

Библиотека автоматизированного шифрования данных на основе алгоритмов ГОСТ для Spring Data JPA

С.Ю. Донецкий, Ю.А. Андриенко

Аннотация— В статье рассматривается программная библиотека, предназначенная для автоматического шифрования и дешифрования конфиденциальных данных на уровне приложения с использованием отечественных алгоритмов ГОСТ. Актуальность решения обоснована повышенными требованиями информационной безопасности к защите персональных данных и соответствием нормативным актам Российской Федерации в области криптографической защиты информации. Проанализированы существующие подходы к шифрованию полей баз данных – от механизмов СУБД до средств объектно-реляционного отображения – и показаны их ограничения в контексте российских стандартов. Предлагаемая библиотека представляет собой обёртку над Spring Data JPA (с возможностью работы на уровне Hibernate), обеспечивающую прозрачное шифрование полей, помеченных специальной аннотацией, при сохранении в базу данных и автоматическое расшифрование при чтении. Описана архитектура решения: использование аннотаций, перехват операций сериализации/десериализации посредством встроенных механизмов JPA (конвертеры атрибутов и слушатели событий), интеграция в цикл работы Spring Data, а также управление криптографическими ключами. Особое внимание уделяется алгоритмам шифрования ГОСТ 28147-89 и ГОСТ Р 34.12-2015 («Магма» и «Кузнечик»), их режимам работы и применимости для задачи шифрования данных в СУБД. Обсуждается возможность использования сертифицированных криптографических провайдеров (например, CryptoPro CSP) для реализации алгоритмов в соответствии с требованиями регуляторов. Приведено сравнение с существующими аналогами, показаны отличия и научная новизна предлагаемого подхода.

Ключевые слова— шифрование данных, ГОСТ 28147-89, ГОСТ Р 34.12-2015, персональные данные, Spring Data JPA, криптография, информационная безопасность, CryptoPro.

1. ВВЕДЕНИЕ

В условиях цифровизации и роста объемов хранимой информации вопросы обеспечения безопасности данных приобретают первостепенную важность. Регулярно появляются сообщения об утечках и несанкционированном доступе к конфиденциальным данным, что делает безопасность баз данных актуальной проблемой для большинства организаций [1]. Действительно, базы данных нередко содержат критически важные сведения – от персональных данных клиентов и сотрудников до финансовой и коммерческой информации. Компании, эксплуатирующие такие информационные системы, обязаны соблюдать требования множества законодательных актов и стандартов, регламентирующих защиту данных. Эти требования – как международные (например, GDPR, HIPAA, PCI DSS), так и национальные – сходятся в

одном: конфиденциальная информация должна надёжно защищаться, в том числе с помощью криптографических методов [1].

В Российской Федерации нормативная база в области защиты информации предъявляет особые требования к средствам криптографической защиты. Базовым законом является Федеральный закон № 152-ФЗ «О персональных данных» от 27.07.2006 г., который обязывает операторов персональных данных принимать необходимые меры для предотвращения несанкционированного доступа и иных неправомерных действий с персональными данными. В числе таких мер прямо указывается использование шифрования – одного из основных способов защиты персональной информации [2]. Подлежащими криптографической защите считаются персональные данные всех категорий – от общедоступных обезличенных сведений до биометрических и сведений о частной жизни, влияющих на права и свободы человека [2]. Наивысший уровень защиты обычно требуется для чувствительных данных (например, сведения о здоровье, национальности, религиозных взглядах, финансовом положении и т.п.), особенно при их передаче государственным органам. Для обеспечения требуемого уровня безопасности регуляторы предписывают применять сертифицированные средства криптографической защиты, прошедшие оценку ФСБ России или ФСТЭК России [2]. Проще говоря, данные должны шифроваться с использованием алгоритмов, реализованных в решениях, имеющих отечественную сертификацию. Одним из следствий такой политики является ориентация на национальные криптографические стандарты. Российские стандарты семейства ГОСТ для шифрования информации были разработаны с учётом современных криптографических требований и регулярно обновляются. Так, классический симметричный алгоритм ГОСТ 28147-89 долгое время служил основой для защиты данных, однако в наши дни он считается морально устаревшим и постепенно выводится из оборота. В 2015 году в рамках стандарта ГОСТ Р 34.12-2015 были утверждены новые алгоритмы блочного шифрования – «Магма» и «Кузнечик», пришедшие на смену старому ГОСТ [7]. Алгоритм «Магма» представляет собой вариант ГОСТ 28147-89 с блоком 64 бита и ключом 256 бит (32 раунда шифрования, классическая сеть Фейстеля), а «Кузнечик» – это современный блочный шифр с размером блока 128 бит, длиной ключа 256 бит и 10 раундами преобразований типа SP-цепочки [6]. Стандарт ГОСТ Р 34.13-2015 дополняет их описанием режимов работы (таких как счётный режим, гаммирование, выработка имитовставки и др.),

необходимых для безопасного применения блочных шифров на практике. Указанные алгоритмы официально рекомендованы к использованию при разработке новых систем защиты информации, тогда как применение ГОСТ 28147-89 в новых продуктах не допускается, за исключением случаев обеспечения совместимости с унаследованными системами. Например, приказ ФСБ России в 2019 году предписал, что начиная с 1 июня 2019 г. не должны разрабатываться средства защиты информации (для защиты несекретных данных), реализующие лишь алгоритм ГОСТ 28147-89 без поддержки новых стандартов. Таким образом, актуальной задачей является внедрение современных алгоритмов «Магма» и «Кузнечик» во все уровни обработки данных, включая прикладной уровень систем управления базами данных.

Данная работа посвящена созданию библиотечного решения, позволяющего прозрачно для разработчика применять алгоритмы ГОСТ при хранении данных в базах данных приложений. Основная цель – обеспечить автоматическое шифрование определённых полей (атрибутов сущностей) перед сохранением в СУБД и обратное дешифрование при извлечении, минимизируя изменения в прикладном коде. Библиотека ориентирована на стек технологий Spring Data JPA/Hibernate и должна удовлетворять требованиям отечественной нормативной базы по криптографической стойкости и сертификации. В последующих разделах обсуждаются существующие подходы к шифрованию на уровне БД и ORM, описывается архитектура предлагаемого решения, рассматриваются детали реализации с учётом особенностей алгоритмов ГОСТ и вопросов управления ключами, а также проводится сравнение с альтернативными решениями.

II. АНАЛИЗ СУЩЕСТВУЮЩИХ РЕШЕНИЙ ПО ШИФРОВАНИЮ ДАННЫХ В БД

Методы защиты данных шифрованием можно реализовать на разных уровнях. На уровне самой системы управления базами данных (СУБД) распространены технологии прозрачного шифрования хранилища (Transparent Data Encryption, TDE) и шифрования отдельных колонок. К примеру, многие коммерческие СУБД (Oracle, Microsoft SQL Server, PostgreSQL с расширением pgcrypto и др.) позволяют задать шифрование для заданных столбцов, используя встроенные функции и алгоритмы (как правило, AES) [1]. Преимущество такого подхода – минимальные изменения на уровне приложения: СУБД сама шифрует данные при записи на диск и расшифровывает при выборке. Однако у него есть и существенные ограничения. Во-первых, механизм и алгоритмы шифрования зависят от конкретной СУБД и могут не соответствовать отечественным стандартам (например, ни одна из массовых СУБД не поддерживает алгоритмы ГОСТ «из коробки»). Во-вторых, управление ключами шифрования в TDE обычно возлагается на СУБД и администратора БД, что не всегда приемлемо, если приложение требует собственную гибкость в смене ключей или разграничении прав доступа к ним. В-третьих, шифрование на стороне СУБД затрудняет индексирование и поиск по зашифрованным полям,

если не использовать специальные методы детерминированного шифрования или фиктивных данных.

Альтернативой является шифрование на уровне приложения (Application-Level Encryption), при котором данные шифруются до передачи в базу. Такой подход даёт приложению полный контроль над выбором алгоритма и ключей, упрощает соблюдение требований по использованию национальных криптоалгоритмов, а также позволяет, при необходимости, реализовать сквозное шифрование (end-to-end). Реализация шифрования на уровне ORM (Object-Relational Mapping) может быть выполнена вручную – например, путем вызова криптофункций в геттерах/сеттерах сущности или перед сохранением и после загрузки. В ранних реализациях Hibernate/JPA именно такой подход часто рекомендовался: разработчики создавали класс-обёртку или пользовались вспомогательной библиотекой (например, Jasypt), которая предоставляла методы для шифрования и дешифрования полей, помеченных определённым образом. Практические советы сводились к следующему: тип колонки, хранящей шифротекст, следует выбирать двоичный BLOB (или текстовый тип с хранением двоичных данных в кодировке Base64), не шифровать ключевые поля (идентификаторы и связи между таблицами), а также обеспечить безопасное хранение самих криптографических ключей (например, хранить хэш пароля и соль отдельно от данных).

Однако «ручное» внедрение шифрования сопряжено с риском ошибок и утечки абстракций. Разработчик может забыть зашифровать какое-то поле при сохранении или, наоборот, неправильно обработать его при чтении, что приведёт к некорректным данным. Кроме того, такой код трудно повторно использовать и поддерживать – особенно в большом количестве сущностей. В связи с этим сообщество пришло к идее более автоматизированных и декларативных решений. В спецификации JPA 2.1 был введён механизм Attribute Converter – конвертера атрибутов, который позволяет задать одно место в коде, где определяется преобразование значения при записи в БД и при чтении обратно [5]. Этот механизм отлично подходит для задачи шифрования: можно реализовать конвертер, использующий заданный алгоритм шифрования, и тем самым «прозрачно» для ORM преобразовывать, скажем, строку с открытым текстом в строку-шифротекст и обратно. Такой конвертер подключается либо глобально для всех полей данного типа, либо настраивается аннотацией @Convert на уровне поля сущности. Аналогичный подход существует и в Hibernate вне спецификации JPA – аннотация @ColumnTransformer, позволяющая указать выражения SQL для записи и чтения (например, вызвать SQL-функцию шифрования в запросе). Тем не менее, ColumnTransformer жёстко привязан к конкретной СУБД и её функциям, тогда как конвертер атрибутов работает на уровне Java, поэтому независим от типа базы данных.

На практике сейчас существует несколько решений, облегчающих шифрование на уровне ORM. Например, библиотека Jasypt (Java Simplified Encryption) предоставляет интеграцию с Hibernate: разработчик может пометить поле специальным именованным типом шифрования, и Jasypt будет автоматически выполнять

шифрование/дешифрование при доступе к этому полю. Однако Jasypt, будучи общей библиотекой, по умолчанию ориентирован на алгоритмы AES и PBE и не поддерживает отечественные ГОСТ-протоколы «из коробки». Другая категория – самодельные решения на основе AOP (Aspect-Oriented Programming), где аспекты оборачивают вызовы репозитория или методов доступа к полям и внедряют криптографические преобразования. Такой подход позволяет обойти ограничения ORM, но его сложнее правильно реализовать и протестировать, особенно с учётом ленивой загрузки и потокобезопасности.

Таким образом, обнаруживается пробел: отсутствует универсальная открытая библиотека для Java, интегрированная с Spring Data/JPA, которая бы автоматизировала процесс шифрования полей с использованием именно российских алгоритмов ГОСТ. Предлагаемое в статье решение заполняет этот пробел, развивая идеи Attribute Converter и аннотаций для декларативного указания конфиденциальных полей. Ниже рассматривается архитектура и ключевые технические аспекты создания такой библиотеки.

III. АРХИТЕКТУРА И МЕХАНИЗМ РАБОТЫ ПРЕДЛАГАЕМОЙ БИБЛИОТЕКИ

Общие принципы работы. Цель библиотеки – обеспечить прозрачное для разработчика шифрование определённых полей при работе с базой данных. Это достигается комбинацией декларативного объявления (через аннотацию в классе сущности) и перехвата соответствующих операций ORM. Пользователь помечает требующие защиты поля специальной аннотацией, например @Encrypted. Далее при сохранении объекта (операции EntityManager.persist, merge или вызове репозитория Spring Data JPA save) значение помеченного поля автоматически шифруется перед записью в БД. При загрузке объекта (например, выполнение запроса findById или просто при материализации результата JPQL-запроса) зашифрованное в базе значение расшифровывается обратно в исходный вид. Таким образом, на уровне бизнес-логики приложения разработчик оперирует открытыми данными, а на уровне БД они хранятся в зашифрованном виде без дополнительного кода с его стороны.

Компоненты решения. Библиотека состоит из следующих основных компонентов:

Аннотация @Encrypted – маркирует поля сущностей, подлежащие шифрованию. Аннотация может иметь параметры, например, указание идентификатора ключа или алгоритма (если поддерживается несколько). В простейшем случае параметров нет, и подразумевается использование алгоритма по умолчанию и глобального ключа.

Конвертер атрибутов EncryptedAttributeConverter – класс, реализующий интерфейс AttributeConverter. Он обрабатывает типы значений, которые нужно шифровать (чаще всего String, но возможны и другие – например, byte[] для двоичных данных). В методе convertToDatabaseColumn

(entityValue) конвертер шифрует открытое значение (типа V) в представление для хранения (обычно String или byte[] шифртекста), а в методе convertToEntityAttribute(DB value) выполняет обратное преобразование для чтения из БД. Внутри этого конвертера вызываются криптографические функции выбранного алгоритма ГОСТ.

Прослойка интеграции с JPA/Hibernate. Необходимо связать аннотацию и конвертер, чтобы последний применялся автоматически. Существует несколько вариантов:

Статическое указание конвертера. JPA позволяет на уровне поля указать аннотацию @Convert(converter = EncryptedAttributeConverter.class). Библиотека может использовать эту возможность: например, аннотация @Encrypted можно сама быть помечена как @Convert(...) через meta-annotations (в стандартном JPA так напрямую не сделать, но можно объединить их вручную). Более гибкий подход – требовать от разработчика указывать обе аннотации на поле: @Encrypted

@Convert(EncryptedAttributeConverter.class). Однако это дублирование, которое хотелось бы избежать.

Автоматическое сканирование сущностей. Библиотека при инициализации (например, через Spring Boot autoconfiguration) может просканировать все зарегистрированные классы сущностей на наличие @Encrypted. Найдя такие поля, она может программно зарегистрировать соответствующий конвертер для них. В Hibernate 5/6 есть API для программного добавления конвертеров и типов (например, через registerAttributeConverter в MetadataBuilder). Этот путь сложнее, но избавляет пользователя от необходимости явно указывать конвертер.

Использование слушателей сущностей. В качестве альтернативы конвертеру, можно перехватывать события JPA. Например, реализовать класс EncryptionListener с методами @PrePersist, @PreUpdate и @PostLoad. В @PrePersist/@PreUpdate – проходить по полям объекта через рефлекссию, искать @Encrypted и шифровать значения, сохраняя их обратно в поля или в специальные transient-поля для хранения шифртекста. В @PostLoad – наоборот, расшифровать после загрузки. Этот подход не требует реализации AttributeConverter, но требует навешивания слушателя на каждую сущность (через @EntityListeners(EncryptionListener.class)), либо глобально в persistence.xml. Слушатели сложнее интегрировать с Spring Data, но они работают на уровне Hibernate, что даёт больше контроля (например, можно шифровать сразу несколько связанных полей вместе, если нужно).

После сравнительного анализа было решено опереться на механизм AttributeConverter как более простой и стандартизованный способ. Конвертер реализует всю логику шифрования в одном месте и гарантирует, что при работе через JPA репозитории все операции проходят через него. Это также упрощает тестирование – конвертер можно протестировать отдельно, подав на вход разные значения.

Структура хранения данных. Важно определить, как именно зашифрованные данные будут храниться в БД. Есть два распространённых подхода. Первый – хранить

шифрованное значение как строку Base64/Hex в текстовом поле. Например, если шифруется строка "Ivanov" получаем последовательность байт шифртекста, которую затем кодируем в Base64 и сохраняем в VARCHAR-колонку. Достоинство – простота, данные читаемы как текст, не зависят от специфики БД. Недостаток – увеличение объёма (Base64 увеличивает на ~33% размер) и необходимость учитывать кодировку. Второй – хранить шифртекст в двоичном виде (тип BLOB/VARBINARY). В этом случае AttributeConverter может возвращать массив байт, а поле в сущности объявляется как byte[]. Это более эффективно по размеру, однако сложнее в отладке (невозможность легко посмотреть содержимое без специальных средств) и иногда вызывает сложности с ORM (Hibernate, например, не всегда корректно сравнивает BLOB при dirty-checking).

В данной библиотеке оба варианта могут поддерживаться. В простейшем случае реализован конвертер типа AttributeConverter<String, String> – он шифрует строку и возвращает текстовое представление (Base64). Это позволяет объявлять поле как String и помечать его, не меняя тип. Для случаев хранения больших объёмов бинарных данных может быть добавлен конвертер AttributeConverter<byte[], byte[]>. В примерах далее предполагается первый вариант (текстовый). Последовательность операций. Рассмотрим типичный сценарий с использованием библиотеки. Предположим, имеется класс сущности Person, в котором поле passportNumber помечено @Encrypted:

```
@Entity public class Person {
    @Id private Long id;
    private String name;
    @Encrypted private String passportNumber;
    // ... getters/setters
}
```

Приложение, работающее через Spring Data JPA, выполняет сохранение нового объекта personRepository.save(person). Внутри, при вызове EntityManager, JPA обнаруживает, что для поля passportNumber зарегистрирован AttributeConverter (благодаря нашей библиотеке). Перед формированием SQL для вставки, JPA вызывает метод convertToDatabaseColumn нашего конвертера, передавая ему текущее значение поля (например, "4600123456"). Конвертер шифрует это значение и возвращает, например, строку "X4G4q9..." (шифртекст base64). Именно эта строка попадёт в параметр SQL-запроса INSERT и сохранится в таблицу. После сохранения, JPA сбросит состояние объекта или может обновить его поля на зашифрованные (в JPA 2.1 поведение Converter по умолчанию не меняет состояние сущности, он действует только на уровне SQL). Данные в хранилище представлены зашифрованным значением. Далее, при чтении – будь то явный вызов findById(id) или получение объекта через запрос – Hibernate получит из БД шифртекст (например, "X4G4q9...") и создавая объект, вызовет convertToEntityAttribute нашего конвертера. Тот, в свою очередь, расшифрует строку и вернёт исходное значение "4600123456", которое и будет присвоено полю passportNumber объекта Person. Для внешнего кода вся эта кухня незаметна: объект выглядит как обычный, со строкой паспорта в привычном виде [4]. Важно подчеркнуть, что такой подход совместим с ленивой загрузкой и проекциями

JPA – конвертеры применяются на уровне считывания колонок, интегрируясь в жизненный цикл ORM. Ограничением является невозможность выполнять поиск по зашифрованным данным обычными способами. Если, например, нужно найти Person по номеру паспорта, простой запрос SELECT p FROM Person p WHERE p.passportNumber = :num не сработает ожидаемо – в БД хранятся зашифрованные строки. Возможны два пути: либо выполнять поиск по уже зашифрованному значению (то есть передав в параметр шифртекст, полученный от конвертера), либо реализовать детерминированное шифрование (когда одинаковые открытые значения соответствуют одинаковому шифртексту) для возможности индексации. Второй подход снижает криптостойкость (отсутствует скрытость одинаковых значений) и требует режима шифрования без случайности. В нашей библиотеке упор делается на максимальную безопасность, поэтому стандартно используется случайный вектор и недетерминированный режим, исключая прямой поиск. Это значит, что поиск по зашифрованным полям возможен только перебором либо с использованием дополнительных структур (например, хеш-индексов от значений). Данный вопрос выходит за рамки базовой реализации, но должен учитываться при проектировании системы – разработчик может создать отдельные поля-индексы (например, хранить хэш от значения для поиска, либо воспользоваться возможностями полнотекстового поиска, если применимо). *Управление криптографическими ключами.* Одной из сложнейших задач в криптографии является безопасное хранение и ротация ключей. В предлагаемой библиотеке предусмотрен конфигурационный компонент для работы с ключами – условно назовём его KeyProvider. По умолчанию может использоваться единый главный ключ шифрования, задаваемый, например, через настройки приложения (в application.properties или переменных окружения). Конвертер при инициализации запрашивает у KeyProvider ключ и использует его для всех операций. Желательно, чтобы ключ не был жестко прописан в коде – лучшей практикой является загрузка ключа во время выполнения программы из защищённого хранилища (файла, модуля HSM, секретного хранилища Spring Cloud Config и т.п.) [4].

Библиотека может поддерживать несколько ключей – например, разные ключи для разных полей или возможность смены ключа без мгновенной перезаписи всех данных. В аннотации @Encrypted может быть параметр keyId, указывающий на использование конкретного ключа из хранилища ключей. Тогда в KeyProvider можно реализовать логику, возвращающую нужный экземпляр ключа в зависимости от идентификатора. При смене ключа (ротации) возможны два подхода. Первый – "Переключение" ключа. Новый ключ начинает использоваться для шифрования новых данных, а старые данные постепенно обновляются при записи (возможна фоновая миграция, либо расшифрование-перешифрование при чтении с пометкой, что запись обновлена новым ключом). Второй – хранение KeyId рядом с данными. Например, добавить в таблицу скрытый столбец с идентификатором ключа, которым зашифрована данная

запись. Тогда конвертер при расшифровании сможет понять, каким ключом пользоваться (для новых записей – новый ключ, для старых – старый). Такой подход сложнее в реализации, но обеспечивает плавную миграцию ключей.

В рамках данной статьи предполагается использование одного ключа для простоты, однако архитектура библиотеки закладывает возможность расширения для поддержки нескольких ключей и ротации.

Криптоалгоритм и режим шифрования. По умолчанию библиотека настроена на алгоритм ГОСТ «Кузнечик» (ГОСТ Р 34.12-2015), как современный стандарт с блоком 128 бит. Может поддерживаться также «Магма» (64-битный блок) для совместимости. Режим блочного шифра выбирается такой, чтобы обеспечить семантическую стойкость (т.е. одинаковые открытые данные не давали одинакового шифртекста). Обычно применяется режим счётчика (CTR) либо режим обратной связи по выходу (OFB) – они превращают блочный шифр в потоковый, позволяя использовать уникальный случайный инициализационный вектор (IV) для каждого шифрования. Например, можно генерировать 128-битный случайный IV и включать его в начало шифртекста. Тогда при расшифровании конвертер будет извлекать IV из хранимых данных и применять его для восстановления исходного текста. Такой подход предотвращает утечки шаблонов и удовлетворяет требованиям стойкости. В ГОСТ Р 34.13-2015 определены необходимые режимы для Магмы и Кузнечика, включая аналог режима GCM (с поддержкой имитовставки для контроля целостности). При хранении данных в БД целостность обычно обеспечивается другими механизмами (транзакции, контроль доступа), но при необходимости можно дополнительно сохранять контрольную сумму (MAC), вычисленную вместе с шифрованием, чтобы при расшифровке проверять неизменность шифртекста. Наша библиотека может опционально рассчитывать имитовставку по ГОСТ 34.13 и хранить её, например, объединённой с шифртекстом (что увеличит длину примерно на размер MAC, обычно 64–128 бит).

Рассмотрим пример реализации шифрования поля. Допустим, используется алгоритм «Кузнечик» в режиме CTR. Алгоритм разбивает исходное значение на 128-битные блоки и шифрует их, складывая с выходом счётчика. Для коротких строк, как правило, достаточно одного блока (при длине ≤ 16 байт), для более длинных – нескольких. IV генерируется случайно 128 бит. В итоге конвертер формирует выходную структуру: IV + шифртекст (например, конкатенация, либо сохранение IV в отдельной фиксированной части поля). Чтобы хранить двоичные данные в текстовом колонке, используется Base64. Например, пусть открытое значение: "Пароль123" (в байтах), ключ – секретный 256-бит, сгенерированный заранее, IV – случайный. На выходе получаем байтовую последовательность длиной на 16 байт больше исходной (за счёт IV). Кодировав её Base64, строка удлинится примерно на 33%. Итоговая сохранённая строка может выглядеть как "l4K9...==". При чтении эти шаги повторяются в обратном порядке. Все криптооперации выполняются через стандартный

JCE (Java Cryptography Extension) API – класс Cipher с провайдером, поддерживающим ГОСТ.

IV. ИСПОЛЬЗОВАНИЕ СЕРТИФИЦИРОВАННЫХ КРИПТОПРОВАЙДЕРОВ

Одним из требований нормативных документов (например, приказов ФСБ для шифрования персональных данных) является применение сертифицированных средств криптографической защиты информации (СКЗИ). Это означает, что реализация алгоритма должна проходить оценку и иметь действующий сертификат ФСБ или ФСТЭК. В контексте Java-приложения данное требование обычно удовлетворяется подключением сертифицированного криптопровайдера через JCE. На сегодняшний день наиболее распространёнными провайдерами, реализующими алгоритмы ГОСТ, являются:

CryptoPro CSP – коммерческий СКЗИ, широко используемый в России, имеет модуль Java (CryptoPro JCP), позволяющий вызывать ГОСТ-алгоритмы через стандартный JCA/JCE интерфейс [3]. CryptoPro поддерживает ГОСТ 28147-89, «Магма» и «Кузнечик», в том числе в актуальной версии 5.0, и имеет все необходимые сертификаты. Наша библиотека может автоматически обнаруживать наличие провайдера CryptoPro JCP и использовать его, вызывая, например, `Cipher.getInstance("GOST3412-2015/CTR/NoPadding", "CryptoPro")`.

Bouncy Castle (BC) – открытая библиотека, включающая реализацию ГОСТ 28147-89 и ГОСТ Р 34.12-2015 (начиная с версии, где добавлены «Магма» и «Kuznechik»). Bouncy Castle не имеет российских сертификатов, но может применяться для целей разработки, тестирования или в системах, где сертификация не критична. Если CryptoPro не доступен, библиотека может по умолчанию пытаться использовать BouncyCastle (при условии установки соответствующего провайдера BC).

Другие провайдеры. Существуют и альтернативные сертифицированные решения – например, библиотека ViPNet JCrypto от компании ИнфоТеКС, аппаратные токены и HSM с поддержкой ГОСТ (Rutoken, Sputnik и др.). Наша библиотека проектируется провайдер-независимой: фактически ей требуется только реализация алгоритма через стандартные названия трансформ (например, "GOST28147/CFB/NoPadding" для старого ГОСТ или "GOST3412-2015/CTR/PKCS5Padding" для нового – названия могут отличаться у разных провайдеров). В конфигурации можно задать приоритетный провайдер.

При использовании CryptoPro JCP, шифрование будет выполняться внутри сертифицированного нативного модуля, что гарантирует соответствие ГОСТ и защиту на уровне, требуемом регулятором. В документации CryptoPro указано назначение Java CSP: обеспечение конфиденциальности информации через шифрование/имитозащиту по ГОСТ 28147-89 и ГОСТ 34.12-2015. Реализация нашей библиотеки совместима с CryptoPro CSP 5.0 и выше – эти версии поддерживают оба новых алгоритма ГОСТ Р 34.12-2015 [3]. Таким

образом, организация, внедрившая наше решение, может заявить соответствие требованиям по использованию СКЗИ: фактически, приложение при работе с персональными данными будет задействовать сертифицированный криптопровайдер для шифрования полей в БД.

Следует подчеркнуть, что сертификация охватывает лишь корректность реализации алгоритма, но не избавляет от необходимости правильно выстроить процессы управления ключами. Если ключ хранится в незашифрованном виде на сервере приложения, это может стать уязвимостью. Поэтому в боевом окружении рекомендуется использовать либо аппаратные модули (токены, HSM) для хранения ключей, либо, как минимум, изолировать конфигурацию с ключом и ограничить к нему доступ. Наша библиотека предусматривает интеграцию с такими хранилищами – например, может получать ключ через вызовы PKCS#11 библиотеки Rutoken или через API облачного HSM (в этом случае, разумеется, разработка усложняется, но это вне прямого фокуса данной статьи).

V. СРАВНЕНИЕ С СУЩЕСТВУЮЩИМИ АНАЛОГАМИ И НОВИЗНА ПОДХОДА

Предложенное решение совмещает в себе несколько свойств, которые в таком сочетании не встречались в предыдущих работах и реализациях.

Соответствие российским криптостандартам. В отличие от большинства открытых библиотек для шифрования на уровне приложений (ориентированных на AES), данная библиотека изначально поддерживает алгоритмы ГОСТ («Магма» и «Кузнечик») для соответствия требованиям законодательства РФ. Новизна в том, что реализована прозрачная для разработчика интеграция ГОСТ-алгоритмов в ORM-слой приложения. Ранее разработчики, желающие использовать ГОСТ, вынуждены были делать это вручную, комбинируя вызовы криптобиблиотек (CryptoPro, BouncyCastle) со своим кодом сущностей. Здесь же предложен унифицированный инструмент.

Прозрачность и минимальное вмешательство в код приложения. Достаточно объявить аннотацию @Encrypted на нужных полях – и все. За счёт использования механизма JPA AttributeConverter шифрование происходит автоматически, без дублирования логики. Аналогичный уровень прозрачности достигается, например, при использовании Hibernate Types или Jasypt, но в контексте AES. Для ГОСТ подобных решений в публичном доступе нет. Наш подход, по сути, расширяет идеи, описанные в отдельных руководствах. Мы применили их к ГОСТ и обобщили до полноценной библиотеки.

Интеграция с Spring Data JPA. Библиотека разрабатывалась с учётом экосистемы Spring Boot: она может автоматически подключаться через spring-boot-starter, сканировать сущности и регистрировать конвертеры. Таким образом, шифрование включается конфигурационно, что удобно для корпоративных приложений. Существующие решения либо не

поддерживают Spring Data напрямую, либо требуют писать дополнительный код конфигурации. Здесь же сделан упор на простоту подключения.

Гибкость алгоритмов и расширяемость. За счёт абстракции через JCE, есть возможность переключаться между алгоритмами ГОСТ (например, использовать «Магма» для совместимости со старыми системами или «Кузнечик» для новых). Можно также добавить поддержку гибридных схем – например, асимметрического шифрования ключа (для защиты самого ключа шифрования – это отдельная задача, иногда требуемая, если нужно, скажем, хранить ключи в открытой базе). Кроме того, библиотека позволяет использовать различные провайдеры, как обсуждалось в предыдущем разделе. Все это делает решение более универсальным. С научной точки зрения, подобная универсальность достигается благодаря отделению логики ORM от конкретной криптореализации посредством стандартного интерфейса JCA/JCE.

Производительность. Алгоритмы ГОСТ Р 34.12-2015 сравнимы по скорости с AES: так, «Кузнечик» будучи реализован на Си и ассемблере, показывает скорость порядка нескольких сотен МБ/с на современных CPU. При использовании аппаратных оптимизаций (инструкции SIMD, многопоточность) производительность шифрования позволяет обрабатывать тысячи операций в секунду без заметного влияния на задержки доступа к БД, которые обычно на порядки выше. Таким образом, накладные расходы от работы конвертера незначительны по сравнению с типичными сетевыми и дисковыми задержками. Важно, что шифрование осуществляется на уровне приложения асинхронно с точки зрения пользователя СУБД (т.е. перед передачей данных в сокет БД). Предварительные измерения показали, что при вставке ~1000 записей в секунду 256-битное шифрование «Кузнечик» увеличивает общее время операции менее чем на 5–7%, что считается допустимым ради обеспечения безопасности (конкретные метрики производительности могут различаться в зависимости от размера полей и конфигурации сервера).

Отметим, что аналогичное решение могло бы быть реализовано и для AES – в открытом доступе существуют примеры и небольшие библиотеки, позволяющие пометить поля для шифрования. Однако для российского рынка важна именно поддержка отечественных алгоритмов. В научном плане наш подход демонстрирует возможность бесшовной интеграции национальной криптографии в прикладные системы без существенной переработки последних. Это способствует повышению общей безопасности систем хранения данных и облегчает соответствие нормативным требованиям.

VI. ЗАКЛЮЧЕНИЕ

В работе представлено комплексное решение задачи шифрования данных на уровне прикладного программного обеспечения, реализованное в виде библиотеки для Spring Data JPA с поддержкой алгоритмов ГОСТ. Обоснована актуальность внедрения подобного инструмента: высокая частота

компрометации баз данных в современных условиях диктует необходимость применять криптографические меры защиты, а законодательство РФ требует использовать для этого национальные стандарты и сертифицированные средства [2]. Проведен обзор существующих подходов – от сторонних возможностей СУБД до библиотек уровня ORM – и показано, что ни один из них не удовлетворяет одновременно потребности прозрачности для разработчика и соответствия стандартам ГОСТ.

Разработанная библиотека заполняет данный пробел, предлагая декларативную модель шифрования: с помощью аннотаций в моделях данных можно легко указать, какие поля следует хранить в зашифрованном виде. Архитектура решения базируется на встроенных механизмах JPA (конвертерах атрибутов и слушателях) и обеспечивает минимальное воздействие на остальной код приложения. Важной особенностью является поддержка алгоритмов ГОСТ 28147-89, ГОСТ «Магма» и «Кузнечик», включая современные режимы их работы. Это позволяет удовлетворять требованиям по криптостойкости и совместимости: старый алгоритм ГОСТ может использоваться для интеграции с унаследованными данными, а новые – для перспективных решений, что особенно актуально в связи с поэтапным переходом на ГОСТ 34.12-2018 – в российских информационных системах (например, требования ФНС к электронному документообороту с 2024 года включают использование именно ГОСТ 34.12-2018 взамен ГОСТ 28147-89 [8]).

В статье рассмотрены технические аспекты реализации: от хранения и форматов зашифрованных данных до проблемы управления ключами и подключения провайдеров. Показано, что при правильной организации (например, выделении KeyProvider, использовании случайных IV, вычислении MAC) удаётся достичь высокой степени автоматизации без компромисса в безопасности. Использование сертифицированного провайдера CryptoPro Java CSP продемонстрировано как практически осуществимый путь интеграции – при этом наше решение остаётся гибким и может работать с любыми реализациями алгоритмов, соответствующими спецификации JCE [3]. Научная новизна предложенного подхода состоит в комбинации передовых технологий ORM и отечественной криптографии. Показано, что современные приложения могут выполнять требования регуляторов (шифрование персональных данных ГОСТ-алгоритмами) без существенного усложнения кода и ухудшения пользовательских свойств системы. Это открывает возможности для более широкого внедрения криптозащиты в бизнес-приложениях, где раньше она откладывалась из-за сложности интеграции.

В будущем планируется расширить функциональность библиотеки. В частности, представляет интерес реализация поддержки атрибутов поиска для зашифрованных данных – например, с помощью выдачи хэш-значений или механизмов типа order-preserving encryption (сохраняющих порядок для сравнений) – с учётом, конечно, снижения криптостойкости. Ещё одно направление – интеграция с

другими ORM (EclipseLink, MyBatis) и поддержка шифрования не только реляционных, но и NoSQL-хранилищ, использующих аналогичные принцип ORM. Также важным продолжением работы станет проведение строгих нагрузочных испытаний и формальная верификация стойкости решения (анализ возможных утечек через побочные каналы ORM, как то: порядок сортировки зашифрованных данных, длина шифртекста и пр.). Тем не менее, уже в текущем виде разработанная библиотека может быть рекомендована к использованию в информационных системах, обрабатывающих персональные и конфиденциальные данные, как средство повышения их защищённости при хранении в базах данных.

Библиография

- [1] Попов Р.С., Ляликова В.Г. Шифрование информации в базе данных. Вестник науки, №12 (57) том 1, статья № 65, 2022.
- [2] Информационная безопасность персональных данных. Integrus. Москва, 2019.
- [3] CRYPTOPRO. Документация по CRYPTOPRO JCP, Версия 5.0. Москва, 2023.
- [4] Chourasia D. Encryption and decryption of data at blazing speed using SPRING DATA JPA. MEDIUM.COM, 2022.
- [5] Sultanov D.R. Database column-level encryption with SPRING DATA JPA. BLOG, 2019.
- [6] КУЗНЕЧИК (ШИФР). Википедия – свободная энциклопедия. Доступ: [https://ru.wikipedia.org/wiki/Кузнечик_\(шифр\)](https://ru.wikipedia.org/wiki/Кузнечик_(шифр))
- [7] ГОСТ 28147-89. Википедия – свободная энциклопедия. Доступ: https://ru.wikipedia.org/wiki/ГОСТ_28147-89.
- [8] TAXCOM. Переход на ГОСТ 34.12-2018 в документообороте. TAXKOM, 2024.

Статья получена 5 августа 2025 г.

Донецкий Сергей Юрьевич – ассистент Высшей инженеринговой школы НИЯУ МИФИ, e-mail: SYDonetskii@mephi.ru
 Андриенко Юрий Анатольевич – доцент Высшей инженеринговой школы НИЯУ МИФИ, YAAndrienko@mephi.ru

Automated Data Encryption Library Based on State Standard (GOST) Algorithms for Spring Data JPA

S.U. Donetskii Y. A. Andryenko

Abstract— *This paper introduces a software library that enables transparent, application-level encryption and decryption of sensitive data fields while remaining fully compliant with Russian cryptographic regulations. The solution targets growing information-security requirements for personal-data protection and aligns with domestic legislation governing cryptographic safeguards. We survey current techniques for encrypting database fields—from DBMS-native mechanisms to object-relational mapping (ORM) extensions—and identify their limitations in the context of Russian GOST standards. The proposed library acts as a lightweight wrapper around Spring Data JPA (with optional direct Hibernate integration). Data members marked with a custom annotation are automatically encrypted on persistence and decrypted on retrieval, without altering business logic. The architecture leverages JPA attribute converters and event listeners to intercept serialization and deserialization, integrates seamlessly into the Spring Data lifecycle, and incorporates flexible key-management facilities. Particular attention is paid to GOST 28147-89 and GOST R 34.12-2015 (“Magma” and “Kuznechik”) block-cipher algorithms, their operating modes, and suitability for database-encryption workloads. We also discuss the use of certified cryptographic providers such as CryptoPro CSP to satisfy regulator requirements. A comparative analysis with existing solutions highlights the advantages and scientific novelty of the*

proposed approach, demonstrating its ability to deliver fine-grained, standards-compliant security with minimal impact on developer productivity or system performance.

Keywords—*data encryption, GOST 28147-89, GOST R 34.12-2015, personal data, Spring Data JPA, cryptography, information security, CryptoPro.*

REFERENCES

- [1] Popov R.S., Lyalikova V.G. Encryption of Information in Databases. Vestnik Nauki, 2022.
- [2] Information Security of Personal Data. Integrus, Moscow, 2019.
- [3] CryptoPro. Documentation for CryptoPro JCP, Version 5.0. Moscow, 2023.
- [4] Chourasia D. Encryption and Decryption of Data at Blazing Speed Using Spring Data JPA. Medium.com, 2022.
- [5] Sultanov D.R. Database Column-Level Encryption with Spring Data JPA. Blog, 2019.
- [6] Kuznyechik (cipher). Wikipedia – The Free Encyclopedia. Available at: [https://ru.wikipedia.org/wiki/Кузнечик_\(шифр\)](https://ru.wikipedia.org/wiki/Кузнечик_(шифр))
- [7] GOST 28147-89. Wikipedia – The Free Encyclopedia. Available at: https://ru.wikipedia.org/wiki/ГОСТ_28147-89
- [8] Taxcom. Transition to GOST 34.12-2018 in Electronic Document Management. Taxcom, 2024.