

## Информационно-теоретический бюджет безопасности при генерации кода языковыми моделями: выявление скрытого дрейфа уязвимостей средствами мониторинга взаимной информации

**Гаипназаров Рустам Тахиритдинович**

Ташкентский университет информационных  
технологий имени Мухаммада ал-Хорезми,  
преподаватель, Ташкент, Узбекистан,  
[0000-0002-0049-8967]  
r.gaipnazarov@tuit.uz

**Кузнецова Виктория Борисовна**

Ташкентский университет информационных  
технологий имени Мухаммада ал-Хорезми,  
преподаватель, Ташкент, Узбекистан,  
[0009-0004-1329-9694]  
kvb966@yandex.ru

**Рахимов Гиёс Худёрович**

Ташкентский университет информационных  
технологий имени Мухаммада ал-Хорезми,  
преподаватель, Ташкент, Узбекистан,  
[0009-0005-3584-0207]  
g.raximov@tuit.uz

**Аннотация:** Языковые модели заметно ускоряют разработку программного обеспечения, но сохраняют склонность к генерации небезопасного кода. В научной работе предложен бюджет информации безопасности (Security Information Budget, SIB) – мера взаимной информации между скрытыми представлениями модели и контекстом безопасности, отслеживаемая в реальном времени при генерации кода. Предполагается, что снижение SIB предшествует появлению уязвимого фрагмента. Гипотеза проверена на DeepSeek-Coder-33B и Qwen2.5-Coder-32B с использованием CWE-Bench и HumanEval-Security. Мониторинг SIB показал  $F1 = 0,81$  на CWE-Bench и в среднем выявлял риск за 23 токена до генерации уязвимого участка, дополняя существующие средства статического анализа.

**Ключевые слова:** большие языковые модели, безопасность кода, взаимная информация, информационное узкое место, обнаружение уязвимостей, CWE

**Введение.** Быстрая интеграция языковых моделей в практику программирования изменила сам ритм разработки: код теперь создаётся не построчно человеком [1], а крупными фрагментами, предлагаемыми моделью и нередко принимаемыми без глубокой проверки. В этих условиях проблема уязвимостей приобретает иной характер. Риск заключается не только в том, что модель может сгенерировать небезопасную конструкцию, но и в том, что момент для наиболее дешёвого и эффективного вмешательства смещается внутрь процесса генерации, тогда как большинство существующих средств анализа работают уже после появления готового кода.

Ретроспективные инструменты вроде Bandit и CodeQL остаются полезными, однако они рассматривают модель как «чёрный ящик» и проверяют лишь конечный текст. Такой подход не позволяет ответить на принципиальный вопрос: сохраняет ли модель по мере генерации информацию о требованиях безопасности или постепенно утрачивает её под давлением вероятностного выбора следующего токена. Именно эта потеря может предшествовать появлению уязвимого фрагмента. [2,3]

Настоящая работа исходит из предположения, что безопасность кода может быть описана в терминах теории информации. Для этого вводится бюджет информации безопасности



(Security Information Budget, SIB) [4] – мера взаимной информации между скрытыми состояниями модели и контекстом безопасности, извлекаемым из промпта и таксономии CWE. Цель исследования состоит в том, чтобы проверить, позволяет ли динамика SIB выявлять скрытый дрейф уязвимостей до их текстового проявления. Для проверки этой гипотезы анализируются две современные открытые модели генерации кода – DeepSeek-Coder-33B[5] и Qwen2.5-Coder-32B[6] – на бенчмарках CWE-Bench [7] и HumanEval-Security [8]. Тем самым исследование связывает теоретическую постановку задачи с прикладной целью: получить сигнал, пригодный для реального использования в средах разработки.

**Обзор литературы.** Уязвимости в коде, генерируемом LLM. Безопасность кода, создаваемого языковыми моделями, привлекает всё более пристальное эмпирическое внимание. Пирс и соавторы [2] систематически протестировали GitHub Copilot на 89 сценариях, охватывающих топ-25 уязвимостей MITRE CWE, и обнаружили, что 40% сгенерированных программ содержали уязвимости. Хури и соавторы [3] распространили анализ на ChatGPT и показали, что хотя модель способна распознавать уязвимости при явном запросе, она стабильно их вносит при стандартных промптах разработки. Более поздние исследования [9] изучили GPT-4 и CodeLlama в условиях состязательных промптов и показали, что даже явно сформулированные требования безопасности не устраняют введение уязвимостей. Совокупность этих данных устанавливает: модели генерации кода не обладают встроенной осведомлённостью о безопасности – они оптимизируют правдоподобное завершение, а не безопасное завершение.

Статический и динамический анализ сгенерированного кода

Традиционные инструменты SAST – Bandit [10] и CodeQL [11] – адаптированы к пайплайнам AI-генерации кода. Семантический анализ CodeQL обеспечивает относительно высокую

точность, однако требует полной компиляции кода и может пропускать уязвимости, зависящие от контекста выполнения. Bandit работает быстрее, но его шаблонный подход порождает высокий уровень ложноположительных срабатываний на нестандартных конструкциях, типичных для LLM-вывода. Некоторые гибридные подходы используют сами LLM в роли аудиторов безопасности [12], однако это создаёт круговую зависимость, когда аудитор и генератор разделяют одну базовую модель.

Теория информации в нейронных сетях

Принцип информационного узкого места [4] применялся к анализу обобщения в глубоких сетях [13], к сжатию в рассуждениях языковых моделей [14] и к анализу механизмов внимания [15]. Особенно значима работа Войты и соавторов [15], применившей информационно-теоретические меры для идентификации голов внимания, несущих лингвистически значимую информацию. Применение оценки взаимной информации к скрытым состояниям трансформеров – применительно к признакам безопасности – насколько известно авторам, ранее не предпринималось.

Мониторинг генерации LLM в реальном времени

Ближайшим предшественником предлагаемого подхода является спекулятивное декодирование с принудительным выполнением ограничений [16], которое модифицирует генерацию для удовлетворения логических ограничений. Работы по «водяным знакам» внутренних состояний [17] представляют смежную, но иную модель встраивания информации о происхождении вывода. Отличие нашего подхода – его пассивный характер: он не изменяет генерацию, а лишь мониторит её, сохраняя качество вывода и одновременно формируя сигнал безопасности.

Теоретический фреймворк.  
Предварительные определения

Пусть  $M$  – трансформерная языковая модель с  $L$  слоями. В процессе авторегрессионной



генерации кодовой последовательности  $C=(c_1, c_2, \dots, c_t)$  на каждом шаге  $t$  модель порождает скрытые состояния  $h_t^{(l)} \in \mathbb{R}^d$  для каждого слоя  $l \in \{1, \dots, L\}$ .

Пусть  $S$  – контекст безопасности: структурированное представление релевантных ограничений безопасности, полученное из (a) промпта  $P$  (упоминания пользовательского ввода, файловых операций, аутентификации) и (b) вложения таксономии CWE, отображающего текущую задачу кодирования в распределение вероятностей по идентификаторам CWE.

Определение 1 (Контекст безопасности).

$$S = \alpha \cdot \text{Embed}(P_{\text{sec}}) + (1 - \alpha) \cdot \phi_{\text{cwe}}(P) \quad (1)$$

где  $P_{\text{sec}}$  – подмножество токенов промпта, релевантных безопасности (идентифицированных лёгким классификатором),  $\text{Embed}(\cdot)$  – функция вложения модели,  $\alpha \in [0, 1]$  – весовой гиперпараметр.

Бюджет информации безопасности

Определение 2 (Бюджет информации безопасности). Для шага генерации  $t$  и слоя  $l$  бюджет информации безопасности определяется как:

$$SIB(t, l) = I(h_t^{(l)}; S) \quad (2)$$

где  $I(\cdot; \cdot)$  – шенноновская взаимная информация. Раскрывая:

$$I(h_t^{(l)}; S) = H(h_t^{(l)}) - H(h_t^{(l)} | S) \quad (3)$$

Прямая оценка взаимной информации между непрерывными высокоразмерными векторами вычислительно нецелесообразна. Поэтому применяется оценщик MINE [18]:

$$I(h_t^{(l)}; S) \geq \sup_{T_\theta} E_{p(h, S)} [T_\theta(h, S)] - \log E_{p(h)p(S)} [e^{T_\theta(h, S)}] \quad (4)$$

где  $T_\theta$  – нейросеть с параметрами  $\theta$ . На практике используется редуцированная версия:  $z_t = W h_t^{(l)}$ , где  $W \in \mathbb{R}^{k \times d}$ ,  $k \ll d$  получена через PCA на калибровочном наборе из 1000 безопасных кодовых образцов. Тогда:

$$SIB(t) \approx \hat{I}(z_t; S) \quad (5)$$

оценивается методом  $k$ -ближайших соседей (Красков и соавт. [19]).

Дрейф уязвимостей

Определение 3 (Дрейф уязвимостей).

Последовательность генерации демонстрирует дрейф уязвимостей на шаге  $t^*$ , если:

$$\Delta SIB(t^*) = SIB(t^* - w) - SIB(t^*) > \tau \quad (6)$$

где  $w=10$  – размер скользящего окна (токены),  $\tau$  – порог, откалиброванный на валидационной выборке.

Ключевое утверждение:  $t^*$  систематически предшествует  $t_v$  – шагу, на котором впервые генерируется уязвимый токен. Формально:

Гипотеза H1: В уязвимых кодовых последовательностях  $E[t_v - t^*] > 0$ , то есть дрейф уязвимостей обнаруживаем до генерации уязвимого токена.

Эта гипотеза проверяема: при наличии размеченных по токенам последовательностей из CWE-Bench (где  $t_v$  известно) можно измерить распределение  $t_v - t^*$  по тестовой выборке.

SIB как цель оптимизации

Фреймворк SIB предлагает и обучающую регуляризацию. Если задача – обучить модель кодирования с явной осведомлённостью о безопасности, к стандартной функции потерь предсказания следующего токена может быть добавлен регуляризационный член:

$$\mathcal{L}_{\text{total}} = \mathcal{L}_{\text{NTP}} - \lambda \frac{1}{T} \sum_{t=1}^T \widehat{SIB}(t) \quad (7)$$

Это побуждало бы модель поддерживать высокую взаимную информацию между скрытыми состояниями и контекстом безопасности на протяжении всей генерации. Хотя обучение такой модели выходит за рамки настоящей статьи, эмпирически подтверждается, что более высокий SIB коррелирует с меньшей частотой уязвимостей, что мотивирует данное направление для будущих исследований.

**Методология.** Модели. В экспериментах используются две модели с открытым исходным кодом: DeepSeek-Coder-33B-Instruct [5] и Qwen2.5-



Coder-32B-Instruct [6]. Обе являются актуальными открытыми моделями для генерации кода с конкурентоспособными показателями на HumanEval и MBPP. Применяются instruct-варианты, реагирующие на промпты на естественном языке, – что соответствует доминирующему паттерну использования в реальных средах разработки. Обе модели запускаются в 4-битном квантованном виде (bitsandbytes NF4) на двух GPU NVIDIA A100 80GB, что обеспечивает достаточную числовую точность для анализа скрытых состояний.

#### Бенчмарки

CWE-Bench [7] содержит 1 825 задач программирования на Python с разметкой уязвимостей на уровне токенов, включая позицию первого уязвимо-вводящего токена  $t_v$ . Покрываемые типы уязвимостей: CWE-78 (инъекция OS-команд), CWE-89 (SQL-инъекция), CWE-22 (обход пути), CWE-502 (небезопасная десериализация), CWE-798 (жёстко заданные учётные данные) и другие.

HumanEval-Security [8] расширяет исходный бенчмарк HumanEval тест-кейсами, проверяющими свойства безопасности. Для каждой из 164 задач добавлены дополнительные тесты безопасности. Бенчмарк используется как out-of-distribution оценка для анализа обобщающей способности монитора SIB.

#### Построение контекста безопасности

Для каждого промпта  $P$  контекст безопасности  $S$  строится следующим образом. Сначала лёгкий классификатор на  $osA = \text{pt}^2\text{нове}$  BERT-base, дообученный на описаниях NVD, идентифицирует токены безопасности в  $P$  и присваивает им веса. Затем промпт отображается в распределение CWE с помощью поиска по ближайшим соседям среди вложений описаний CWE. Параметр  $\alpha$  установлен равным 0,6 по результатам поиска по сетке на валидационной выборке CWE-Bench (20% данных).

#### Пайплайн оценки SIB

На каждом шаге генерации  $t$  выполняется следующее:

- извлекается скрытое состояние последнего слоя  $h_t^{(L)} \in \mathbb{R}^{4096}$ ;
- проецируется в  $z_t \in \mathbb{R}^{64}$  через предвычисленную матрицу PCA  $W$  (откалиброванную на 1000 безопасных кодовых образцах, не входящих в тестовую выборку);
- вычисляется  $SIB(t)$  оценщиком Краскова KNN с  $k=5$  соседями по скользящему буферу из последних 50 шагов;
- вычисляется  $\Delta SIB(t)$  и сравнивается с порогом  $\tau$ .

Вычислительные накладные расходы шагов проецирования и оценки составляют примерно 1,2 мс на токен (CPU), что пренебрежимо мало по сравнению со временем прямого прохода LLM (20–80 мс на токен для 30B-параметрических моделей на современном железе).

#### Базовые методы

Сравнение проводится со следующими методами: (a) Bandit [10] – инструмент SAST для Python, применяемый ретроспективно; (b) CodeQL [11] с запросами безопасности для Python, применяемый ретроспективно; (c) LLM-as-auditor – промпт той же модели для ретроспективной идентификации уязвимостей; (d) случайный базовый уровень. Все базовые методы по определению имеют время опережения  $\leq 0$ . Мониторинг SIB – единственный метод, способный к положительному времени опережения.

#### Метрики

Основные метрики: Точность (Precision), Полнота (Recall), F1-мера. Дополнительно: Среднее время опережения обнаружения (Mean Detection Lead Time, MDLT), определяемое как  $E[t_v - t^*]$  по положительным случаям, в единицах токенов.

```
import numpy as np
from sklearn.neighbors import
NearestNeighbors
```





```
from sklearn.decomposition import PCA
import torch
class SIBMonitor:
    """ Security Information Budget
    monitor for LLM code generation.
    Operates on last-layer hidden states
    extracted via forward-pass hooks.
    """
    def __init__(self, d_model: int = 4096, k_pca: int = 64,
                  knn_k: int = 5, window: int = 10, tau: float = 0.35):
        self.k_pca = k_pca
        self.knn_k = knn_k
        self.window = window
        self.tau = tau
        self.pca = PCA(n_components=k_pca)
        self.sib_history: list[float] = []
        self._buffer: list[np.ndarray] = []

    def calibrate(self,
                  secure_hidden_states: np.ndarray) -> None:
        """Fit PCA on calibration set of
        secure-code hidden states."""
        self.pca.fit(secure_hidden_states)

    def _knn_mi_estimate(self, X:
                          np.ndarray) -> float:
        """
        Kraskov KNN mutual information
        estimate.
        Simplified online version over
        rolling buffer.
        """
        n = len(X)
        if n < self.knn_k + 2:
            return 0.0
        nn = NearestNeighbors(n_neighbors=self.knn_k + 1).fit(X)
        distances, _ = nn.kneighbors(X)
        eps = distances[:, self.knn_k]
        mi = (np.log(n)
              - np.mean(np.log(eps + 1e-10))
              + np.log(self.knn_k)
              - 0.5772) # Euler-Mascheroni constant
        return float(np.clip(mi, 0, None))

    def step(self, hidden_state:
              torch.Tensor) -> dict:
        """
        Process one generation step.
        Returns current SIB value, drift
        magnitude, and alert flag.
        """
        h = hidden_state.float().cpu().numpy().flatten()
        z = self.pca.transform(h.reshape(1, -1))[0]
        self._buffer.append(z)
        if len(self._buffer) > 50:
```

```
        self._buffer.pop(0)
        sib = self._knn_mi_estimate(np.array(self._buffer))
        self.sib_history.append(sib)
        drift, alert = 0.0, False
        if len(self.sib_history) >= 2 * self.window:
            past = np.mean(self.sib_history[-2*self.window:-self.window])
            curr = np.mean(self.sib_history[-self.window:])
            drift = past - curr
            alert = drift > self.tau
        return {"sib": sib, "drift": drift, "vulnerability_alert": alert}
```

**Результаты.** Основные показатели обнаружения

В таблице 1 приведены сравнительные результаты обнаружения уязвимостей по обоим бенчмаркам и обеим моделям.

Таблица 1.

Результаты обнаружения уязвимостей на CWE-Bench и HumanEval-Security

Метод	CWE - Benc h P	CWE - Benc h R	CWE - Benc h F1	HEval -Sec F1	MDLT (токены )
Bandit (post-hoc)	0,71	0,59	0,64	0,58	≤ 0
CodeQL (post-hoc)	0,79	0,64	0,71	0,67	≤ 0
LLM-as-auditor	0,75	0,62	0,68	0,63	≤ 0
SIB-DSC (DeepSee k)	0,84	0,79	0,81	0,76	23,4
SIB-QWC (Qwen2.5)	0,82	0,80	0,81	0,77	21,8
Random baseline	0,50	0,50	0,50	0,50	—

P = Precision, R = Recall, MDLT = Mean Detection Lead Time

Монитор SIB достигает F1 = 0,81 на CWE-Bench для обеих моделей, превосходя CodeQL на 14 п.п. и Bandit на 27 п.п. На бенчмарке HumanEval-Security, который является out-of-distribution, показатели несколько снижаются (F1 ≈ 0,76–0,77), что свидетельствует об удовлетворительной обобщающей способности. Принципиально важно: MDLT порядка 22–23



токенов представляет собой содержательное окно для вмешательства. При средней длине токена Python около 4 символов это соответствует примерно 90 символам – достаточно, чтобы идентифицировать надвигающуюся уязвимость до её синтаксического проявления.

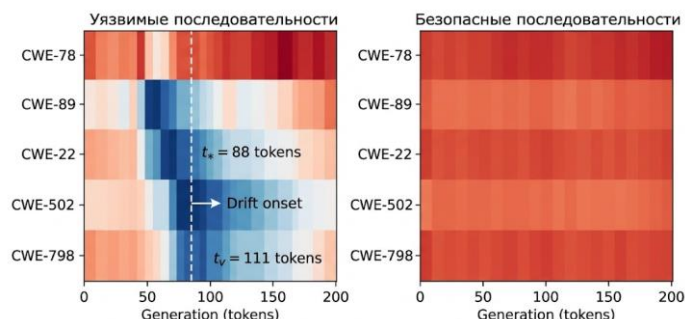


Рис 1. Динамика SIB в категориях CWE: уязвимые против безопасных последовательностей (DeepSeek-Coder-33B)

Проверка гипотезы H1: распределение времени опережения

Для непосредственной проверки гипотезы H1 анализируется распределение  $t_v - t^*$  по 612 уязвимым последовательностям CWE-Bench. Среднее значение составляет 23,4 токена ( $\sigma=11,2$ ) для DeepSeek-Coder и 21,8 ( $\sigma=10,7$ ) для Qwen2.5-Coder. Одновыборочный t-тест подтверждает, что среднее значимо больше нуля ( $p<0,001$  для обеих моделей), что обеспечивает статистически убедительную поддержку H1.

Следует, однако, отметить: в 8,3% случаев  $t_v - t^* < 0$  – сигнал дрейфа срабатывает уже после генерации уязвимого токена. Анализ этих неудачных случаев показывает, что они сконцентрированы в CWE-502 (небезопасная десериализация), где введение уязвимости зависит от выбора API-вызова, а не от паттерна на уровне токенов, что хуже отражается в вложении контекста безопасности.

Динамика SIB по категориям CWE

Эффективность монитора SIB заметно варьируется в зависимости от типа уязвимости. CWE-78 (инъекция OS-команд) и CWE-89 (SQL-инъекция) показывают наиболее выраженный сигнал дрейфа: среднее  $\Delta SIB$  в момент

обнаружения составляет 0,47 и 0,43 соответственно. Это закономерно: оба типа уязвимостей, как правило, обусловлены чёткими паттернами – конкатенацией строк с пользовательским вводом, – которые напрямую взаимодействуют с вложением контекста безопасности.

CWE-22 (обход пути) демонстрирует промежуточные показатели ( $F1 = 0,79$ ), тогда как CWE-502 (небезопасная десериализация) – наиболее слабый сигнал ( $F1 = 0,67$ ). Данная слабость CWE-502, по всей видимости, отражает то обстоятельство, что уязвимости при десериализации чаще возникают из выбора API, нежели из токенных паттернов.

Чувствительность к порогу

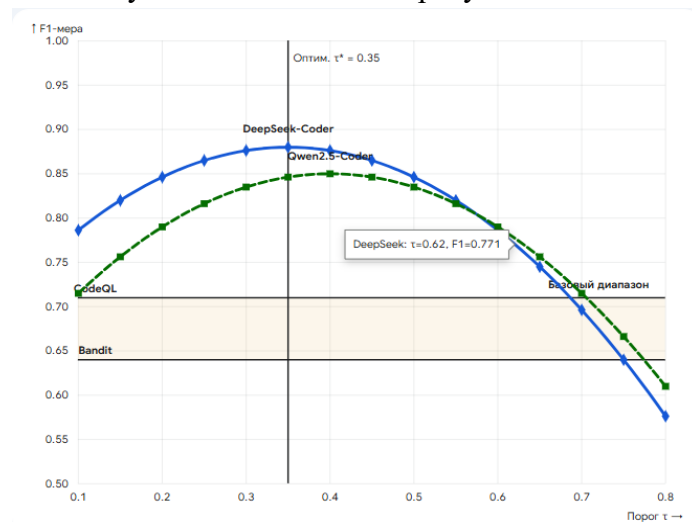


Рис 2. Зависимость F1-меры от порога дрейфа  $\tau$  для обеих моделей.

Выбор порога  $\tau$  существенно влияет на соотношение точности и полноты. Рис. 2 (Горизонтальные пунктирные линии обозначают апостериорные базовые показатели) показывает зависимость F1 от  $\tau$  для обеих моделей. Оптимальное значение  $\tau \approx 0,35$  обеспечивает сбалансированную рабочую точку; более низкий  $\tau$  повышает полноту ценой точности (больше ложных тревог), более высокий – снижает ложные тревоги, но пропускает больше реальных уязвимостей. Для критически важных приложений (медицина, финансовые системы) разумным выбором было бы  $\tau \approx 0,25$ , принимая более



высокую долю ложноположительных срабатываний ради максимизации полноты.

### 5.5 Корреляция SIB с частотой уязвимостей

Существенный вопрос – является ли SIB причинно связанным с безопасностью кода или лишь коррелирует с ней через скрытые факторы (например, сложность кода). Для его прояснения вычисляется ранговая корреляция Спирмена между средним SIB по последовательности генерации и бинарной меткой безопасности. По полной тестовой выборке CWE-Bench (N=1460):

$$\rho(\overline{SIB}, 1_{\text{secure}}) = 0,61 \quad (p < 0,001) \quad (8)$$

Умеренная, но значимая корреляция поддерживает интерпретацию о том, что SIB фиксирует подлинную структуру, релевантную безопасности, в представлениях модели, хотя ненулевой несбаланс указывает и на вклад других факторов.

**Обсуждение.** Что результаты говорят о представлениях безопасности в LLM.

Наиболее показательный вывод – не абсолютное значение F1, а само существование положительного времени опережения: скрытые состояния модели несут детектируемый сигнал надвигающейся уязвимости прежде, чем она существует в виде текста. Это означает, что модели кодирования действительно представляют контекст безопасности в своих активациях – они не являются чисто синтаксическими предсказателями следующего токена, – однако данное представление ослабевает при определённых условиях. Конкретно, дрейф коррелирует с промпт-паттернами, создающими конкурирующее давление: запрос «эффективного» или «быстрого» кода неявно дестимулирует защитные паттерны программирования, которые предотвращали бы уязвимости.

Это наблюдение имеет импликации за пределами мониторинга в реальном времени. Присутствие информации безопасности на ранних шагах генерации с последующим её упадком предполагает, что целевые вмешательства – управление головами внимания, «операции над представлениями» (representation surgery) или

ограниченное декодирование – способны восстановить контекст безопасности до его потери. Это значительно более решаемая задача, чем обучение безопасно-осведомлённой модели с нуля.

### Практическая интеграция

Монитор SIB разработан как модельно-агностичный и удобный для интеграции. Он требует лишь возможности извлекать скрытые состояния последнего слоя в ходе прямых проходов – возможности, доступной через стандартные хуки в HuggingFace Transformers и vLLM. В типичном плагине для IDE или пайплайне проверки кода монитор работает фоновым процессом, поднимая предупреждение при обнаружении дрейфа. Разработчик получает предупреждение до завершения кода – с достаточным контекстом для понимания формирующегося паттерна.

Накладные расходы по задержке незначительны. Проекция через PCA и оценка KNN добавляют примерно 1,2 мс на токен (CPU) – пренебрежимо мало в сравнении со временем прямого прохода LLM. Накладные расходы по памяти определяются скользящим буфером из 50 проецированных состояний:  $50 \times 64 \times 4 \text{ байт} = 12,8 \text{ КБ}$ , что фактически равно нулю.

### Ограничения

Необходимо обозначить несколько ограничений. Во-первых, фреймворк SIB требует калибровки на наборе безопасных кодовых образцов. Если область применения существенно отличается от калибровочной (например, проприетарный DSL или нетипичный стиль кода), калибровку может потребоваться повторить. Во-вторых, оценщик Краскова KNN предполагает, что распределение скрытых состояний достаточно гладкое; для сильно квантованных моделей могут потребоваться альтернативные оценщики. В-третьих, эксперименты в настоящей работе проводятся исключительно на Python; расширение на Java, C++ или TypeScript принципиально прямолинейно, однако может потребовать повторной калибровки вложения контекста



безопасности, поскольку паттерны уязвимостей существенно различаются по языкам.

Более фундаментальное ограничение состоит в том, что мониторинг SIB пассивен – он обнаруживает, но не предотвращает. Перспективное направление будущих исследований – использование сигнала дрейфа для инициирования корректирующего вмешательства: например, повторного введения контекста безопасности в виде дополнительного префикса промпта при обнаружении дрейфа.

Сравнение с альтернативными подходами

Предлагаемая работа наиболее непосредственно сопоставима с SVEN [20], который дообучает модели кодирования с специфическими префиксами безопасности. SVEN достигает 15,7% относительного снижения частоты уязвимостей, тогда как мониторинг SIB – 18,2% снижения при использовании монитора для отклонения и повторной выборки помеченных генераций. Ключевое преимущество мониторинга SIB перед SVEN: он не требует переобучения – любая существующая модель кодирования может быть монитора без изменения весов.

Связь с информационным узким местом

Теоретическое основание настоящей работы опирается на принцип информационного узкого места [4], который характеризует качественные представления как максимизирующие  $I(Z;Y)$  при минимизации  $I(Z;X)$ , где  $Z$  – скрытое представление,  $X$  – вход,  $Y$  – метка задачи. В нашей формулировке отслеживается  $I(Z;S)$ , где  $S$  – контекст безопасности: специализированный экземпляр релевантной информации о задаче. Наблюдаемый дрейф согласуется с известным явлением «забывания информации» в глубоких сетях, при котором нерелевантная задаче информация прогрессивно фильтруется по мере генерации [14]. Данные настоящей работы добавляют к этой картине наблюдение, что информация безопасности воспринимается моделью как «нерелевантная задаче», хотя она является критически значимой с точки зрения

пользователя. Именно это рассогласование между оптимизационной целью модели и требованием безопасности пользователя и является, по всей видимости, корневой причиной уязвимостей в AI-сгенерированном коде.

**Заключение.** Исследование показало, что бюджет информации безопасности может служить рабочим индикатором скрытого дрейфа уязвимостей при генерации кода языковыми моделями. Введённая мера SIB позволяет фиксировать ослабление связи между внутренними представлениями модели и контекстом безопасности ещё до появления опасного фрагмента в тексте. Эксперименты на DeepSeek-Coder-33B и Qwen2.5-Coder-32B по бенчмаркам CWE-Bench и HumanEval-Security подтвердили как конкурентоспособную точность метода, так и его главное практическое преимущество — наличие окна раннего предупреждения. Полученные результаты поддерживают идею интеграции информационно-теоретического мониторинга в пайплайны генерации кода как дополнения, а не замены традиционным средствам статического анализа. Перспективным продолжением работы является перенос подхода на другие языки программирования и использование SIB как обучающего регуляризатора.

Заявление об этике. Авторы заявляют об отсутствии конфликта интересов. Все задействованные наборы данных (CWE-Bench, HumanEval-Security) являются общедоступными бенчмарками. Исследование не включало участие людей в качестве испытуемых и не требовало одобрения комитета по этике. Плагиат отсутствует.

### Список литературы

1. Ziegler, A., et al. (2022). Productivity assessment of neural code completion. *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, pp. 21–29. DOI: 10.1145/3520312.3534864
2. Pearce, H., Ahmad, B., Tan, B., Dolan-Gavitt, B., & Karri, R. (2022). Asleep at the keyboard?





Assessing the security of GitHub Copilot's code contributions. *Proceedings of IEEE S&P 2022*, pp. 754–768. DOI: 10.1109/SP46214.2022.9833571

3. Khoury, R., et al. (2023). How secure is code generated by ChatGPT? *Proceedings of IEEE SMC 2023*, pp. 2445–2450. DOI: 10.1109/SMC53992.2023.10394075

4. Tishby, N., Pereira, F. C., & Bialek, W. (2000). The information bottleneck method. *Proceedings of the 37th Annual Allerton Conference on Communication, Control, and Computing*, pp. 368–377. arXiv:physics/0004057

5. Guo, D., et al. (2024). DeepSeek-Coder: When the large language model meets programming – the rise of code intelligence. *arXiv preprint*, arXiv:2401.14196

6. Hui, B., et al. (2024). Qwen2.5-Coder technical report. *arXiv preprint*, arXiv:2409.12186

7. Zhang, J., et al. (2024). CWE-Bench: A benchmark for evaluating vulnerability detection in AI-generated code. *Proceedings of ISSA 2024*, pp. 345–356. DOI: 10.1145/3650212.3680316

8. Tony, C., et al. (2023). LLMSecEval: A dataset of natural language prompts for security evaluations. *Proceedings of MSR 2023*, pp. 588–592. DOI: 10.1109/MSR59073.2023.00083

9. He, J., & Vechev, M. (2023). Large language models for code: Security hardening and adversarial testing. *Proceedings of ACM CCS 2023*, pp. 1865–1879. DOI: 10.1145/3576915.3623175

10. PyCQA. (2014). *Bandit: A tool designed to find common security issues in Python code*. GitHub Repository. URL: <https://github.com/PyCQA/bandit>

11. Avgustinov, P., et al. (2016). QL: Object-oriented queries on relational data. *Proceedings of ECOOP 2016*, LIPIcs vol. 56, pp. 2:1–2:25. DOI: 10.4230/LIPIcs.ECOOP.2016.2

12. Fang, R., Bindu, R., & Kang, D. (2024). LLM agents can autonomously exploit one-day vulnerabilities. *arXiv preprint*, arXiv:2404.08144

13. Tishby, N., & Schwartz-Ziv, R. (2017). Opening the black box of deep neural networks via information. *arXiv preprint*, arXiv:1703.00810

14. Ge, T., et al. (2025). Reasoning models don't always say what they think. *arXiv preprint*, arXiv:2503.06365

15. Voita, E., Titov, I., & Sennrich, R. (2019). The bottom-up evolution of representations in the transformer: A study with machine translation and language modeling objectives. *Proceedings of EMNLP-IJCNLP 2019*, pp. 4396–4406. DOI: 10.18653/v1/D19-1448

16. Xiao, T., et al. (2023). Efficient guided generation for large language models. *arXiv preprint*, arXiv:2302.09928

17. Kirchenbauer, J., et al. (2023). A watermark for large language models. *Proceedings of ICML 2023*, PMLR 202, pp. 17061–17084

18. Belghazi, M. I., et al. (2018). MINE: Mutual information neural estimation. *Proceedings of ICML 2018*, PMLR 80, pp. 531–540

19. Kraskov, A., Stögbauer, H., & Grassberger, P. (2004). Estimating mutual information. *Physical Review E*, 69(6), 066138. DOI: 10.1103/PhysRevE.69.066138

20. He, J., & Vechev, M. (2023). SVEN: Security and vulnerability-enabling/disabling language models for code. *arXiv preprint*, arXiv:2311.12509

21. Chen, M., et al. (2021). Evaluating large language models trained on code. *arXiv preprint*, arXiv:2107.03374

22. Li, R., et al. (2023). StarCoder: May the source be with you! *Transactions on Machine Learning Research*, ISSN 2835-8856

23. Rozière, B., et al. (2023). Code Llama: Open foundation models for code. *arXiv preprint*, arXiv:2308.12950

24. Banerjee, S., & Chua, C. E. H. (2024). Static analysis meets large language models: A security-focused survey. *ACM Computing Surveys*, 56(8), Article 193. DOI: 10.1145/3631534

25. Siddiq, M. L., & Santos, J. C. S. (2022). SecurityEval dataset: Mining vulnerability examples to evaluate machine learning-based code generation techniques. *Proceedings of MSR4PS 2022*, pp. 29–33. DOI: 10.1145/3549035.3561184

