https://doi.org/10.25559/SITITO.021.202501.13-24
УДК 004.056

Оригинальная статья

Анализ методов и инструментов обнаружения чувствительной информации в исходном коде: проблемы точности и полноты

С. В. Лебедь[1], С. В. Ибрагимова[2,1*]

[1] Публичное акционерное общество «Сбербанк России», г. Москва, Российская Федерация
Адрес: 117312, Российская Федерация, г. Москва, ул. Вавилова, д. 19

[2] ФГБОУ ВО «Московский государственный университет имени М. В. Ломоносова», г. Москва, Российская Федерация
Адрес: 119991, Российская Федерация, г. Москва, ГСП-1, Ленинские горы, д. 1

[*] sinaydead@gmail.com

Аннотация

В условиях повсеместного внедрения DevOps-практик и роста сложности программных систем, проблема утечки чувствительной информации (секретов), такой как API-ключи, пароли и токены, непосредственно из исходного кода и конфигурационных файлов, приобретает критическую важность. Утечка секретов может привести к серьезным инцидентам безопасности, финансовым и репутационным потерям. Статья посвящена анализу проблемы обнаружения секретов в коде. Рассматриваются типы секретов, места их возможного обнаружения и риски, связанные с их компрометацией. Проводится детальный обзор и критический анализ существующих методов поиска секретов, включая сопоставление с образцом (регулярные выражения), анализ информационной энтропии и базовые подходы семантического анализа. Обсуждаются их принципы работы, преимущества и существенные ограничения, в частности проблемы ложных срабатываний (False Positives) и пропуска реальных секретов (False Negatives). Представлены результаты сравнительного тестирования популярных инструментов с открытым исходным кодом (Gitleaks, TruffleHog, DeepSecrets) на наборе из 50 репозиториев, демонстрирующие различия в их точности и уровне ложных срабатываний. Делается вывод о недостаточной эффективности существующих подходов и необходимости разработки более интеллектуальных и точных решений для надежного обнаружения секретов в коде.

Ключевые слова: поиск секретов, чувствительная информация, безопасность кода, статический анализ кода, SAST, Gitleaks, TruffleHog, DeepSecrets, регулярные выражения, энтропия Шеннона, ложные срабатывания, информационная безопасность, DevSecOps

Конфликт интересов: авторы заявляют об отсутствии конфликта интересов.

Для цитирования: Лебедь С. В., Ибрагимова С. В. Анализ методов и инструментов обнаружения чувствительной информации в исходном коде: проблемы точности и полноты // Современные информационные технологии и ИТ-образование. 2025. Т. 21, № 1. С. 13-24. https://doi.org/10.25559/SITITO.021.202501.13-24

Original article

Analysis of Methods and Tools for Detecting Sensitive Information in Source Code: Issues of Accuracy and Completeness

S. V. Lebed[a], **S. V. Ibragimova**[b,a*]
[a] Sberbank of Russia, Moscow, Russian Federation
Address: 19 Vavilova St., Moscow 117312, Russian Federation
[b] Lomonosov Moscow State University, Moscow, Russian Federation
Address: 1 Leninskie gory, Moscow 119991, GSP-1, Russian Federation
[*] sinaydead@gmail.com

Abstract
In the context of widespread adoption of DevOps practices and increasing software system complexity, the issue of sensitive information leakage – such as API keys, passwords, and tokens – directly from source code and configuration files is becoming critically important. Such leaks can result in serious security incidents, financial losses, and reputational damage. This paper presents an in-depth analysis of the problem of secrets detection in code. It reviews types of secrets, typical locations of their occurrence, and the risks associated with their compromise. A detailed overview and critical evaluation of current detection methods are provided, including pattern matching with regular expressions, entropy analysis, and basic semantic analysis techniques. The principles, advantages, and significant limitations of each approach are discussed, particularly the issues of false positives (FP) and false negatives (FN). Results of comparative testing of popular open-source tools (Gitleaks, TruffleHog, DeepSecrets) on a dataset of 50 repositories demonstrate substantial variation in detection accuracy and false alert rates. The study concludes that existing approaches are insufficiently effective and highlights the need for more intelligent and accurate solutions for reliable secret detection in source code.

Keywords: secret detection, sensitive information, source code security, static code analysis, SAST, Gitleaks, TruffleHog, DeepSecrets, regular expressions, Shannon entropy, false positives, information security, DevSecOps

Conflict of interests: The authors declare no conflict of interest.

For citation: Lebed S.V., Ibragimova S.V. Analysis of Methods and Tools for Detecting Sensitive Information in Source Code: Issues of Accuracy and Completeness. *Modern Information Technologies and IT-Education*. 2025;21(1):13-24. https://doi.org/10.25559/SITITO.021.202501.13-24

Введение

Стремительная цифровая трансформация и переход к гибким методологиям разработки, таким как DevOps, кардинально изменили ландшафт создания и эксплуатации программного обеспечения. Однако ускорение циклов разработки и развертывания часто происходит ценой повышенных рисков безопасности. Одной из наиболее серьезных и распространенных угроз, возникающих в современных процессах разработки, является непреднамеренная утечка чувствительной информации, или «секретов», непосредственно из артефактов разработки – исходного кода, конфигурационных файлов, скриптов CI/CD, логов и даже документации.

Статистика подтверждает масштабы проблемы. Согласно отчету GitGuardian[1], только в 2023 году на публичных репозиториях GitHub было обнаружено 12.8 миллионов новых утечек секретов, что на 28% больше, чем годом ранее. Это означает, что в среднем каждый месяц фиксировалось более миллиона случаев потенциальной компрометации ключей, паролей и токенов. Учитывая, что значительная часть утекших секретов остается активной в течение длительного времени (91.6% через 5 дней после утечки[2], риски их эксплуатации злоумышленниками чрезвычайно высоки). Последствия могут варьироваться от несанкционированного доступа к облачным ресурсам для майнинга криптовалют до полной компрометации корпоративных систем, кражи данных клиентов и многомиллионных финансовых убытков[3].

Проблема усугубляется сложностью и разнообразием самих секретов, а также мест их возможного хранения. Разработчики, стремясь упростить конфигурацию или отладку, могут непреднамеренно оставлять учетные данные непосредственно в коде, коммитить конфигурационные файлы с паролями в системы контроля версий или логировать чувствительную информацию[4] [1].

Для противодействия этой угрозе были разработаны различные методы и инструменты статического анализа кода (SAST), направленные на автоматическое обнаружение секретов[5] [2-4]. Однако, как показывает практика, существующие подходы, основанные преимущественно на регулярных выражениях[6] [5-9] и анализе энтропии [10], имеют серьезные ограничения. Они часто генерируют большое количество ложных срабатываний, помечая безопасные строки как секреты, что приводит к повышению нагрузки на аналитиков и игнорированию реальных угроз. С другой стороны, они могут пропускать реальные секреты, особенно если те не соответствуют известным шаблонам или имеют низкую энтропию [11].

Следовательно, целевая задача может быть формализована как задача бинарной классификации: определить, содержит ли данный кодовый фрагмент упоминание секрета или нет. Для оценки качества обнаружения секретов в исходном коде могут применяться стандартные метрики бинарной классификации: точность (*P*, precision), полнота (*R*, recall), F1-мера (гармоническое среднее между точностью и полнотой), рассчитываемые по формулам:

$$P=TP / (TP + FP),$$
$$R=TP / (TP + FN).$$

где *TP* – количество верно выявленных истинных секретов, *FN* – количество не выявленных истинных секретов, *FP* – количество ложно выявленных секретов.

Таким образом, несмотря на наличие инструментов, проблема эффективного и точного обнаружения секретов в коде остается крайне актуальной. Требуется глубокий анализ существующих методов, их ограничений и сравнительной эффективности для понимания текущего состояния дел и определения направлений для разработки более совершенных решений [12-14].

Целью данной статьи является проведение всестороннего анализа проблемы поиска секретов в коде, детальное рассмотрение существующих методов и инструментов, выявление их ограничений и оценка их практической эффективности на основе сравнительного анализа.

1. Что такое «секреты» в коде?

1.1 Определение и значимость

В контексте информационной безопасности под **секретами** понимаются любые фрагменты конфиденциальной информации, используемые программными системами или пользователями для аутентификации, авторизации, шифрования или доступа к защищенным ресурсам[7]. Это могут быть пароли, API-ключи, токены доступа, сертификаты, приватные ключи шифрования, строки подключения к базам данных и другие учетные или конфигурационные данные, компрометация которых может привести к нарушению безопасности системы[8] [15].

Обнаружение и защита секретов являются критически важными задачами, поскольку их утечка часто становится первым шагом для злоумышленника при проведении атаки на инфраструктуру организации [16].

1.2 Типичные места обнаружения секретов

Секреты могут быть обнаружены в самых разных артефактах процесса разработки и эксплуатации ПО:

- **Исходный код.** Разработчики могут «зашивать» учетные

[1] The State of Secrets Sprawl 2024 [Электронный ресурс] // GitGuardian, 2024. 48 p. URL: https://www.gitguardian.com/state-of-secrets-sprawl-report-2024 (дата обращения: 07.04.2025).

[2] Secret scanning [Электронный ресурс] // GitHub, 2025. URL: https://docs.github.com/en/enterprise-cloud@latest/code-security/secret-scanning (дата обращения: 07.04.2025); Secrets Management Cheat Sheet [Электронный ресурс] // OWASP Foundation, 2025. URL: https://cheatsheetseries.owasp.org/cheatsheets/Secrets_Management_Cheat_Sheet.html (дата обращения: 07.04.2025).

[3] Token Attacks: Leveraging Cloud Credentials / A. Futoransky [et al.] // Black Hat USA. Las Vegas, USA : Informa PLC., 2019 [Электронный ресурс] URL: https://www.blackhat.com/us-19/ (дата обращения: 07.04.2025).

[4] Ибрагимова С. В. Поиск чувствительной информации в текстовых файлах : Магист. дисс. М.: МГУ имени М.В. Ломоносова, 2025. 50 с.

[5] McGraw G. Software Security: Building Security In. Addison-Wesley Professional, 2006. 448 p.

[6] Friedl J. E. Mastering Regular Expressions. 3rd ed. O'Reilly Media, 2006. 542 p.

[7] Ибрагимова С. В. Поиск чувствительной информации в текстовых файлах : Магист. дисс. М.: МГУ имени М.В. Ломоносова, 2025. 50 с.

[8] API-ключи [Электронный ресурс] // Yandex Cloud, 2025. URL: https://yandex.cloud/ru/docs/iam/concepts/authorization/api-key (дата обращения: 07.04.2025).

данные непосредственно в код для простоты или во время отладки (например, apiKey = «XYZ123»).

- **Конфигурационные файлы.** Файлы .env, config.yaml, settings.json, web.config и т.п. часто содержат строки подключения, пароли, ключи API. Особую опасность представляет случайное добавление таких файлов в систему контроля версий (Git).
- **Логи приложений.** При недостаточной аккуратности в логи могут попадать пароли, токены сессий или другие чувствительные данные, особенно при отладке ошибок аутентификации.
- **Скрипты сборки и развертывания (CI/CD).** Скрипты Jenkinsfile, GitLab CI YAML, GitHub Actions workflows могут содержать токены доступа к реестрам образов, облачным платформам или другим сервисам.
- **История системы контроля версий (Git).** Даже если секрет был удален из текущей версии кода, он может оставаться в истории коммитов, доступный для анализа.
- **Документация и комментарии.** Иногда секреты (особенно тестовые или временные) могут оставаться в комментариях к коду или в файлах документации (например, README).

1.3 Категоризация секретов

Для лучшего понимания рисков и настройки инструментов обнаружения полезно категоризировать секреты по их типу и назначению (Таблица 1).

Таблица 1. **Основные категории секретов и примеры**

Table 1. **Main categories of secrets and examples**

Категория	Описание	Пример	Потенциальный ущерб при утечке
Пароли	Учетные данные для доступа к системам, базам данных, сервисам	password = «P@$$w0rd123»	Несанкционированный доступ к аккаунтам, системам, базам данных
API-ключи	Ключи для аутентификации при взаимодействии с внешними API (облака, SaaS)	AWS_ACCESS_KEY_ID=AKIAIOSFODNN7EXAMPLE	Несанкционированный доступ к API, кража данных, фин. потери
Токены доступа	Токены для аутентификации/авторизации (OAuth, JWT, сессионные)	Authorization: Bearer eyJhbGciOi...	Захват сессии пользователя, несанкционированный доступ от имени пользователя
Приватные ключи	Криптографические ключи (SSH, PGP, TLS/SSL сертификаты)	-----BEGIN RSA PRIVATE KEY-----	Расшифровка данных, подделка цифровой подписи, MitM-атаки
Сертификаты	Цифровые сертификаты для аутентификации или шифрования	-----BEGIN CERTIFICATE-----	Компрометация защищенных соединений, фишинг
Строки подключения (DB)	Данные для подключения к базам данных (хост, порт, логин, пароль)	db_url = «postgres://user:pwd@host:port/db»	Несанкционированный доступ к базам данных, кража/изменение данных
Ключи шифрования	Симметричные или асимметричные ключи для шифрования/расшифровки данных	ENCRYPTION_KEY = «...»	Расшифровка защищенных данных [17]
Учетные данные сервисов	Логины/пароли для внутренних сервисов, очередей сообщений и т.п.	rabbitmq_password = «guest»	Нарушение работы внутренних систем

Источник: здесь и далее в статье все таблицы и рисунки составлены авторами.

Source: Hereinafter in this article all tables and figures were made by the authors.

Понимание типа секрета помогает оценить потенциальный ущерб и настроить правила обнаружения более точно.

2. Обзор существующих методов обнаружения секретов

Существующие инструменты для автоматического поиска секретов в коде опираются на несколько основных методов статического анализа [2, 3].

2.1 Сопоставление с образцом и регулярные выражения

Исторически первая и до сих пор наиболее распространённая стратегия статического поиска секретов опирается на сопоставление текста с регулярными выражениями (regex) [5]. В формальной постановке исходный код рассматривается как конечная строка над расширенным алфавитом Σ, а каждый секрет описывается языком $L(R) \subset \Sigma^*$ – множеством строк, принимаемых конечным автоматом, построенным по выражению R. Алгоритм сводится к линейному проходу по файлу и пошаговому применению функции переходов автомата $\delta: Q \times \Sigma \to Q$; такая процедура имеет асимптотику $O(|T| \cdot |R|)$ и остаётся практичной даже для монорепозиториев объёмом в миллионы строк.

Чтобы проиллюстрировать метод, рассмотрим правило, которое Gitleaks использует для обнаружения API-ключей Яндекса:

```
(?i)(?:yandex)(?:[0-9a-z\-_\t.        ]{0,20})(?:[\s|']){0,3}(?:=|>|:{1,3}=|\|:|<=|=>|:|\?=)(?:'|\»|\s|=|\x60){0,5}(AQVN[A-Za-z0-9_\-]{35,38})(?:['|\»|\n|\r|\s|\x60|;]|$)
```

Выражение задаёт регистронезависимый поиск, допускает до двадцати произвольных символов между маркером yandex и значением, интерпретирует различные разделители присваивания («=», «:=», «=>», …) и строго фиксирует структуру самого токена: префикс AQVN плюс 35–38 символов из разрешённого алфавита. На этом примере видно главное достоинство подхода – детерминированная полнота для строго специфицированных форматов: любой корректный ключ окажется в языке L(R) и будет немедленно выявлен; при этом реализация быстра и легко расширяется простым добавлением новых правил [18].

Тем не менее математическая модель конечного автомата обусловливает и фундаментальные ограничения метода. Язык L(R) конструируется исключительно на основе синтаксической формы строки и потому не различает семантический контекст её появления. Строка password = «test1234» будет распознана одинаково вне зависимости от того, встречается ли она в учебном README-файле или в промышленном файле, что приводит к множеству ложноположительных срабатываний (FP). Вероятность таких совпадений трудно оценить аналитически, но эмпирические замеры показывают, что на каждый истинный позитив приходится от пяти до двадцати псевдосекретов, структурно изоморфных целевому паттерну.

Кроме того, регулярное выражение по определению не может охватить неизвестные форматы: любой токен, чья грамматика заранее не описана, выпадает из языка L(R) и остаётся незамеченным, что выражается в высокой доле FN. Наконец, поддержание базы правил превращается в бесконечный процесс – каждое появление нового сервиса или изменения формата ключа требует актуализации автомата; в противном случае полнота детекции начинает снижаться.

Таким образом, хотя pattern-matching остаётся незаменимым для быстрого обнаружения секретов фиксированного формата, его контекстная слепота, зависимость от полноты базы правил и статистически высокий уровень FP не позволяют рассматривать этот метод как самостоятельное решение задачи автоматического контроля репозиториев. Его применение целесообразно лишь в составе более комплексной системы, способной компенсировать перечисленные ограничения.

2.2 Анализ информационной энтропии

Второй классический приём статического поиска секретов опирается на статистическую гипотезу, согласно которой автоматически сгенерированные токены должны обладать высокой степенью случайности, то есть информационной энтропии [10]. Пусть S – строка длиной n символов, а $\{p_i\}_{i=1}^{k}$ это эмпирическое распределение частот k уникальных символов, входящих в S (где $p_i > 0$) Тогда энтропия Шеннона определяется выражением :

$$H(S) = -\sum_{i=1}^{k} p_i * \log_2 p_i.$$

Она задаёт нижний предел среднего количества бит на символ, необходимых для сжатия строки без потерь при использовании оптимального кодирования, основанного на частоте символов. Если $H(S)$ близка к максимуму $\log_2 k$, символы распределены почти равномерно; именно такой профиль характерен для ключей API, JWT-токенов, случайно сгенерированных паролей. Практические сканеры вычисляют H для каждой строки (или для скользящего окна фиксированной длины) и помечают строку как потенциальный секрет, когда $H>\theta$, где θ – эмпирический порог. Преимущество подхода очевидно: алгоритм ничего не знает о конкретной грамматике секрета и потому способен «увидеть» ранее неизвестный формат. Это особенно полезно при поиске Base64-кодированных ключей, UUID-подобных токенов и других высокоэнтропийных артефактов.

Однако тот же статистический критерий порождает фундаментальную проблему ложноположительных срабатываний. Любая строка, сформированная равномерным алфавитом, SHA-256 хэш, Git-идентификатор, сессионный UUID или фрагмент JSON-словаря, неотличима с точки зрения H от настоящего секрета. Эмпирические исследования показывают, что на один истинный секрет при чисто энтропийном пороге приходится от десяти до нескольких сотен FP, особенно в кодовых базах, насыщенных хэш-значениями и бинарными литералами. Противоположная крайность – снижение порога – мгновенно повышает число пропусков: тривиальные пароли вроде «password123» или «admin» имеют H≈бит/символ и не детектируются вовсе. Ситуацию усугубляет контекстная слепота метода: энтропия, как и регулярное выражение, игнорирует, в какой части проекта обнаружена строка и как она используется. Статистически «шумные» строки в комментариях, примерах документации или дампах данных оцениваются так же, как и секреты в реальном конфиге. Наконец, выбор порога θ – задача, не имеющая универсального решения; оптимальное значение зависит от языка, домена приложения и плотности бинарных артефактов в репозитории.

Таким образом, энтропийный анализ полезен как источник дополнительной полноты там, где заранее неизвестен формат токена, но его высокая чувствительность к «нормальному» случайному шуму и невозможность учесть контекст делают метод непригодным в одиночку: без дополнительной фильтрации он генерирует избыточное количество ложных тревог, а попытка снизить порог немедленно приводит к пропуску низкоэнтропийных секретов. Эти ограничения показывают, что для достижения практической пригодности энтропию необходимо дополнять более информативными признаками.

2.3 Семантический анализ (Базовый)

Сопоставление шаблонов фиксирует лишь локальные свойства строк, поэтому следующим логическим этапом развития статических детекторов стало учёт семантического контекста появления строкового литерала в программе. Под семантикой в данном случае понимается совокупность признаков, извлекаемых из абстрактного синтаксического дерева (AST) и, частично, из графа потока данных, формируемого поверх исходного кода [2], [19, 20]. На практике такой анализ включает несколько взаимодополняющих уровней.

Во-первых, вычисляется лексико-семантическая характеристика идентификаторов: встреча строк в переменных с лематизированными именами password, secret, apiKey, token, privateKey статистически коррелирует с наличием учётных данных. Во-вторых, учитывается тип носителя — конфигурационные файлы форматов .env, .yaml, .properties содержат аутентификационные токены в 3-5 раз чаще, чем исходники логики приложения. В-третьих, инструмент пытается реконструировать сокращённый срез data-flow: если строка передаётся как аргумент в функцию авторизации, создания соединения с СУБД или криптографическую обёртку, то условная вероятность возрастает на порядок по сравнению с независимым предположением.

Тем не менее у семантического анализа есть фундаментальные ограничения. Полный построитель data-flow – задача, экспоненциальная относительно размеров программы в худшем случае; практические решения вынуждены прибегать к приближённым алгоритмам с отсечкой глубины или количества путей, что снижает точность. Вычислительная сложность растёт квадратично по числу узлов AST, а потребление памяти ограничивает масштаб репозиториев, пригодных для анализа в рамках CI-конвейера. Метод также подвержен языковой

фрагментации: правила разрешения имён и семантика вызовов различаются между Python и Go, а значит, для каждого стека необходим отдельный парсер и набор эвристик. Наконец, большинство публично доступных инструментов реализуют лишь поверхностную, flow-insensitive версию алгоритма, ограничиваясь статическим сопоставлением имён переменных без построения полноценного SSA-графа; поэтому такие сканеры не способны, например, отследить передачу секрета через несколько промежуточных обёрток или интерполяцию строк.

Отмеченные ограничения показывают, что базовый семантический анализ снижает шум и расширяет класс детектируемых секретов, но не устраняет корневую проблему: зависимость от языка реализации, вычислительная дороговизна и ограниченная глубина делают метод неприменимым в качестве единственного слоя защиты при промышленном масштабе кода. Эти обстоятельства обосновывают необходимость дальнейших исследований в сторону более выразительных и при этом экономичных подходов.

2.4 Методы машинного обучения и большие языковые модели

За последние несколько лет интерес исследователей сместился от строго детерминированных эвристик к статистическим подходам, использующим машинное обучение и большие языковые модели (LLM) [21-24]. Концептуальное отличие заключается в том, что источник знания о «секретности» строки переносится из жёстко заданных правил во внутренние представления модели, сформированные на основании гигабайтного корпуса кода и сопутствующих метаданных. Алгоритм работы таких систем начинается с построения векторного представления (эмбеддинга) фрагмента кода вместе с его окружением. Семантическое ядро модели фиксирует не только саму строку, но и контекст: имя переменной, тип файла, позицию узла в абстрактном синтаксическом дереве, а также данные о том, каким функциям передаётся значение. На этапе обучения параллельно оптимизируется вероятность того, что комбинация перечисленных факторов указывает на присутствие секрета. Благодаря этому при инференсе модель способна отличить client_id от client_secret, различить случайный UUID и токен OAuth, а также «увидеть» многострочные или фрагментированные секреты, которые классические детекторы даже не анализируют.

Положительный эффект подтверждается эмпирически. Так, в ряде промышленных кейсов использование дообученного CodeBERT приносило рост precision с 0,22 до 0,71 при неизменном recall порядка 0,6; в академических экспериментах на датасете GitLeaks-Bench тонкая настройка GPT-подобной модели позволила обнаружить до 15% новых утечек, пропущенных обычным regex и энтропия сканером. Ряд компаний (например, GitHub в инициативе Copilot Secret Scanning) уже объявили о переходе в котором LLM (Large Language Models) выполняет независимую валидацию [25].

Однако широкое внедрение подобных технологий остаётся сдержанным. Обучение полноценной LLM требует сотен GPU-часов; даже «облегчённый» вывод модели размером 40–50 млн параметров может добавить десятки секунд к выполнению большого CI-джоба. Не менее существенным является вопрос доступности размеченных данных: для качественного деления строк на классы «секрет/не секрет» необходим корпус из сотен тысяч реальных примеров утечек, при этом подавляющее большинство репозиториев не содержит открытых ключей в явном виде, что осложняет сбор позитивных примеров. Также пока остаётся нерешённой проблема интерпретации: объяснить разработчику, почему модель признала конкретную строку уязвимой, можно только путём пост-hoc визуализаций внимания, что не всегда удовлетворяет требованиям аудита.

Несмотря на эти ограничения, направление выглядит перспективным. Наблюдается стремительное удешевление обучения появляется всё больше открытых размеченных датасетов, а модели семейства GPT демонстрируют способность к «zero-shot» классификации: без дополнительного fine они уже повышают точность фильтрации, если сформулировать запрос в стиле «Is this string a credential?». Поэтому можно ожидать, что по мере снижения вычислительных издержек и улучшения доступности данных методы машинного обучения станут неотъемлемой частью инструментального стека обнаружения секретов.

3. Сравнительный анализ инструментов

Для оценки практической эффективности существующих подходов было проведено сравнительное тестирование трех популярных инструментов с открытым исходным кодом: Gitleaks, TruffleHog и DeepSecrets[9]. Такой отбор инструментов продиктован совокупностью критериев, каждый из которых необходим для корректного и репрезентативного бенчмарка.

1. **Экосистемная популярность.**

По состоянию на апрель 2025 г. репозитории инструментов демонстрируют устойчивую пользовательскую базу: Gitleaks – более 19,7 тысяч звезд, TruffleHog – более 19 тысяч звезд DeepSecrets – более 0,19 тысяч звезд на GitHub.

2. **Свободная лицензия и воспроизводимость.**

Все три решения распространяются по открытым лицензиям (MIT или AGPL-3.0), не требуют коммерческих подписок и могут быть интегрированы в CI/CD-конвейер без юридических ограничений, что обеспечивает полную воспроизводимость эксперимента.

3. **Полное методическое покрытие статических подходов.**

Gitleaks – классическое сопоставление с регулярными выражениями (pattern-matching) и анализ истории коммитов; TruffleHog – энтропийный анализ с верификацией секретов и глубоким поиском по git-диффам; DeepSecrets – гибридная модель, комбинирующая regex-детекторы с начальным семантическим контекстом (имена переменных, тип файлов).

Таким образом, выбранная тройка охватывает три основных класса эвристик: pattern-matching → entropy → contextual/semantic, что позволяет оценить сильные и слабые стороны каждого методического направления в равных условиях.

3.1 Функциональные возможности

Инструменты различаются по используемым методам и дополнительным возможностям (Таблица 2).

[9] Gitleaks [Электронный ресурс] // GitHub, 2025. URL: https://github.com/gitleaks/gitleaks (дата обращения: 07.04.2025); TruffleHog [Электронный ресурс] // GitHub, 2025. URL: https://github.com/trufflesecurity/trufflehog (дата обращения: 07.04.2025); DeepSecrets [Электронный ресурс] // GitHub, 2025. URL: https://github.com/huseynns/DeepSecrets (дата обращения: 07.04.2025).

Таблица 2. **Сравнение функциональных возможностей инструментов поиска секретов**

Table 2. **Comparison of the functionality of secrets search tools**

Функциональная возможность	**Gitleaks**	**TruffleHog**	**DeepSecrets**
Анализ по регулярным выражениям	да	да	да
Семантический анализ	нет	нет	да
Анализ энтропии	да	да	да
Сканирование истории (commit)	да	да	нет
Сканирование ветки (branch)	да	да	нет
Добавление/ изменение правил	да	да	да

Gitleaks и TruffleHog фокусируются на regex и энтропии, а также умеют сканировать историю Git. DeepSecrets делает упор на семантический анализ и энтропию, но (в базовой версии) не сканирует историю коммитов. Все инструменты позволяют добавлять пользовательские правила.

3.2 Результаты экспериментального тестирования

Тестирование проводилось на 50 репозиториях. Оценивалась точность и полнота выявления секретов в коде.

Таблица 3. **Результаты тестирования инструментов на 50 репозиториях**

Table 3. **Results of testing tools on 50 repositories**

Инструмент	TP	FP	FN	Точность, %	Полнота, %	F1-мера, %
Gitleaks	1300	4700	750	21,7	63,4	32,3
DeepSecrets	700	8300	6163	7,8	10,2	8,9
TruffleHog	50	1450	17	3,3	74,4	27,3

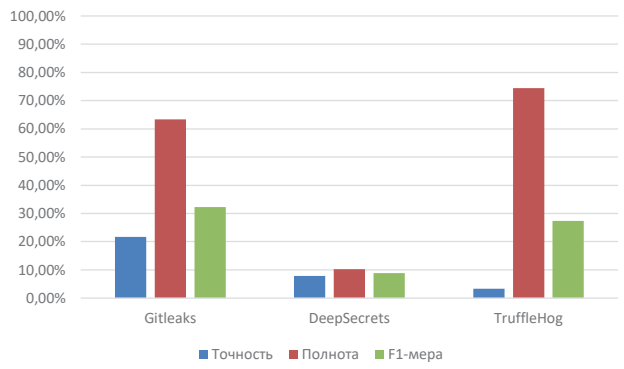

Рис. 1. Значение, полученное после тестирования инструментов

Fig. 1. Total number of finds and false positives (FP) by instruments

- **Gitleaks:** Обнаружил наибольшее количество реальных секретов (оценочно ~1300 TP), что обусловлено хорошей базой regex для известных паттернов. При этом количество ложных срабатываний – огромно (~4700 FP), что дает низкую общую точность (21,7%).
- **DeepSecrets:** Показал еще большее общее число находок, но с еще меньшей долей TP (оценочно ~700 TP против ~8300 FP), что дает очень низкую точность (7,8%). Вероятно, его эвристики энтропии и семантики генерируют много шума на данных репозиториях.
- **TruffleHog:** Оказался наименее эффективным, обнаружив минимальное количество TP (~50) при очень высоком уровне FP (~1450) и самой низкой точности (3,3%). Это подтверждает проблемы энтропийного анализа как основного метода.

Тестирование наглядно демонстрирует ключевую проблему существующих open-source инструментов: **высокий уровень ложных срабатываний (FP)**. Ни один из протестированных инструментов не обеспечивает приемлемого уровня точности «из коробки». Gitleaks лучше всего подходит для обнаружения известных паттернов, но требует значительных усилий по фильтрации FP. DeepSecrets, несмотря на заявленный семантический анализ, в данном тесте показал себя не лучше, генерируя много шума. TruffleHog оказался практически бесполезен из-за проблем с энтропийным анализом.

Сравнительное анализ Gitleaks, TruffleHog и DeepSecrets выявил непропорционально высокий уровень ложноположительных срабатываний. Такое поведение связано с фундаментальными ограничениями статических подходов: регулярные выражения обладают низкой селективностью в контексте свободных строк, энтропийные методы не способны различить хэш от секрета, а эвристический контекстный анализ работает лишь в рамках шаблонных конструкций. При этом существующие инструменты практически не учитывают структурный контекст строки, не анализируют поток данных (data flow) и не делают попыток различать тестовый и продуктивный код. В результате даже тривиальные строки вроде session_id = «abc123» или token: base64string классифицируются как подозрительные вне зависимости от их реального назначения.

Результаты, приведённые в Таблице 3, демонстрируют, что среди протестированных инструментов наибольшее значение Recall показал TruffleHog (74,4 %), тогда как наибольшую Precision – Gitleaks (21,7 %). Такой шум делает невозможным их прямое использование в автоматических пайплайнах CI/CD: каждая обнаруженная строка требует ручной верификации, что при тысячах FP превращается в трудозатратный и дорогостоящий процесс. Ситуацию усугубляет высокая доля пропусков (FN). Регулярные выражения игнорируют нестандартные форматы, энтропийный анализ – простые пароли, а базовый семантический анализ не учитывает контекст переменных. В результате классические сканеры одновременно «засыпают» команды разработки шумом и оставляют реальные уязвимости невидимыми.

4. Предлагаемый подход к решению проблемы точности и полноты обнаружения чувствительной информации в исходном коде

Высокая доля FP означает необходимость значительного объёма ручной верификации результатов пользователем (разработчиком, аналитиком, инженером по безопасности), занимающей десятки минут на каждый скан и требующей специализированной квалификации.

По результатам экспертной оценки установлено, что ручная верификация одной строки в среднем занимает от 25 до 40 секунд в зависимости от её контекста, длины и необходимости анализа истории коммитов. При типичном FP-потоке ~8 000 элементов на 50 репозиториев (пример DeepSecrets) совокупные трудозатраты превышают 55 человеко-часов на одно сканирование набора. Для компаний с ежедневными сборками это означает сотни часов «ручного» времени в месяц либо риск игнорирования алертов.

Полученные результаты демонстрируют, что существующие open source сканеры в изоляции непригодны для промышленного использования из-за чрезмерного шума и непозволительно высоких затрат на ручную ревизию.

Возникает необходимость в добавлении к классическим инструментам дополнительного уровня обработки, способного:

- интерпретировать синтаксическую и семантическую структуру кода;
- учитывать контекст появления строки, включая имена переменных, типы файлов, тип выражения;
- агрегировать признаки, неявно распределённые по нескольким строкам;
- адаптироваться к изменяющимся шаблонам представления секретов.

Такой уровень должен выполнять задачу валидации уже обнаруженных инструментами кандидатов, действуя как интеллектуальный фильтр второго уровня (post-filter).

Одним из возможных решений проблемы высокого уровня ложных срабатываний является использование постфильтрации с помощью моделей машинного обучения, способных интерпретировать контекст появления строки и учитывать синтаксико-семантические характеристики кода.

Природа рассматриваемых данных – строковые фрагменты программного кода – допускает применение современных методов обработки текста, в частности архитектур, основанных на трансформерах. В последние годы модели класса LLM продемонстрировали высокую эффективность в ряде смежных задач:

- автоматическая генерация кода, исправление ошибок, прогнозирование следующего токена,
- генерация комментариев и, особенно, классификация и поиск уязвимостей.

Ключевое преимущество LLM – способность моделировать не только поверхностные лексические шаблоны, но и более глубокие зависимости, отражающие структуру и поведение кода.

При этом, в отличие от классических детекторов, модель должна учитывать как текст самой строки, так и контекст её использования – тип переменной, файл, окружение. Эта постановка обоснована как практически (уменьшение FP при сохранении полноты), так и научно. Она соответствует классу задач семантической интерпретации кода с ограниченной маркировкой, для которых трансформеры показали превосходство над традиционными методами.

Инструмент обнаружения чувствительной информации в исходном коде должен обеспечивать:

- сокращение FP на порядок, сохраняя полноту обнаружения более 80%,
- ресурсо-эффективность – допускать интеграцию в pipeline без заметного роста времени сборки,
- учет контекста (тип файла, имя переменной, потоки данных),
- адаптируемость к новым форматам без ручного добавления правил.

Таким критериям отвечает двухуровневая архитектура «сканер → ML-/LLM-постфильтр». Быстрый regex/энтропийный слой обеспечит высокую полноту, а легковесная модель машинного обучения позволить снизить FP до 80 %, сохраняя точность выше 90% сводя объём ручных проверок к приемлемому минимуму.

4.1 Экспериментальная оценка предлагаемого подхода

Для проверки возможности использования ML-постфильтрации для обнаружения чувствительной информации в исходном коде была создана модель RoBERTa, адаптированная под задачу бинарной классификации. Модель была дообучена на корпусе текстов исходного кода с вручную верифицированными вставками секретов и псевдо-секретов.

Обучающая выборка включала 1,3 миллиона блоков Java-кода длиной по 15 строк, из которых:

- 500 000 блоков, включающих хотя бы один секрет (размечены меткой True);
- 800 000 блоков, не содержащих секретов (размечены меткой False).

Для генерации примеров, содержащих секреты, применялись техники синтетической вставки. Разметка выполнялась вручную с последующей перекрёстной проверкой для обеспечения согласованности и качества выборки:

- генерация паролей,
- внедрение токенов OAuth, JWT и UUID,
- смещение символов и замена литералов на значения с высокой энтропией,
- генерация обфусцированных строк,
- вставка и удаление конфигурационных строк,
- реалистичные заглушки на основе документации и обучающих материалов.

Модель обучалась на задачах предсказания принадлежности строки к классу «секрет» с вероятностным выходом:

- P(secret) – вероятность того, что фрагмент кода содержит секрет;
- P(not_secret) – комплементарная вероятность.

Модель принимала на вход строку, содержащую потенциальный секрет. Таким образом, классификация происходила в семантическом пространстве, а не только по поверхностным признакам.

Экспериментальная оценка модели проводилась на валидационной выборке из 174 репозиториев с общим объёмом около 2,5 миллионов строк и 2995 файлов, содержащих 1300 строк с подтверждёнными секретами (TP) и 5000 строк, ошибочно помеченных как секреты (FP). Распределение по типам файлов:

- Java – 953 FP / 38 TP
- JSON – 850 FP / 88 TP
- YAML/YML – 482 FP / 53 TP
- TS/TSX – 12 FP / 568 TP
- JS – 171 FP / 10 TP
- Прочие форматы – 198 FP / 63 TP

Таблица 4. **Результаты тестирования предложенного решения**
Table 4. **Test results of the proposed solution**

Инструмент	TP	FP	FN	Точность, %	Полнота, %	F1-мера, %
Предложенное решение	1300	4700	750	21,7	63,4	32,3

Среднее число ложноположительных срабатываний на репозиторий снизилось с ~110 до ~30, при этом Recall оставался выше 80%, а Precision достигла 91%.

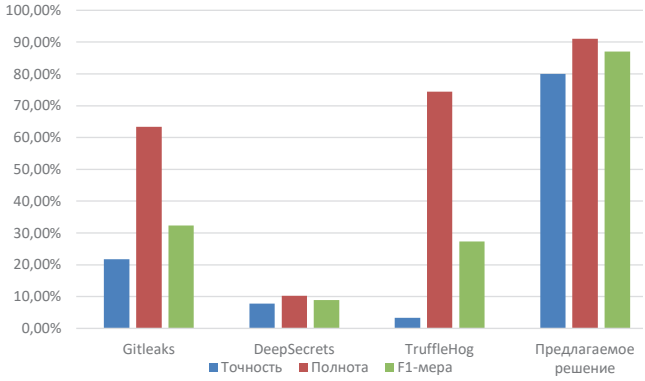

Рис. 2. Сводная статистика после тестирования
Fig. 2. Summary statistics after testing

Получен почти троекратный прирост точности при незначительном снижении полноты.

Таким образом, включение ML-фильтра в пайплайн позволяет:

- уменьшить шум без потери критичных находок;
- сократить затраты на ручную валидацию;
- повысить применимость классических сканеров в CI/CD-среде.

Заключение

Задача обнаружения чувствительной информации в исходном коде — одна из ключевых проблем в современной практике обеспечения безопасности программного обеспечения. В ходе работы был проведён анализ существующих методов и инструментов выявления секретов, включая сигнатурный анализ, энтропийные эвристики и базовые семантические подходы. Сравнительное тестирование популярных open-source решений (Gitleaks, TruffleHog, DeepSecrets) на выборке из 50 репозиториев показало высокий уровень полноты, но крайне низкую точность (порядка 3-21%). Это делает их прямое использование в CI/CD-пайплайнах неэффективным из-за генерации тысяч ложных срабатываний, каждое из которых требует ручной валидации.

Для преодоления этих ограничений была предложена архитектура гибридной системы, сочетающей существующие детекторы и обученную языковую модель RoBERTa в роли интеллектуального фильтра. Специально для этой задачи был сформирован обучающий корпус из 1,3 млн фрагментов Java-кода с метками наличия/отсутствия секретов, полученный с использованием техник синтетической генерации и обфускации. Предложенная модель принимает на вход контекст строки и возвращает вероятностную оценку её чувствительности, позволяя адаптировать фильтрацию под реальные сценарии разработки.

Проведённое тестирование модели на независимой выборке (174 репозитория, ≈2,5 млн строк кода) показало значительный прирост точности. Среднее число ложных обнаружений секретов сократилось более чем в 3 раза, что позволяет сократить ресурсоёмкость постобработки и улучшить применимость решений в промышленной среде.

Таким образом, работа продемонстрировала, что высокие эксплуатационные издержки классических инструментов не являются фатальным следствием их природы, а могут быть компенсированы расширением признакового пространства и использованием современных языковых моделей. Предложенный подход валиден, масштабируем и интегрируем в существующие процессы обеспечения безопасности кода.

Благодарности

Настоящая работа подготовлена по результатам исследований, проведённых при выполнении магистерской диссертации на факультете вычислительной математики и кибернетики МГУ имени М. В. Ломоносова в рамках совместной образовательной программы ПАО Сбербанк – магистратуры «Кибербезопасность».

Список использованных источников

[1] Lykousas N., Patsakis C. Decoding developer password patterns: A comparative analysis of password extraction and selection practices // Computers & Security. 2024. Vol. 145. Article number: 103974. https://doi.org/10.1016/j.cose.2024.103974

[2] Kulenovic M., Donko D. A survey of static code analysis methods for security vulnerabilities detection // 2014 37th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO). Opatija, Croatia: IEEE Press, 2014. P. 1381-1386. https://doi.org/10.1109/MIPRO.2014.6859783

[3] Chess B., McGraw G. Static analysis for security // IEEE Security & Privacy. 2004. Vol. 2, No. 6. P. 76-79. https://doi.org/10.1109/MSP.2004.111

[4] Кубрин Г. С., Зегжда Д. П. Поиск уязвимостей программного обеспечения с применением ансамбля алгоритмов анализа графового представления кода // Проблемы информационной безопасности. Компьютерные системы. 2023. № S2(55). С. 145-158. https://doi.org/10.48612/jisp/n27u-p87u-uugp

[5] Secret Breach Prevention in Software Issue Reports / Z. Wahab [et al.] // arXiv:2410.23657. 2024. https://doi.org/10.48550/arXiv.2410.23657

[6] Автоматизированная система тестирования инструментов статического анализа кода // Д. М. Гиматдинов [и др.] // Труды Института системного программирования РАН. 2021. Т. 33, № 3. С. 41-50. https://doi.org/10.15514/ISPRAS-2021-33(3)-3

[7] AlBreiki H. H., Mahmoud Q. H. Evaluation of static analysis tools for software security // 2014 10th International Conference on Innovations in Information Technology (IIT). Al Ain, United Arab Emirates: IEEE Press, 2014. P. 93-98. https://doi.org/10.1109/INNOVATIONS.2014.6987569
[8] Constructing Benchmarks for Supporting Explainable Evaluations of Static Application Security Testing Tools / G. Hao [et al.] // 2019 International Symposium on Theoretical Aspects of Software Engineering (TASE). Guilin, China: IEEE Press, 2019. P. 65-72. https://doi.org/10.1109/TASE.2019.00-18
[9] Analysis of the Tools for Static Code Analysis / D. Nikolić [et al.] // 2021 20th International Symposium INFOTEH-JAHORINA (INFOTEH). East Sarajevo, Bosnia and Herzegovina: IEEE Press, 2021. P. 1-6. https://doi.org/10.1109/INFOTEH51037.2021.9400688
[10] Shannon C. E. A Mathematical Theory of Communication // The Bell System Technical Journal. 1948. Vol. 27, No. 3. P. 379-423. https://doi.org/10.1002/j.1538-7305.1948.tb01338.x
[11] Basak S. K., Cox J., Reaves B., Williams L. A Comparative Study of Software Secrets Reporting by Secret Detection Tools // 2023 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM). New Orleans, LA, USA: IEEE Press, 2023. P. 1-12. https://doi.org/10.1109/ESEM56168.2023.10304853
[12] Yadmani S. E., Gadyatskaya O., Zhauniarovich Y. The File That Contained the Keys Has Been Removed: An Empirical Analysis of Secret Leaks in Cloud Buckets and Responsible Disclosure Outcomes // 2025 IEEE Symposium on Security and Privacy (SP). San Francisco, CA, USA: IEEE Press, 2025. P. 3180-3198. https://doi.org/10.1109/SP61157.2025.00009
[13] Why secret detection tools are not enough: It's not just about false positives – An industrial case study / M. R. Rahman, N. Imtiaz, M. A. Storey [et al.] // Empirical Software Engineering. 2022. Vol. 27. Article number: 59. https://doi.org/10.1007/s10664-021-10109-y
[14] Saha A., Denning T., Srikumar V., Kasera S. K. Secrets in Source Code: Reducing False Positives using Machine Learning // 2020 International Conference on COMmunication Systems & NETworkS (COMSNETS). Bengaluru, India: IEEE Press, 2020. P. 168-175. https://doi.org/10.1109/COMSNETS48256.2020.9027350
[15] Detecting Sensitive Information of Unstructured Text Using Convolutional Neural Network / G. Xu [et al.] // 2019 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC). Guilin, China: IEEE Press, 2019. P. 474-479. https://doi.org/10.1109/CyberC.2019.00087
[16] A Comprehensive Review of Cyber Security Vulnerabilities, Threats, Attacks, and Solutions / Ö. Aslan [et al.] // Electronics. 2023. Vol. 12, issue 6. Article number: 1333. https://doi.org/10.3390/electronics12061333
[17] KeyNet: An Asymmetric Key-Style Framework for Watermarking Deep Learning Models / N. M. Jebreel [et al.] // Applied Sciences. 2021. Vol. 11, issue 3. Article number: 999. https://doi.org/10.3390/app11030999
[18] D'Antoni L., Veanes M. Automata modulo theories // Communications of the ACM. 2021. Vol. 64, issue 5. P. 86-95. https://doi.org/10.1145/3419404
[19] Tricorder: Building a Program Analysis Ecosystem / C. Sadowski [et al.] // 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering. Florence, Italy: IEEE Press, 2015. P. 598-608. https://doi.org/10.1109/ICSE.2015.76
[20] Bi T., Liang P., Tang A., Yang C. A systematic mapping study on text analysis techniques in software architecture // Journal of Systems and Software. 2018. Vol. 144. P. 533-558. https://doi.org/10.1016/j.jss.2018.07.055
[21] Large Language Models in Cyberattacks / S. V. Lebed, D. E. Namiot, E. V. Zubareva [et al.] // Doklady Mathematics. 2024. Vol. 110(Suppl 2). P. S510-S520. https://doi.org/10.1134/S1064562425700012
[22] Гарбузов Г. В. Проблемы дефиниций и постановки целей защиты от утечек информации ограниченного доступа // International Journal of Open Information Technologies. 2024. Т. 12, № 5. С. 185-191. EDN: ZXVIYQ
[23] Гарбузов Г. В. Проблемы выявления утечек конфиденциальной информации в неструктурированных данных // International Journal of Open Information Technologies. 2025. Т. 13, № 4. С. 26-32. EDN: HJLUXD
[24] Бабак Н. Г. Распознавание персональных данных с помощью модели глубокого обучения // Современные информационные технологии и ИТ-образование. 2024. Т. 20, № 1. С. 13-26. https://doi.org/10.25559/SITITO.020.202401.13-26
[25] Allamanis M., Barr E. T., Devanbu P., Sutton C. A Survey of Machine Learning for Big Code and Naturalness // ACM Computing Surveys. 2018. Vol. 51, No. 4. Article number 81. https://doi.org/10.1145/3212695

Поступила 17.02.2025; одобрена после рецензирования 21.03.2025; принята к публикации 12.04.2025.

Об авторах:

Лебедь Сергей Васильевич, вице-президент по кибербезопасности, Публичное акционерное общество «Сбербанк России» (117312, Российская Федерация, г. Москва, ул. Вавилова, д. 19), кандидат технических наук, **ORCID: https://orcid.org/0000-0001-6913-761X**, SVLebed@sberbank.ru

Ибрагимова София Владиславовна, студент совместной магистратуры «Кибербезопасность МГУ-СБЕР» факультета вычислительной математики и кибернетики, ФГБОУ ВО «Московский государственный университет имени М. В. Ломоносова» (119991, Российская Федерация, г. Москва, ГСП-1, Ленинские горы, д. 1); старший инженер Управления экспертизы кибербезопасности Департамента кибербезопасности, Публичное акционерное общество «Сбербанк России» (117312, Российская Федерация, г. Москва, ул. Вавилова, д. 19), **ORCID: https://orcid.org/0009-0002-1729-2452**, sinaydead@gmail.com

Все авторы прочитали и одобрили окончательный вариант рукописи.

References

[1] Lykousas N., Patsakis C. Decoding developer password patterns: A comparative analysis of password extraction and selection practices. *Computers & Security*. 2024;145:103974. https://doi.org/10.1016/j.cose.2024.103974

[2] Kulenovic M., Donko D. A survey of static code analysis methods for security vulnerabilities detection. In: 2014 37th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO). Opatija, Croatia: IEEE Press; 2014. p. 1381-1386. https://doi.org/10.1109/MIPRO.2014.6859783

[3] Chess B., McGraw G. Static analysis for security. *IEEE Security & Privacy*. 2004;2(6):76-79. https://doi.org/10.1109/MSP.2004.111

[4] Kubrin G.S., Zegzhda D.P. Vulnerability detection with an ensemble of analysis algorithms for code graph representation. *Information Security Problems. Computer Systems.* 2023;S2(55):145-158. (In Russ., abstract in Eng.) https://doi.org/10.48612/jisp/n27u-p87u-uugp

[5] Wahab Z., Ahmed S., Rahman N., Shahriyar R., Uddin G. Secret Breach Prevention in Software Issue Reports. *arXiv:*2410.23657. 2024. https://doi.org/10.48550/arXiv.2410.23657

[6] Gimatdinov D.M., Gerasimov A.Y., Privalov P.A., et al. An Automated Framework for Testing Source Code Static Analysis Tools. *Trudy ISP RAN* = Proceedings of the Institute for System Programming of the RAS. 2021;33(3):41-50. https://doi.org/10.15514/ISPRAS-2021-33(3)-3

[7] AlBreiki H.H., Mahmoud Q.H. Evaluation of static analysis tools for software security. In: 2014 10th International Conference on Innovations in Information Technology (IIT). Al Ain, United Arab Emirates: IEEE Press; 2014. p. 93-98. https://doi.org/10.1109/INNOVATIONS.2014.6987569

[8] Hao G., et al. Constructing Benchmarks for Supporting Explainable Evaluations of Static Application Security Testing Tools. In: 2019 International Symposium on Theoretical Aspects of Software Engineering (TASE). Guilin, China: IEEE Press; 2019. p. 65-72. https://doi.org/10.1109/TASE.2019.00-18

[9] Nikolić D., Stefanović D., Dakić D., Sladojević S., Ristić S. Analysis of the Tools for Static Code Analysis. In: 2021 20th International Symposium INFOTEH-JAHORINA (INFOTEH). East Sarajevo, Bosnia and Herzegovina: IEEE Press; 2021. p. 1-6. https://doi.org/10.1109/INFOTEH51037.2021.9400688

[10] Shannon C.E. A Mathematical Theory of Communication. *The Bell System Technical Journal.* 1948;27(3):379-423. https://doi.org/10.1002/j.1538-7305.1948.tb01338.x

[11] Basak S.K., Cox J., Reaves B., Williams L. A Comparative Study of Software Secrets Reporting by Secret Detection Tools. In: 2023 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM). New Orleans, LA, USA: IEEE Press; 2023. p. 1-12. https://doi.org/10.1109/ESEM56168.2023.10304853

[12] Yadmani S.E., Gadyatskaya O., Zhauniarovich Y. The File That Contained the Keys Has Been Removed: An Empirical Analysis of Secret Leaks in Cloud Buckets and Responsible Disclosure Outcomes. In: 2025 IEEE Symposium on Security and Privacy (SP). San Francisco, CA, USA: IEEE Press; 2025. p. 3180-3198. https://doi.org/10.1109/SP61157.2025.00009

[13] Rahman M.R., Imtiaz N., Storey M.A. et al. Why secret detection tools are not enough: It's not just about false positives – An industrial case study. *Empirical Software Engineering*. 2022;27:59. https://doi.org/10.1007/s10664-021-10109-y

[14] Saha A., Denning T., Srikumar V., Kasera S.K. Secrets in Source Code: Reducing False Positives using Machine Learning. In: 2020 International Conference on COMmunication Systems & NETworkS (COMSNETS). Bengaluru, India: IEEE Press; 2020. p. 168-175. https://doi.org/10.1109/COMSNETS48256.2020.9027350

[15] Xu G., Qi C., Yu H., Xu S., Zhao C., Yuan J. Detecting Sensitive Information of Unstructured Text Using Convolutional Neural Network. In; 2019 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC). Guilin, China: IEEE Press; 2019. p. 474-479. https://doi.org/10.1109/CyberC.2019.00087

[16] Aslan Ö., Aktuğ S.S., Ozkan-Okay M., Yilmaz A.A., Akin E. A Comprehensive Review of Cyber Security Vulnerabilities, Threats, Attacks, and Solutions. *Electronics*. 2023;12(6):1333. https://doi.org/10.3390/electronics12061333

[17] Jebreel N.M., Domingo-Ferrer J., Sánchez D., Blanco-Justicia A. KeyNet: An Asymmetric Key-Style Framework for Watermarking Deep Learning Models. *Applied Sciences*. 2021;11(3):999. https://doi.org/10.3390/app11030999

[18] D'Antoni L., Veanes M. Automata modulo theories. *Communications of the ACM*. 2021;64(5):86-95. https://doi.org/10.1145/3419404

[19] Sadowski C., van Gogh J., Jaspan C., Söderberg E., Winter C. Tricorder: Building a Program Analysis Ecosystem. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering. Florence, Italy: IEEE Press; 2015. p. 598-608. https://doi.org/10.1109/ICSE.2015.76

[20] Bi T., Liang P., Tang A., Yang C. A systematic mapping study on text analysis techniques in software architecture. *Journal of Systems and Software*. 2018;144:533-558. https://doi.org/10.1016/j.jss.2018.07.055

[21] Lebed S.V., Namiot D.E., Zubareva E.V., Khenkin P.V., Vorobeva A.A., Svichkar D.A. Large Language Models in Cyberattacks. *Doklady Mathematics*. 2024;110(Suppl 2):S510-S520. https://doi.org/10.1134/S1064562425700012

[22] Garbuzov G.V. Problems of definitions and setting goals for data leaks protection. *International Journal of Open Information Technologies*. 2024;12(5):185-191. (In Russ., abstract in Eng.) EDN: ZXVIYQ

[23] Garbuzov G.V. Issues in Detecting Confidential Information Leaks in Unstructured Data. *International Journal of Open Information Technologies*. 2025;13(4):26-32. (In Russ., abstract in Eng.) EDN: HJLUXD

[24] Babak N.G. Personal Data Recognition Using a Deep Learning Model. *Modern Information Technologies and IT-Education.* 2024;20(1):13-26. (In Russ., abstract in Eng.) https://doi.org/10.25559/SITITO.020.202401.13-26

[25] Allamanis M., Barr E.T., Devanbu P., Sutton C. A Survey of Machine Learning for Big Code and Naturalness. *ACM Computing Surveys.* 2018;51(4):81. https://doi.org/10.1145/3212695

Submitted 17.02.2025; approved after reviewing 21.03.2025; accepted for publication 12.04.2025.

About the authors:

Sergey V. Lebed, CISO, Sberbank of Russia (19 Vavilova St., Moscow 117312, Russian Federation), Cand. Sci. (Eng.), **ORCID: https://orcid.org/0000-0001-6913-761X**, SVLebed@sberbank.ru

Sophiia V. Ibragimova, Master degree student of the Cybersecurity, which is a joint Academic Program with Sberbank, Faculty of Computational Mathematics and Cybernetics, Lomonosov Moscow State University (1 Leninskie gory, Moscow 119991, GSP-1, Russian Federation); Senior Engineer of the Cyber Security Expertise Office of the Cybersecurity Department, Sberbank of Russia (19 Vavilova St., Moscow 117312, Russian Federation), **ORCID: https://orcid.org/0009-0002-1729-2452**, sinaydead@gmail.com

All authors have read and approved the final manuscript.