УДК 004.056.5:37.018.43

Минасян Т.А.
студент 4 курса направления 01.03.02 «Прикладная математика и информатика»,
Институт компьютерных наук и технологий, ПГНИУ
Россия, г. Пермь
Мушкалов Е.А.
студент 4 курса направления 01.03.02 «Прикладная математика и информатика»,
Институт компьютерных наук и технологий, ПГНИУ
Россия, г. Пермь

АРХИТЕКТУРА И АЛГОРИТМЫ МНОГОУРОВНЕВОЙ СИСТЕМЫ БЕЗОПАСНОЙ ПРОВЕРКИ ПРОГРАММНЫХ РЕШЕНИЙ ДЛЯ ОБРАЗОВАТЕЛЬНЫХ ОНЛАЙН-ПЛАТФОРМ

Аннотация: в статье рассматривается задача безопасной автоматизированной проверки программных решений в образовательных онлайн-платформах. Показано, что запуск пользовательского кода нельзя рассматривать как обычную серверную операцию, поскольку такой код может содержать ошибки, бесконечные циклы, попытки доступа к файловой системе или намеренно опасные конструкции. Предложена многоуровневая архитектура, в которой основной сервер платформы отделен от контура выполнения, а проверка переносится в изолированный runner-узел с очередью задач, одноразовой средой запуска и ограничениями ресурсов. Результатом работы является модель угроз, алгоритм обработки решения и набор проектных мер, позволяющих снизить риск компрометации основной системы без отказа от интерактивной проверки заданий.

Ключевые слова: образовательная платформа, автоматическая проверка кода, песочница, Docker, gVisor, runner-сервер, модель угроз.

Minasyan T.A.
4th-year student, Applied Mathematics and Computer Science,
Institute of Computer Science and Technology, Perm State University
Russia, Perm
Mushkalov E.A.
4th-year student, Applied Mathematics and Computer Science,
Institute of Computer Science and Technology, Perm State University
Russia, Perm

ARCHITECTURE AND ALGORITHMS OF A MULTI-LEVEL SYSTEM FOR SECURE PROGRAMMING SOLUTION ASSESSMENT IN EDUCATIONAL ONLINE PLATFORMS

Abstract: the article examines the problem of secure automated assessment of programming solutions in educational online platforms. It is shown that user code execution cannot be treated as a regular server-side operation, because submitted code may contain errors, infinite loops, attempts to access the file system, or intentionally dangerous constructs. A multi-level architecture is proposed in which the main platform server is separated from the execution contour, while checking is moved to an isolated runner node with a task queue, disposable runtime environment, and resource limits. The result of the study is a threat model, a solution processing algorithm, and a set of design measures that reduce the risk of compromising the core system without abandoning interactive task assessment.

Keywords: educational platform, automated code assessment, sandbox, Docker, gVisor, runner server, threat model.

Введение

Автоматическая проверка программных решений стала обычной частью цифрового обучения программированию, анализу данных и машинному обучению. Студент отправляет код через браузер и почти сразу получает результат проверки. Такой сценарий дает быструю обратную связь и снижает нагрузку на преподавателя, поэтому системы автоматического оценивания давно применяются в учебных курсах [1].

Главная трудность состоит в том, что проверка требует запуска недоверенного пользовательского кода. Даже без злого умысла решение может содержать бесконечный цикл, чрезмерное потребление памяти или некорректную работу с вводом и выводом. В более жесткой модели угроз код может пытаться обратиться к файловой системе, внутренней сети, переменным окружения и системным вызовам.

Если основной сервер платформы одновременно хранит пользователей, задания, базу данных и запускает студенческий код, компрометация механизма проверки затрагивает всю систему. Поэтому безопасная проверка программных решений должна проектироваться как отдельный архитектурный контур, а не как локальная настройка интерпретатора.

Цель работы - спроектировать многоуровневую систему безопасной проверки программных решений для образовательных онлайн-платформ. Для этого выделяются основные угрозы, задаются роли компонентов, формируется алгоритм обработки решения и сопоставляются угрозы с механизмами защиты.

Методы и исследования

Методической основой выбран принцип защиты в глубину: безопасность не должна зависеть от одного барьера. Если статический анализ пропускает опасный код, его должен ограничивать контур выполнения. Если среда запуска скомпрометирована, ущерб должен

оставаться внутри runner-узла, не имеющего доступа к основной базе данных.

Процесс проверки разделяется на доверенный и недоверенный контуры. К доверенному относится основной сервер платформы: регистрация пользователей, хранение заданий, прием решений и отображение результата. К недоверенному относится выполнение кода. Этот контур выносится на отдельный runner-сервер, который получает только текст программы, язык выполнения, входные данные тестов и лимиты ресурсов.

Первым рубежом является статический анализ исходного кода. Для Python-решений он может выполняться через абстрактное синтаксическое дерево: модуль ast позволяет разбирать программный текст до запуска [2]. Так можно заранее отклонить импорт os, subprocess, socket, вызовы open, eval, exec и другие операции, не нужные учебной задаче. При этом AST-анализ остается фильтром, а не гарантией безопасности, поскольку динамические приемы языка и обфускация могут обходить простые правила.

Вторым элементом является очередь задач. Она отделяет прием решений от выполнения, сглаживает пики нагрузки и позволяет добавлять runner-узлы без изменения основного сервера. Распределенная очередь, например Celery, поддерживает асинхронную обработку, повтор задач при сбое и приоритеты для разных типов проверок [3].

Третьим элементом служит одноразовая среда запуска. Базовым вариантом может быть Docker-контейнер с лимитами CPU и памяти [4]. Однако контейнер не является полной границей безопасности, поэтому его следует дополнять seccomp-профилями [5], AppArmor или SELinux, запретом лишних capabilities, непривилегированным пользователем, read-only файловой системой и отключением сети; сходные меры приводятся и в практических рекомендациях OWASP [6]. В усиленном варианте

применяются gVisor [7], Kata Containers [8] или Firecracker microVM [9], которые сильнее отделяют пользовательский код от ядра хоста.

Результаты оригинального авторского исследования

В результате проектирования предложена распределенная архитектура безопасной проверки программных решений. Ее основной принцип: сервер образовательной платформы не выполняет пользовательский код. Он принимает решение, проводит предварительную проверку, сохраняет попытку и передает задачу в очередь. Выполнение переносится на runner-сервер, который не имеет прямого доступа к основной базе данных и возвращает только технический результат проверки.

Архитектура включает основной сервер платформы, очередь задач, runner-сервер и одноразовую среду запуска. Такая схема не делает выполнение недоверенного кода абсолютно безопасным, но снижает возможный ущерб: даже при успешном побеге из песочницы атакующий оказывается не на основном сервере с пользовательскими данными, а в ограниченном контуре выполнения.

Предлагаемое разделение близко к практике специализированных систем автоматической проверки, в которых выполнение кода отделяется от пользовательского интерфейса и учебной логики. В качестве примеров таких решений можно привести Judge0 [10] и CodeRunner [11].

Рисунок 1 - Общая схема распределенной проверки программного решения

Алгоритм обработки попытки включает прием решения, AST-предпроверку, постановку задачи в очередь, запуск на runner-узле и возврат нормализованного результата. Если на этапе анализа обнаружены запрещенные конструкции, попытка отклоняется без запуска. Если

проверка пройдена, runner создает новую одноразовую среду, копирует в нее код и запускает тесты под наблюдением.

Во время запуска контролируются wall-clock timeout и CPU-time limit, лимит памяти, число процессов, квота на временную файловую систему и максимальный размер stdout/stderr. После завершения проверки среда уничтожается, а основной сервер получает только статус, фрагмент вывода, сообщение об ошибке и технические метрики.

1. Принять решение пользователя и сохранить попытку со статусом pending.
2. Выполнить AST-предпроверку запрещенных импортов, вызовов и обращений.
3. При нарушении правил завершить попытку без запуска пользовательского кода.
4. При успешной предпроверке поставить задачу в очередь с тестами и лимитами.
5. Передать задачу runner-узлу, не имеющему доступа к основной базе данных.
6. Создать одноразовую среду выполнения: контейнер, gVisor-контейнер или microVM.
7. Запустить код под ограничениями ресурсов и времени.
8. Уничтожить среду выполнения и вернуть нормализованный результат.
9. Обновить попытку студента и сохранить признаки подозрительной активности.

Таблица 1 - Модель угроз и меры защиты

Угроза	**Механизм защиты**
Побег из контейнера или повышение привилегий	Непривилегированный пользователь, запрет capabilities, read-only FS, seccomp/AppArmor, gVisor или microVM.

Угроза	Механизм защиты
Fork-бомба и создание множества процессов	pids limit через cgroups и жесткий таймаут на выполнение.
Исчерпание CPU или памяти	CPU quota, memory limit, отключение swap и завершение задачи при превышении лимита.
Зависание без нагрузки на процессор	Одновременное использование wall-clock timeout и CPU-time limit.
Переполнение диска или временной директории	tmpfs с квотой, запрет монтирования файлов хоста, удаление среды после каждой попытки.
Бесконечный вывод в stdout/stderr	Ограничение размера вывода и обрезка результата перед возвратом в очередь.
DoS через массовую отправку решений	Rate limiting на пользователя, приоритеты очереди, ограничение числа активных попыток.
Подмена задачи или результата	Аутентификация между сервисами, токены или mTLS, подпись сообщений.

Модель допускает постепенное внедрение. Минимальный вариант может состоять из AST-предпроверки, Docker-контейнера и ограничения числа параллельных запусков. Расширенный вариант добавляет очередь задач, отдельный runner-узел, одноразовые среды, ограничения cgroups, seccomp/AppArmor, отключение сети и мониторинг. Усиленный вариант заменяет обычный Docker runtime на gVisor или microVM-подход.

Масштабируемость достигается за счет горизонтального увеличения числа runner-узлов. Очередь скрывает от основного сервера количество исполнителей, поэтому при росте нагрузки добавляются новые workers. Для разных языков программирования можно создавать отдельные пулы, а для контрольных работ - приоритетную очередь.

Подход имеет ограничения. Docker, gVisor и microVM уменьшают риск, но не дают абсолютной гарантии безопасности. Усиленная изоляция увеличивает задержку проверки, а слишком строгие лимиты могут мешать

корректным решениям. Поэтому для разных типов заданий нужны разные профили ресурсов и изоляции.

Разделение педагогической и инфраструктурной логики делает архитектуру применимой не только к курсам программирования, но и к задачам анализа данных, машинного обучения, SQL и другим дисциплинам, где студент отправляет исполняемый фрагмент решения.

Заключение

В статье предложена архитектура многоуровневой системы безопасной проверки программных решений для образовательных онлайн-платформ. Основной результат состоит в переносе выполнения пользовательского кода из основного приложения в изолированный контур, включающий очередь задач, runner-сервер и одноразовую среду запуска.

Сформирована модель угроз, включающая побег из среды выполнения, исчерпание ресурсов, сетевые атаки из контейнера, DoS через очередь, подмену задач и утечку состояния между пользователями. Для каждой угрозы предложены меры защиты: AST-предпроверка, rate limiting, аутентификация межсервисного обмена, контейнерная или microVM-изоляция, cgroups-лимиты, ограничение stdout/stderr и запрет сетевого доступа.

Практическая ценность архитектуры заключается в ее поэтапной реализуемости. Минимальный вариант подходит для небольшой учебной платформы, а расширенный позволяет масштабировать проверку за счет дополнительных runner-узлов и приоритетных очередей. Дальнейшая работа может быть связана со сравнением Docker, gVisor и microVM в учебных сценариях.

Использованные источники

1. Ihantola P. Review of recent systems for automatic assessment of programming assignments / P. Ihantola, T. Ahoniemi, V. Karavirta, O.

Seppälä // Proceedings of the 10th Koli Calling International Conference on Computing Education Research. - ACM, 2010. - P. 86-93. - DOI: 10.1145/1930464.1930480. - URL: https://researchportal.helsinki.fi/en/publications/review-of-recent-systems-for-automatic-assessment-of-programming-/ (дата обращения: 02.06.2026).

2. Ast - Abstract syntax trees [Электронный ресурс] // Python Documentation. - URL: https://docs.python.org/3/library/ast.html (дата обращения: 02.06.2026).

3. Celery - Distributed Task Queue [Электронный ресурс] // Celery Documentation. - URL: https://docs.celeryq.dev/ (дата обращения: 02.06.2026).

4. Resource constraints [Электронный ресурс] // Docker Docs. - URL: https://docs.docker.com/engine/containers/resource_constraints/ (дата обращения: 02.06.2026).

5. Seccomp security profiles for Docker [Электронный ресурс] // Docker Docs. - URL: https://docs.docker.com/engine/security/seccomp/ (дата обращения: 02.06.2026).

6. Docker Security Cheat Sheet [Электронный ресурс] // OWASP Cheat Sheet Series. - URL: https://cheatsheetseries.owasp.org/cheatsheets/Docker_Security_Cheat_Sheet.html (дата обращения: 02.06.2026).

7. What is gVisor? [Электронный ресурс] // gVisor Documentation. - URL: https://gvisor.dev/docs/ (дата обращения: 02.06.2026).

8. Kata Containers [Электронный ресурс]. - URL: https://katacontainers.io/ (дата обращения: 02.06.2026).

9. Firecracker microVMs [Электронный ресурс]. - URL: https://firecracker-microvm.github.io/ (дата обращения: 02.06.2026).

10. Judge0 CE API Docs [Электронный ресурс]. - URL: https://ce.judge0.com/ (дата обращения: 02.06.2026).

11. CodeRunner [Электронный ресурс]. - URL: https://coderunner.org.nz/ (дата обращения: 02.06.2026).