної логіки у DevSecOps не лише автоматизує верифікацію політик доступу, але й робить безпеку системи перевіреною, відтворюваною та формально обґрунтованою.

Мета дослідження: показати, як інтеграція комп'ютерної логіки у DevSecOps-процеси дозволяє автоматизувати перевірку безпеки, зокрема виявлення логічних суперечностей у

політиках доступу та управлінні правами користувачів.

Завдання дослідження:

1) проаналізувати роль формальної логіки у процесах DevSecOps;
2) розробити модель інтеграції логічного аналізатора у CI/CD-конвеєр;
3) навести приклади застосування логічних перевірок на основі політик доступу;
4) порівняти правильний та неправильний сценарії налаштування доступів;
5) визначити переваги використання логічних методів для забезпечення безпеки в автоматизованих середовищах.

Огляд DevSecOps і ролі логіки

DevSecOps – підхід, який інтегрує безпеку як невід'ємну частину DevOps-процесу: від планування й розробки до тестування, розгортання й моніторингу. Мета – зрушити вправо (shift-left) перевірки безпеки, перетворивши їх зі ступеня контролю після релізу на автоматизовані ґейт-чекі у CI/CD. Практики DevSecOps включають політики доступу, автоматичні сканери, інтегровані тести безпеки та моніторинг у продуктиві [1,2].

Класичні тести та лінтери виявляють технічні дефекти, але рідко виявляють логічні суперечності в політиках доступу або умовах взаємодії компонентів (наприклад: дві політики, які в сумі дають неочікувані права).

Комп'ютерна логіка дозволяє формалізувати політики доступу як логічні формули і математично довести (або спростувати) їх властивості: відсутність суперечностей, неможливість досягнення небажаного стану, збереження інваріантів. Ця формальна перевірка робить безпеку відтворюваною та придатною для автоматизації в пайплайні [3-5].

Основні логічні підходи та техніки застосовні в DevSecOps це декларативні правила, створювані Policy-as-code з врахуванням движків політик та SMT / SAT-перевірки.

Політики доступу оформлюються як код (Rego/OPA, Gatekeeper, Conftest) і запускаються у Continuous Integration (CI) – безперервній інтеграції змін у коді для блокування небезпечних змін (інфраструктури, конфігурацій, дозволів) до розгортання. Такий підхід дозволяє писати тести політик, версувати їх і виконувати у пайплайні [6].

SMT / SAT-перевірка (Z3, cvc5 тощо), SMT-солвери (Satisfiability Modulo Theories) використовуються для перевірки логічних тверджень про код або моделі (наприклад, чи існує конфігурація, яка дозволяє обійти політику). Інтеграція SMT у CI дозволяє автоматично шукати контр приклади до специфікацій [2].

Для систем із конкуренцією або часовими властивостями (порядок транзакцій, протоколи автентифікації) застосовують модельні перевірники і темпоральні логіки (LTL, CTL, TLA+). Вони дозволяють формально доказати властивості типу «завжди не відбудеться X» (safety) або «зрештою відбудеться Y» (liveness). Застосовні інструменти – NuSMV, SPIN, TLA+ (TLC, Apalache) [7-8].

Для верифікації поведінки програм і смарт-контрактів використовують символьне або конколік (символічне + конкретне) виконання, що генерує умови (path constraints), вирішення яких відбувається за допомогою SMT-солверів. Це ефективно для коротких, детермінованих програм (наприклад, smart-contracts) [9].

Формальні методи поєднують з ML-модулями для пріоритизація виявлених проблем, генерації тестів або автоматичної класифікації попереджень, з метою підвищити масштабованість та зменшити шум у сповіщеннях. Розвивається напрямок «формальна логіка + AI» для автоматичного виявлення патернів вразливостей [10].

Останні дослідження у сфері символічного виконання програмного забезпечення демонструють значний потенціал для інтеграції методів комп'ютерної логіки в DevSecOps-процеси з метою підвищення безпеки.

Зокрема, [11] та [12] надають комплексні огляди застосувань символічного виконання у виявленні вразливостей, аналізі шкідливого ПЗ, прошивок та протоколів, що підкреслює можливість його використання для автоматизованого тестування та забезпечення безпечної розробки.

Інструменти та набір взаємопов'язаних технологій, мов програмування, фреймворків, бібліотек та інструментів (стек), які використовуються разом для створення або підтримки системи /стек, які використовують у практиці DevSecOps:

1) Open Policy Agent (OPA) / Rego – політики як код для CI/CD, Kubernetes і сервіс-шлюзів, підтримує тестування політик і запуск у пайплайні [6];
2) SMT / Theorem provers (Microsoft Z3, cvc5) – для перевірки властивостей програм, криптопротоколів, аналізу ризиків [2];
3) Model checkers (NuSMV, SPIN, TLA+ tools: TLC/Apalache) – для аналізу протоколів, порядок-часових властивостей і виявлення дедлоків/державних станів [7];
4) Symbolic execution tools (Mythril для EVM) – автоматичний аналіз смарт-контрактів [9];
5) CI/CD інтеграції та практики – GitLab/Jenkins/Azure Pipelines інтегрують policy-checks, тестування і валідацію IaC (інфраструктури як код) перед розгортанням.

Документи та посібники (NIST, DoD) описують шаблони вбудування таких перевірок у пайплайн [13].

Наведемо приклади практичних патернів (шаблони інтеграції у CI/CD):

- перевірки політик перед злиттям – запуск OPA/Rego або unit-тестів політик при створенні Pull Request, злиття блокується, якщо є невідповідності [14];

- перевірки на контрольному етапі SMT/SAT – на кожному етапі релізу створюються формальні моделі (зі специфікацій або конфігурацій), які перевіряються через Z3 або NuSMV, якщо знаходиться помилка, пайплайн зупиняється [2];

- безперервна перевірка конфігурацій і інфраструктури, як коду – перевірка файлів Terraform, Helm та Manifest за політиками в коді (Conftest + OPA) перед їх застосуванням [15];

- моніторинг після розгортання та формальні перевірки – після релізу система контролює ключові властивості в продуктиві (знімки стану, перевірка властивостей) і при порушеннях виправляє або відкочується до попередньої версії [16].

Переваги та обмеження застосування формальної логіки в DevSecOps

Переваги застосування формальної логіки в DevSecOps:

- виявлення логічних вразливостей, які пропускають звичайні тести [2] ;

- автоматизація контролю політик («policy-as-code») і відтворюваність перевірок [14];

- формальні гарантії для критичних модулів (платежі, автентифікація, smart-контракти) [17].

Обмеження / виклики:

- масштабованість і state-explosion: модельні перевірки та символьний аналіз страждають від експоненціального росту станів для великих систем; потрібні абстракції і евристики [7];

- витрати на формалізацію: якісна специфікація – це трудомісткий процес; без «чистих» специфікацій інструменти малоефективні [8];

- шум у повідомленнях / хибні спрацьовування: щоб система працювала ефективно, формальні перевірки треба поєднувати з відбором важливих сповіщень (triage) та фільтрацією за допомогою ML [10].

Дослідження останніх років підкреслюють ефективність інтеграції комп'ютерної логіки у DevSecOps-процеси для забезпечення безпеки програмних систем.

Так, [18] пропонують методи та моделі автоматизованого пошуку вразливостей у веб-додатках, що демонструє практичну реалізацію безпечного тестування.

Автор роботи [19] акцентує на використанні DevSecOps для аналізу сучасних загроз інформаційної безпеки, тоді як [20] проводить порівняльний аналіз методів формальної верифікації для систем критичного призначення, що забезпечує надійність і точність інтегрованих рішень.

Тож, літературні джерела надають наступні рекомендації для практичної реалізації процесу інтеграції формальних методів та логічної перевірки в DevSecOps-пайплайн:

а) починати з політик (OPA/Rego) для найчастіших і критичних правил; версіонувати і тестувати політики [6];

б) для ключових компонентів (автентифікація, транзакції, smart-контракти) застосувати символьний аналіз, SMT-розв'язувачі або model checking за межами основних юніт-тестів [2], для перевірки прихованих логічних помилок, станів гонок, недетермінізму чи порушення безпекових інваріантів;

в) вбудувати формальні перевірки у пайплайн у ролі контрольного бар'єра (до злиття, до розгортання) та підключити автоматичні сповіщення і приглушення (інтеграції: Slack, Jira)) [14].

В рекомендації «а)» логічні моделі визначають інваріанти безпеки у вигляді правил доступу, сумісності конфігурацій або залежностей компонентів, що дає змогу автоматично виявляти порушення політик ще до виконання коду;

За рекомендацією «б)» Z3 та подібні інструменти дозволяють формально довести, що певні умови завжди істинні, або ж небезпечні стани недосяжні.

На етапі вбудовування формальних перевірок логічна перевірка виконує роль контрольного фільтра, який блокує злиття коду, якщо логічні умови безпеки порушено, тобто стає частиною автоматизованого механізму допуску змін до продакшн-середовища, під яким розуміємо робоче середовище, в якому програмне забезпечення або система фактично використовуються кінцевими користувачами для виконання реальних завдань.

Іншими словами, це «живе» середовище, де програма працює у повному обсязі, обробляє реальні дані та має критичне значення для бізнесу або процесів.

DevSecOps як безперервна інтеграція, тестування, доставка й безпека

DevSecOps – це підхід до розробки програмного забезпечення, який інтегрує безпеку (Security) в кожну фазу життєвого циклу DevOps (див. таблицю 1).

Основна ідея: безпека не додається після розробки, а вбудовується з самого початку – у

процеси CI/CD (Continuous Integration / Continuous Delivery).

Таблиця 1 – Етапи життєвого циклу DevOps

	Етап	Основні дії
1	Continuous Integration (CI)	Компіляція, юніт-тести, збірка артефактів
2	Continuous Testing (CT)	Автоматичне тестування якості, функціональності, безпеки
3	Continuous Delivery (CD)	Автоматичне розгортання на тестових або продакшн-середовищах
4	Continuous Security	Постійний моніторинг, верифікація політик, контроль доступу

Інструменти, які застосовуються на цих етапах та їх значення для безпеки наведено в таблиці 2.

Таблиця 2 – Інструментарій для DevSecOps

	Приклади інструментів	Внесок у безпеку
1	Jenkins, GitLab CI	Аналіз вразливостей у коді (SAST, Linting)
2	SonarQube, OWASP ZAP	Виявлення помилок і порушень політик
3	ArgoCD, Spinnaker	Перевірка відповідності конфігурацій безпеки
4	OPA (Open Policy Agent), HashiCorp Sentinel	Забезпечення узгодженості дій із політиками безпеки

У DevSecOps часто застосовують формальні логічні моделі для автоматичного аналізу та перевірки політик доступу, авторизації чи конфігурації.

Комп'ютерна логіка традиційно базується на булевій логіці, де змінні приймають значення true або false. Булева логіка дозволяє формалізувати поведінку програм і протоколів у вигляді логічних виразів.

Предикатна логіка враховує властивості об'єктів і квантори (∀, ∃). Приклад її використання: за умовою: ∀u (isManager(u) → canApprove(u)) – усі менеджери мають право затвердження

Логіка політик (Policy Logic) – як поєднання булевої, предикатної та модальної логіки. Вона використовується в системах OPA/Rego для формальної перевірки правил доступу

Перевагами логічної перевірки політик є :

- мінімізація людських помилок;
- формальна доказовість коректності;
- можливість автоматичної перевірки за допомогою політик.

Наведемо приклад: автоматичної перевірки логічних умов. Наприклад, задано політику:

IF (isAdmin → canManageUsers) ∧ (¬isManager → ¬canModifyAccounts)

Це означає, що якщо користувач є адміністратором, він повинен мати право керування користувачами. Якщо користувач не є менеджером, він не повинен мати права змінювати акаунти (табл.3).

Таблиця 3 - Таблиця істинності для першої частини політики

isAdmin	Can ManageUsers	Результат (→)
0	0	1
0	1	1
1	0	0
1	1	1

Результат - істина у всіх випадках, крім коли адміністратор не має права керування користувачами.

Таблиця істинності для другої частини політики (¬isManager → ¬canModifyAccounts) – наведена в таблиці 4:

Таблиця 4 – Таблиця істинності для другої частини політики

Is Manager	Can Modify Accounts	¬is Manager	¬can Modify Accounts	Результат (→)
0	0	1	1	1
0	1	1	0	0
1	0	0	1	1
1	1	0	0	1

Результат хибний лише тоді, коли не-менеджер має право змінювати акаунти.

Таблиця 5 демонструє комбіновану умову: (isAdmin → canManageUsers) ∧ (¬isManager → ¬canModifyAccounts)

Таблиця 5 – Таблиця істинності для комбінованої умови

Is Admin	Can Manage Users	is Manager	Can Modify Accounts	Результат
0	0	0	0	1
0	1	0	0	1
1	1	1	1	1
1	0	0	1	0
0	1	0	1	0
1	0	1	0	0

Отже, політика виконується лише тоді, коли адміністратор має право керування, а не-менеджер не має права змінювати акаунти

Rego/OPA-приклад автоматизації наведено на рис. 1.

```
package access
default allow = false
allow {
  input.isAdmin
  input.canManageUsers
}
deny {
  not input.isManager
  input.canModifyAccounts
}
```

Рисунок 1 – Rego/OPA-приклад

Policy-as-code автоматично перевірить вхідні дані користувача й визначить, чи відповідає він заданим правилам.

Приклад показує, що формальна логіка в DevSecOps дозволяє описувати політики доступу верифіковано та відтворювано, що робить процес безпеки частиною безперервного циклу CI/CD, а не окремим етапом.

Модель інтеграції логічного аналізатора у CI/CD-конвеєр

Загальна концепція моделі (таблиця 6).

Мета: вбудувати логічний аналізатор у конвеєр CI/CD для автоматичної верифікації політик безпеки, правил доступу, та умов конфігурації.

Ідея: логічний аналізатор перевіряє формальні умови типу

IF (isAdmin → canManageUsers)

до або перед розгортанням системи.

Таблиця 6 – Структура моделі інтеграції

Етап CI/CD	Опис дії
1. Commit & Lint	Розробник комітить код і політики у репозиторій
2. Static Analysis (SAST)	Аналіз вихідного коду на вразливості
3. Logic Verification Stage	Формальний логічний аналіз політик
4. Test & Deploy	Автоматичне тестування і розгортання
Етап CI/CD	Опис дії

Інструментами етапів, наведених у таблиці 6 відповідно номеру етапа є:

1) Git, Pre-Commit Hooks;
2) SonarQube, Semgrep;
3) Custom Logic Analyzer, OPA/Rego, Z3;
4) Jenkins, GitLab CI/CD;
5) Prometheus, ELK, Sentinel.

Також, відповідно до етапів наведемо роль логічного аналізатора:

1) перевірка синтаксису правил політик доступу;
2) виявлення логічних суперечностей у правилах;
3) розгортання лише після успішної верифікації;
4) перевірка істинності та узгодженості умов;
5) виявлення порушень логічних умов у роботі.

Архітектурна схема моделі інтеграції логічного аналізатора у CI/CD-конвеєр представлена на рисунку 2.

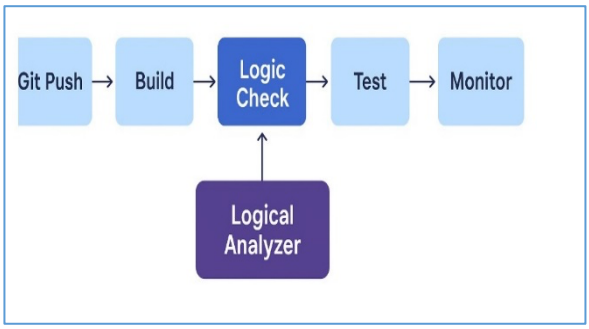

Рисунок 2 – Архітектурна схема моделі

Логічний аналізатор вбудовується між етапами "Build" і "Test", де він перевіряє:

- валідність булевих та предикатних правил;
- відсутність суперечностей у політиках;
- відповідність заданим умовам безпеки.

Наведемо приклад верифікації політик доступу для логічних умов (табл. 7):

1) адмін повинен мати право керування (isAdmin → canManageUsers);
2) не-менеджер не може змінювати акаунти (¬isManager → ¬canModifyAccounts);
3) тільки менеджер може проводити затвердження (canApprove → isManager).

Таблиця 7 – Приклад верифікації політик

Лог.умова	Результат перевірки	Дія системи
1	True	Пропустити
2	True	Пропустити
3	False	Зупинити

На рисунку 3 наведено приклад реалізації у CI/CD (Jenkins + OPA + Python).

```
        stage('Logic Verification') {
            steps {
                sh 'python logic_analyzer.py
policies.json'
                sh 'opa eval --data policies.rego
"data.access.allow" --input input.json'
            }
        }
```

Рисунок 3 – Фрагмент Jenkinsfile

Нижче наведено простий Python-скрипт для перевірки логіки політик (рисунок 4).

```
import json
def implies(a, b):
    return (not a) or b
with open("policies.json") as f:
    policies = json.load(f)
checks = [
    ("Admin rule", implies(policies["isAdmin"],
policies["canManageUsers"])),
    ("Manager rule", implies(not
policies["isManager"], not
policies["canModifyAccounts"]))
]
for name, result in checks:
    if not result:
        print(f"[FAIL] {name}")
        exit(1)
    else:
        print(f"[OK] {name}")
```

Рисунок 4 – Приклад logic_analyzer.py

Перший блок коду – частина Jenkins Pipeline, яка визначає етап «Logic Verification» (перевірка логіки). По-суті, виконуються формальні перевірки політик доступу за допомогою власного Python-аналізатора, а потім політики оцінюються через OPA для конкретного вхідного сценарію.

Результат цього етапу дозволяє переконатися, що правила політик коректні і застосовуються правильно, перш ніж рухатись далі в пайплайн.

Другий скрипт автоматично перевіряє, чи виконуються логічні залежності між політиками користувачів, і сигналізує про невідповідності, зупиняючи процес у разі помилки. Журнальну сторінку прикладу виконання пайплайну (лог) наведено в табл. 8.

Таблиця 8 – Лог виконання пайплайну

Етап	Повідомлення	Статус
Build	Code compiled successfully	True
Logic Verification	[OK] Admin rule [OK] Manager rule	True
Test	All unit tests passed	True
Deploy	Deployed to staging	True

У разі, якщо будь-яке логічне правило повертає False, конвеєр зупиняється автоматично – тобто безпека інтегрована в процес розгортання.

Запропонована модель інтеграції логічного аналізатора в CI/CD-конвеєр забезпечує формальну перевірку політик безпеки на етапі збірки та перед розгортанням, роблячи DevSecOps-процес більш надійним і контрольованим. Логічний аналізатор діє як автоматизований верифікатор умов доступу та правил конфігурації, що дозволяє:

- виявляти логічні суперечності на ранніх етапах;
- гарантувати відповідність політик корпоративним вимогам;
- зупиняти конвеєр у разі порушення правил безпеки.

Таким чином, безпека переходить із реактивного до проактивного рівня, коли її перевірка є невід'ємною частиною безперервної інтеграції та доставки.

Приклад реалізації моделювання політик доступу: кожна роль користувача формалізується через булеві змінні та логічні правила (наприклад, isAdmin → canModifyAccounts).

Автоматизована перевірка: при кожному пуші або пул-реквесті SAT-солвер перевіряє, чи можливі комбінації вхідних даних, які порушують політики доступу.

Сповіщення та блокування: якщо виявляється суперечність, CI/CD pipeline автоматично блокує розгортання і надсилає повідомлення розробникам із деталями проблеми.

Переваги підходу:

- раннє виявлення логічних помилок, що підвищує безпеку системи;
- зменшення ризику несанкціонованого доступу та витоку даних;
- автоматизація перевірки без потреби ручного аудиту.

Результат: інтеграція комп'ютерної логіки у DevSecOps дозволяє ефективно контролювати доступ у критичних системах та забезпечує надійність та безпеку релізів.

Приклади можливого застосування моделі:

- моделювання станів доступу до ресурсів (дозволено / заборонено);
- перевірка умов авторизації та автентифікації;
- виявлення суперечливих або непередбачуваних сценаріїв роботи програмного забезпечення.

Результати експериментів

Приклад розрахунку перевірки логіки доступу для уявного підприємства, який ілюструє застосування комп'ютерної логіки в DevSecOps.

Приклад розрахунку перевірки логіки доступу для уявного підприємства, який ілюструє

застосування комп'ютерної логіки в DevSecOps наведено на рисунку 5 та описано нижче.

Приклад: перевірка політик доступу на уявному підприємстві.

Умова: уявне підприємство "FinSecure" має веб-додаток для внутрішніх фінансових операцій. Є три ролі користувачів:

- адміністратор (Admin) – повний доступ до всіх функцій;
- менеджер (Manager) – доступ до управління транзакціями, але не до адміністрування користувачів;
- працівник (Employee) – лише перегляд власних даних.

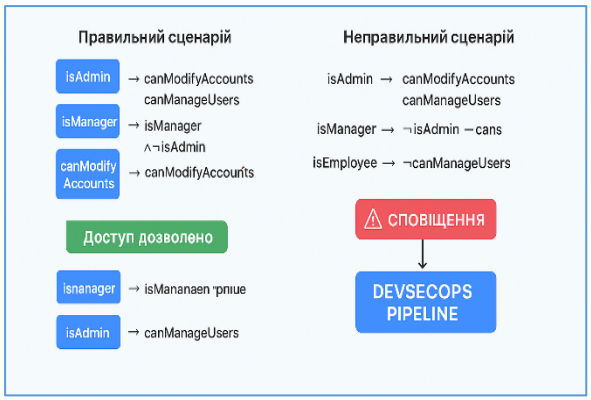

Рисунок 5 – DevSecOps інтеграція комп'ютерної логіки для перевірки безпеки (правильний та невірний сценарії)

Логічні правила:

1) isAdmin → canModifyAccounts ∧ canManageUsers;

2) isManager ∧ ¬isAdmin → canModifyAccounts;

3) isEmployee ∧ ¬isManager ∧ ¬isAdmin → canViewOwnData;

4) ¬isAdmin → ¬canManageUsers.

Перевірка правил (правильний сценарій).

Уявимо, що новий користувач має роль Manager (isManager = true, isAdmin = false, isEmployee = false).

Перевірка правил логіки доступу:

1) isAdmin → canModifyAccounts ∧ canManageUsers

isAdmin = false → правило не застосовується.

2) isManager ∧ ¬isAdmin → canModifyAccounts

true ∧ true → canModifyAccounts

canModifyAccounts = true → доступ дозволено

3) ¬isAdmin → ¬canManageUsers

true → ¬canManageUsers

canManageUsers = false → правило дотримано

4) isEmployee ∧ ¬isManager ∧ ¬isAdmin → canViewOwnData

false ∧ false ∧ true → canViewOwnData

правило не застосовується.

Висновок:

- менеджер отримує доступ до управління рахунками (canModifyAccounts = true);
- менеджер не має доступу до адміністрування користувачів (canManageUsers = false);
- всі логічні правила дотримано, суперечностей немає.

Приклад 2 ("неправильний" сценарій). Демонструє, як логічна перевірка виявляє порушення політики доступу:

Неправильний сценарій перевірки логіки доступу.

Умова: той самий додаток у підприємстві "FinSecure". Припустимо, що помилково в коді менеджеру (Manager) надано повний доступ до адміністрування користувачів.

Змінні користувача:

- isManager = true;
- isAdmin = false;
- isEmployee = false;
- canModifyAccounts = true (правильно);
- canManageUsers = true (помилка!).

Перевірка правил:

1) isAdmin → canModifyAccounts ∧ canManageUsers

isAdmin = false → правило не застосовується;

2) isManager ∧ ¬isAdmin → canModifyAccounts:

true ∧ true → canModifyAccounts

canModifyAccounts = true → правило дотримано;

3) ¬isAdmin → ¬canManageUsers

¬false → ¬true → true → false.

! Порушення: менеджер не повинен мати доступ до адміністрування користувачів, але в коді йому надано canManageUsers = true.

4) isEmployee ∧ ¬isManager ∧ ¬isAdmin → canViewOwnData

false ∧ false ∧ true → canViewOwnData

правило не застосовується.

Таким чином, логічна перевірка виявила порушення політики доступу (canManageUsers = true для користувача, який не є адміністратором).

DevSecOps пайплайн блокує розгортання і сповіщає розробників для виправлення помилки.

Результат: проблема виявлена на ранньому етапі, без ризику витоку даних або несанкціонованого доступу.

Дозволяє автоматизувати контроль безпеки у всіх наступних релізах.

Висновки

У результаті проведеного дослідження обґрунтовано та реалізовано підхід до інтеграції комп'ютерної логіки в процес DevSecOps для підвищення ефективності автоматизованої перевірки безпеки. В межах сформульованих задач:

- проведено аналітичний огляд сучасних

підходів DevSecOps та виявлено обмеження класичних методів тестування безпеки у CI/CD-конвеєрах;

- досліджено можливості застосування формальних методів, зокрема символьного виконання, SMT-розв'язувачів (Z3) та модельної перевірки (NuSMV), для автоматизованої логічної верифікації політик і коду;

- розроблено концептуальну модель інтеграції логічного аналізатора у DevSecOps-пайплайн із використанням Open Policy Agent, Rego;

- створено приклад перевірки політик безпеки для уявного підприємства із демонстрацією правильного та помилкового сценаріїв доступу, що підтвердило доцільність використання логічного контролю як етапу "security gate";

- сформовано рекомендації щодо впровадження логічних перевірок у процес безперервної інтеграції та доставки – із використанням інструментів автоматичного тестування, моніторингу та повідомлення (Slack, Jira).

Наукова новизна дослідження полягає у поєднанні принципів DevSecOps із методами комп'ютерної логіки та формальної верифікації для побудови системи автоматизованого логічного контролю безпеки; розробці узагальненої схеми інтеграції SMT-розв'язувачів у пайплайн CI/CD як окремого етапу безпечного розгортання; застосуванні концепції політик, що розширює традиційне policy-as-code і забезпечує перевірку несуперечливості та інваріантів безпеки засобами логічного висновку.

Практична цінність результатів полягає у можливості впровадження розробленої моделі для автоматичної перевірки політик доступу, налаштувань середовища та транзакцій перед розгортанням; зниження кількості вразливостей, спричинених людським фактором або логічними помилками у конфігураціях; підвищення рівня довіри до безперервної інтеграції через формальне підтвердження коректності умов безпеки.

Отримані результати створюють основу для подальших досліджень у напрямі побудови інтелектуальних систем логічного аналізу в DevSecOps, здатних до адаптивної перевірки конфігурацій, контейнерних середовищ та політик безпеки у режимі реального часу.

Література

1. Myrbakken H., Colomo-Palacios R. DevSecOps: A Multivocal Literature Review. *Proc. Int. Conf. Software Process Improvement and Capability Determination (SPICE 2017).* 2017. С. 17–29. DOI: 10.1007/978-3-319-67383-7_2. URL: https://www.researchgate.net/publication/319633880_DevSecOps_A_Multivocal_Literature_Review.
2. Zhao X., et. al. Identifying the primary dimensions of DevSecOps: A multi-method study. *Journal of Systems and Software.* 2024. URL: https://www.sciencedirect.com.
3. Doussot G. Software Verification and Analysis Using Z3. NCC Group Research Blog. 2021. URL: https://www.nccgroup.com/research-blog/software-verification-and-analysis-using-z3/
4. Bjorner N et. al. Z3 – An Efficient SMT Solver. Microsoft Research. URL: https://www.microsoft.com/en-us/research/project/z3-3/.
5. Z3Guide: A Scalable, Student-Centered and Extensible Z3 Guide. Microsoft Research Technical Report. 2025. URL: https://www.microsoft.com/en-us/research/wp-content/uploads/2025/06/Z3Guide_Paper_Tech_Report.pdf.
6. Open Policy Agent (OPA). Проєкт, документація. URL: https://www.openpolicyagent.org.
7. Cavada R., Cimatti A., Olivetti F., Pistore M., Roveri M. NuSMV 2.1 User Manual. IRST (NuSMV Project). URL: https://nusmv.fbk.eu/userman/v21/nusmv.pdf.
8. Cimatti A., Clarke E., Giunchiglia F., Roveri M. NuSMV: A New Symbolic Model Checker. *Software Tools for Technology Transfer (STTT).* 2000. Vol. 2, No. 4. P. 410–425. DOI: 10.1007/s100090050046. URL: https://www.cs.cmu.edu/~emc/papers/Papers%20In%20Refereed%20Journals/NuSMV-ANewSymbolicModelChecker.pdf.
9. ConsenSys Diligence. Mythril – Security Analysis Tool for Ethereum Smart Contracts. Release v0.23.9 (05.09.2022). URL: https://mythril-classic.readthedocs.io/en/master/.
10. White Paper: The Future of DevSecOps in a Fully Autonomous CI/CD Pipeline. DevOps.com (white paper). 2025. URL: https://devops.com/white-paper-the-future-of-devsecops-in-a-fully-autonomous-ci-cd-pipeline.
11. Bailey J. et. al. Symbolic Execution in Practice: A Survey of Applications in Vulnerability, Malware, Firmware and Protocol Analysis. *arXiv preprint.* 2025. URL: https://arxiv.org/abs/2508.06643.
12. Thiede C. A Survey of New Trends in Symbolic Execution for Software Testing and Analysis. (Report). 2023. URL: https://linqlover.github.io/symbolic-execution-survey/report.pdf.

13. CI/CD Baseline Architecture with Azure Pipelines. Microsoft Learn Documentation. 2025. URL: https://learn.microsoft.com/en-us/azure/devops/pipelines/architectures/devops-pipelines-baseline-architecture.
14. Using OPA in CI/CD Pipelines. Open Policy Agent Docs. URL: https://www.openpolicyagent.org/docs/cicd.
15. Czech S. Leveraging OPA and Rego to Automate Compliance in a CI/CD Pipeline. CodiLime Blog. 2023. URL: https://codilime.com/blog/leveraging-opa-and-rego-to-automate-compliance.
16. Golan A. Improving Continuous Verification: Deploy Fast and Safely to Production. Spot.io Blog. 2021. URL: https://spot.io/blog/improving-continuous-verification-deploy-fast-and-safely-to-production.
17. What is Mythril? Mythril Classic Documentation (Read the Docs). HITBSecConf 2018. URL: https://mythril-classic.readthedocs.io/en/master/about.html.
18. Івануса А. І., Ткачук Р. Л., Брич Т. Б., Балацька В. С., Ткаченко А. М. Методи та моделі проєктування системи автоматизованого пошуку вразливостей у WEB-додатках. *Вісник Львівського державного університету безпеки життєдіяльності.* 2024. Т. 30. С. 11–22. DOI: 10.32447/20784643.30.2024.11.
19. Сусукайло В. Використання підходу DevSecOps для аналізу сучасних загроз інформаційної безпеки. *Збірник наукових праць Центру кібербезпеки Київського університету (Csecurity).* 2021. URL: https://csecurity.kubg.edu.ua/index.php/journal/article/view/311.
20. Шкарупило В. Сценарії, методи та засоби формальної верифікації: монографія. Київ : Logos Science Publishing, 2024. URL: https://publishing.logos-science.com/index.php/books/article/view/smtzfvappskp-monograph.2021.

References

1. Myrbakken, H., and Colomo-Palacios, R. (2017), "DevSecOps: a multivocal literature review", *In Software Process Improvement and Capability Determination (SPICE 2017)*, pp. 17–29, Springer, DOI: 10.1007/978-3-319-67383-7_2.
2. Zhao, X., et al. (2024), "Identifying the primary dimensions of DevSecOps: a multi-method study", *Journal of Systems and Software*, available at https://www.sciencedirect.com.
3. Doussot, G. (2021, January 29), "Software verification and analysis using Z3", *NCC Group Research Blog*, available at: https://www.nccgroup.com/research-blog/software-verification-and-analysis-using-z3/.
4. Bjorner, N., et al. (2025), "Z3 — an efficient SMT solver", *Microsoft Research*, available at: https://www.microsoft.com/en-us/research/project/z3-3/.
5. Microsoft Research. (2025), *Z3Guide: a scalable, student-centered, and extensible Z3 guide* [Technical report], available at: https://www.microsoft.com/en-us/research/wp-content/uploads/2025/06/Z3Guide_Paper_Tech_Report.pdf.
6. Open Policy Agent (OPA). *Official documentation*, accessed 18 Oct 2025, available at: https://www.openpolicyagent.org.
7. Cavada, R., Cimatti, A., Olivetti, F., Pistore, M., and Roveri, M. (2002), *NuSMV 2.1 User Manual*, IRST (NuSMV Project), available at: https://nusmv.fbk.eu/userman/v21/nusmv.pdf.
8. Cimatti, A., Clarke, E., Giunchiglia, F., and Roveri, M. (2000), "NuSMV: a new symbolic model checker", *Software Tools for Technology Transfer*, 2(4), pp. 410–425, DOI: 10.1007/s100090050046.
9. ConsenSys Diligence. (2022, September 5), "Mythril — security analysis tool for Ethereum smart contracts (Version 0.23.9)", available at: https://mythril-classic.readthedocs.io/en/master/.
10. DevOps.com. (2025, August 21), "White paper: the future of DevSecOps in a fully autonomous CI/CD pipeline", available at: https://devops.com/white-paper-the-future-of-devsecops-in-a-fully-autonomous-ci-cd-pipeline.
11. Bailey, J., et al. (2025), "Symbolic execution in practice: a survey of applications in vulnerability, malware, firmware and protocol analysis", *arXiv preprint*, available at: https://arxiv.org/abs/2508.06643.
12. Thiede, C. (2023), "A survey of new trends in symbolic execution for software testing and analysis", available at: https://linqlover.github.io/symbolic-execution-survey/report.pdf.
13. Microsoft Learn. (2025, February 3), "CI/CD baseline architecture with Azure Pipelines", available at: https://learn.microsoft.com/en-us/azure/devops/pipelines/architectures/devops-pipelines-baseline-architecture.
14. Open Policy Agent (OPA). (n.d.), "Using OPA in CI/CD pipelines", accessed 18 Oct 2025, available at: https://www.openpolicyagent.org/docs/cicd.
15. Czech, S. (2023, May 25), "Leveraging OPA and Rego to automate compliance in a CI/CD pipeline",

CodiLime Blog, available at: https://codilime.com/blog/leveraging-opa-and-rego-to-automate-compliance.
16. Golan, A. (2021, December 7), "Improving continuous verification: deploy fast and safely to production", *Spot.io Blog*, available at: https://spot.io/blog/improving-continuous-verification-deploy-fast-and-safely-to-production.
17. *Mythril Classic Documentation*. (2018), "What is Mythril?", *Read the Docs*, available at: https://mythril-classic.readthedocs.io/en/master/about.html
18. Ivanusa, A.I., Tkachuk, R.L., Brych, T.B., Balatska, V.S., and Tkachenko, A.M. (2024), "Methods and models for designing an automated system for vulnerability search in web applications", ["Metody ta modeli proiektuvannia systemy avtomatyzovanoho poshuku vrazlyvostei u WEB-dodatkakh"] *Visnyk Lviv State University of Life Safety*, 30, pp. 11–22, DOI: 10.32447/20784643.30.2024.11.
19. Susukailo, V. (2021), "Using the DevSecOps approach to analyze modern information security threats", ["Vykorystannia pidkhodu DevSecOps dlia analizu suchasnykh zahroz informatsiinoi bezpeky"] *Csecurity Journal*, Kyiv University Center for Cybersecurity, available at: https://csecurity.kubg.edu.ua/index.php/journal/article/view/311.
20. Shkarupylo, V. (2024), "Scenarios, methods and means of formal verification", ["Stsenarii, metody ta zasoby formalnoi veryfikatsii"] *Logos Science Publishing*, available at: https://publishing.logos-science.com/index.php/books/article/view/smtzfvappskp-monograph.2021.

Надійшла до редакції 19.10.2025

[1,2]N.MASLOVA, [2]Y.SHOLOMYNSKYI
[1]Lviv State University of Life Safety, Lviv, Ukraine
[2]Donetsk National Technical University, Drohobych, Ukraine
[1]masgpp2@gmail.com, [1]nataliia.maslova@donntu.edu.ua,
[2]solompaber@gmail.com

INTEGRATING COMPUTER LOGIC IN DEVSECOPS FOR SECURITY VERIFICATION

The article explores an approach to integrating computer logic into the DevSecOps process as a means of enhancing the efficiency of automated security verification. It presents an analysis of modern methods based on formal models, symbolic execution, and SMT solvers such as Z3, which are applied within CI/CD pipelines for verifying security policies, logical consistency of components, and potential vulnerabilities. A model for embedding a logical analyzer into the development pipeline is proposed, enabling stepwise control of access policies (policy-as-code) through the Open Policy Agent (OPA) and the Rego language. A practical implementation example demonstrates both correct and erroneous scenarios of logical condition verification in a DevSecOps environment. The paper highlights the advantages of combining formal verification techniques with agile DevOps practices, which enable proactive detection of security flaws at early stages of software development. The results of this study can be applied to the creation of intelligent security systems capable of performing logical analysis of configurations, transactions, and access policies in an automated manner.

As a result of this study, an approach to integrating computer logic into the DevSecOps process for enhancing the efficiency of automated security verification has been substantiated and implemented. Within the defined tasks: an analytical review of modern DevSecOps approaches was conducted, identifying limitations of traditional security testing methods in CI/CD pipelines; the potential of formal methods, including symbolic execution, SMT solvers (Z3), and model checking (NuSMV), for automated logical verification of policies and code was explored; a conceptual model for integrating a logic analyzer into the DevSecOps pipeline using Open Policy Agent (OPA) and Rego was developed; a practical example of security policy verification for a hypothetical enterprise was created, demonstrating both correct and incorrect access scenarios, confirming the feasibility of logical control as a "security gate" stage; recommendations were formulated for implementing logical checks within continuous integration and delivery processes using automated testing, monitoring, and notification tools (Slack, Jira).

The scientific novelty of the study lies in combining DevSecOps principles with computer logic and formal verification methods to build an automated logical security control system; developing a generalized scheme for integrating SMT solvers into the CI/CD pipeline as a separate secure deployment stage; and applying a policy concept that extends traditional policy-as-code to ensure consistency and security invariants through logical reasoning.

The practical significance of the results is in enabling automatic verification of access policies, environment settings, and transactions before deployment; reducing vulnerabilities caused by human error or logical configuration mistakes; and increasing trust in continuous integration through formal validation of security conditions.

These results provide a foundation for further research in developing intelligent logic-analysis systems in DevSecOps, capable of adaptive verification of configurations, containerized environments, and security policies in real time.

Keywords: DevSecOps, computer logic, policy-as-code, SMT solver, Z3, formal methods, CI/CD pipeline, security verification