ПРОГРАММИРОВАНИЕ, 2023, № 4, с. 39–57

ИНФОРМАЦИОННАЯ БЕЗОПАСНОСТЬ

УДК 004.43

ВАРИАНТ РЕАЛИЗАЦИИ ПРОЦЕДУРЫ АНАЛИЗА ИНФОРМАЦИОННЫХ ПОТОКОВ В ПРОГРАММНЫХ БЛОКАХ PL/SQL С ИСПОЛЬЗОВАНИЕМ ПЛАТФОРМЫ PLIF

© 2023 г. А. А. Тимаков[a],* (ORCID: 0000-0003-4306-789X)

[a]МИРЭА – Российский технологический университет
119454 Москва, Проспект Вернадского, д. 78, Россия

*****E-mail: timakov@mirea.ru*

Поступила в редакцию 15.07.2022 г.
После доработки 15.12.2022 г.
Принята к публикации 13.01.2023 г.

Формальное доказательство эффективности реализуемых мер защиты и безопасности вычислений (обработки информации) является важнейшим условием доверия к критическим информационным системам. Важно понимать, что при построении таких систем формальная проверка безопасности должна применяться на всех инфраструктурных уровнях (от физического до прикладного). В настоящее время на практике проблемой остается формальная проверка безопасности вычислений на прикладном уровне, требующая сложной разметки элементов среды вычислений. Для решения данной задачи традиционно используются методы, в основе которых лежит контроль информационных потоков (КИП). В отличие от методов на основе управления доступом, нашедших широкое практическое применение в операционных системах (ОС) и системах управления базами данных (СУБД), КИП в программном обеспечении на практике применяется весьма ограниченно и в основном сводится к анализу помеченных данных – Taint Tracking. В работе приводится вариант реализации КИП в программных блоках PL/SQL с использованием платформы PLIF. Кроме того, описана общая схема проверки безопасности вычислений в приложениях уровня предприятия, работающих с реляционными базами данных. Преимуществом подхода можно считать явное разделение функций разработчиков программного обеспечения и аналитиков безопасности.

DOI: 10.31857/S0132347423040118, **EDN:** RDRCOC

1. ВВЕДЕНИЕ

Требования к компьютерным системам заданного класса защиты определяются соответствующими оценочными стандартами. Первым подобным стандартом стали “Критерии определения безопасности компьютерных систем” Министерства обороны США [1]. В современном мире требования по безопасности формулируются в терминах “Общих критериев оценки защищенности информационных технологий” [2] и делятся на функциональные требования и требования доверия. Анализ показывает, что существует три важных категории условий, учитываемых при определении класса защиты компьютерной системы, в общем виде их можно сформулировать следующим образом:

1. Формальное доказательство эффективности реализованных механизмов защиты (модулей, реализующих функции защиты) или безопасности вычислений (обработки информации).

2. Контроль скрытых каналов.

3. Контроль “легальных” траекторий и сред распространения информации (управление доступом в случае ОС).

Далее для наглядности ограничимся грубым разбиением абстрактной компьютерной системы на три уровня: физический, системный и прикладной. Будем полагать, что физический уровень включает оконечные рабочие станции – пользовательские и серверные, а также сетевое оборудование.

Сложность выполнения обозначенных выше условий на практике возрастает от физического к прикладному уровню.

На физическом уровне “легальные” траектории распространения информации включают пользователей компьютерной системы, получающих доступ в выделенные помещения и взаимодействующих со средствами вычислительной техники посредством стандартных интерфейсов ввода-вывода. Контроль такого взаимодействия на практике успешно осуществляется путем реализации определенных организационно-техниче-

ских мер: ограничением лиц, допущенных в выделенные помещения и к работе с установленными там средствами вычислительной техники, внедрением систем контроля и управления доступом (СКУД), установкой систем видеонаблюдения и сигнализации, выполнением требований к блокировке входных и оконных проемов и т.д. Контроль скрытых акустических, электромагнитных и оптических каналов на физическом уровне (также называемых техническими каналами) реализуется специальными аппаратными средствами защиты информации.

Передача данных в компьютерных сетях осуществляется в соответствии со спецификациями сетевых протоколов. “Легальные” траектории распространения информации на этом уровне ограничиваются фильтрацией трафика по ip-адресам и номерам портов, реже – по полезному содержимому сетевых пакетов, сегментацией сетей; на канальном уровне – жесткой привязкой сетевых устройств к локальной вычислительной сети или портам коммутатора (mac filtering, port security). Возможные скрытые каналы по памяти (передача данных в заголовках сетевых пакетов, организация туннелей) контролируются системами обнаружения компьютерных атак, системами предотвращения утечек данных (DLP) и др. Для предотвращения более сложных скрытых каналов, связанных с манипулированием параметрами сигнала, также применяются специальные программно-аппаратные средства защиты информации.

Объекты файловых систем и баз данных являются стандартными контейнерами для хранения информации на системном уровне. Соответственно основным механизмом, обеспечивающим безопасную работу с такими объектами, выступает управление доступом. Современные операционные системы включают механизмы управления доступом, построенные на основе верифицированных моделей, таких, как например МРОСЛ ДП-модель [3]. Контроль скрытых каналов на системном уровне, по сути, осуществляется таким же способом. Практическая реализация монитора безопасности ОС в этом случае предполагает разделение процессов на доверенные и недоверенные. При этом доверенные процессы реализуют функции безопасности и не вступают в кооперацию с недоверенными при реализации запрещенных информационных потоков. Возможность нарушения целостности программ, инициирующих доверенные процессы исключается. Такой подход, на наш взгляд, позволяет решить проблему контроля скрытых каналов в системном программном обеспечении лишь частично. Скрытые каналы, по сути, имеют место в случаях, когда в соответствующей среде передачи (обработки) данных возникают какие-либо эффекты, различимые потенциальным нарушителем. Далеко не все эффекты, возникающие в системном программном обеспечении, имеют отношение к объектам файловой системы, баз данных или сетевым сокетам. Примерами подобных эффектов могут выступать: терминальное состояние программы, временные задержки, возникающие в процессе вычислений, интенсивность использования системных ресурсов, системный кэш и кэш базы данных – структуры, которые могут участвовать в межпроцессном взаимодействии. Второй проблемой, которая уже отмечалась в [4], является доверие к несистемным сервисам, осуществляющим деклассификацию – контролируемое раскрытие информации.

Наибольшую сложность представляет проверка безопасности вычислений на прикладном уровне. На практике проблема полноценно не решается даже в контексте контроля “легальных” траекторий распространения информации. В программном обеспечении такие траектории можно получить из графа потока управления при условии гарантии его целостности. Традиционно с целью обеспечения конфиденциальности и целостности данных в процессе их обработки в программном обеспечении используется сочетание формальных, полуформальных и неформальных методов, используемых на разных стадиях разработки системы. К таким методам относятся: динамический и статический анализ кода, символьное (косимвольное) выполнение, фаззинг-тестирование, формальная верификация. Кроме того, дополнительные защитные меры (рандомизация смещений динамической памяти, выделяемой процессу, защита от исполнения на стеке, контроль целостности потока управления и др.) реализуются на уровне платформы и компилятора. Проверки, реализуемые с использованием перечисленных методов, в основном, не учитывают специфику обрабатываемых данных и бизнес-логику приложения, но играют существенную роль в обеспечении целостности алгоритма. К формальным методам, учитывающим специфику данных, относятся, как говорилось ранее, методы на основе КИП. Полноценное внедрение КИП в языковые платформы в настоящее время не реализовано по ряду причин (см. [4]), однако при разработке защищенных систем все чаще применяется родственный механизм – анализ помеченных данных (Taint Tracking), иногда его относят к одному из видов КИП. Существенным ограничением анализа помеченных данных является его применимость лишь для выявления явных информационных потоков:

Таблица 1. Практическая реализация требований доверия к КС

	Физический уровень	Системный уровень	Уровень специального программного обеспечения
Контроль “легальных” траекторий и сред распространения информации	+	+	±
Контроль скрытых каналов	+	±	–
Формальное доказательство эффективности механизмов ЗИ	+	±	–

```
voidmain (){    voidfoo(z){
    a = new A();    x = z.g;
    b = a.g         w = source() ;
    foo(a);         x.f = w;
    sink ( b.f );   }
}
```

При этом неявные информационные потоки [5], которые, очевидно, также представляют собой “легальные” траектории распространения информации, легко могут быть использованы для обхода данного вида анализа:

```
void copy ( tainted char *src ,
            untainted char *dst , int len) {
  untainted int i, j ;
    for (i = 0; i < len; i++) {
    for (j = 0; j < sizeof(char)*256; j++) {
        if ( src[i] == (char)j )
            dst[i] == (char)j ; //illegal
    }
  }
}
```

Таким образом, с учетом изложенного выше, на текущий момент возможность выполнения условий, критичных для построения доверенных компьютерных систем, в целом может быть представлена как показано в табл. 1. Иные требования, например к резервированию данных [6], в работе воспринимаются как вспомогательные с точки зрения сложности реализации в конкретной системе.

Работа является логическим продолжением и дополнением исследований, описанных в [4], в ней детализирована процедура выявления запрещенных информационных потоков в программных блоках *PL/SQL* СУБД *Oracle* с использованием разработанной при участии автора платформы *PLIF* [7]. Основная идея заключается в формальной верификации сгенерированных на основе реализующих критичные вычисления программных блоков спецификаций *TLA+* инструментом “проигрывания моделей” – *TLC*. Принимая во внимание известное ограничение базового метода – существенное падение производительности с расширением пространства состояний [8], предполагается, что для адекватного абстрагирования проверяемых модулей критичные вычисления будут выноситься разработчиками на отдельный уровень, в нашем случае – уровень хранимых процедур *PL/SQL* (рис. 1). Выбор *PL/SQL* обусловлен несколькими факторами: близость к данным, относительно небольшой объем кода (как правило, не более 60 строк для одной процедуры), отсутствие вспомогательных вычислений, наличие встроенного механизма выявления прямых и косвенных зависимостей и др. Распространение информации от проверенных хранимых процедур и функций до соответствующих элементов графического интерфейса или выходных потоков во внешнем программном слое может жестко контролироваться статическим Taint Tracking анализом. Однако, данный вопрос требует дополнительного изучения. В работе также описана общая схема проверки безопасности вычислений в приложениях уровня предприятия, работающих с реляционными базами данных. Концептуальные сведения о платформе *PLIF* изложены в следующем разделе, после чего рассматривается пример использования платформы для проведения исследования.

2. PLIF. ОБЩИЕ СВЕДЕНИЯ

Внедрение механизма КИП, по мнению автора, должно включать следующие этапы:

– выбор языка описания политики безопасности (уровни и метки доступа, роли и привилегии и т.д.);

– определение безопасности семантики;

– разработку механизма проверки условий безопасности и доказательство его корректности;

– практическую реализацию;

– исследование влияния на написание кода и встраивание анализа в процесс разработки программного обеспечения.

Активные исследования по отдельным направлениям продолжаются уже несколько деся-

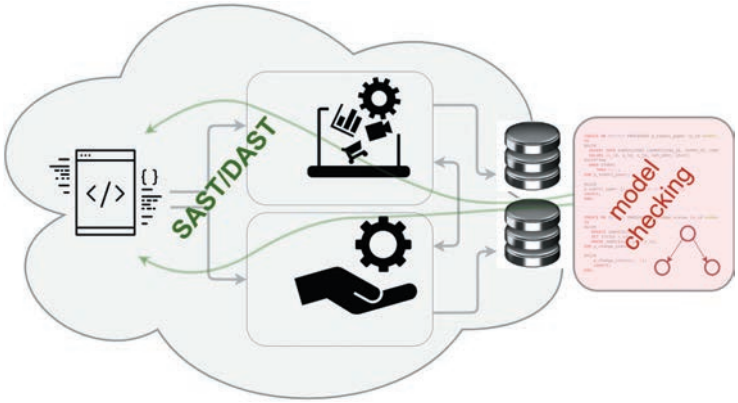


Рис. 1. Выбор приемлемого уровня абстрагирования.

тилетий. Ретроспективный взгляд на них изложен в [4]. Применительно к *PLIF* данные аспекты предстают в следующем виде.

Язык описания политики безопасности. В качестве языка описания политики безопасности выбрано подмножество языка *Paralocks* [9]. К основным его преимуществам следует отнести возможность встраивания в выражения политик условий деклассификации данных, гибкость и возможность интеграции с различными системами управления доступом (ролевыми, мандатными и др.).

В общем виде, как показано в [9], некоторая политика безопасности P_k определяется формулой логики предикатов, представимой в виде конъюнкции определенных дизъюнктов Хорна: $P_k \stackrel{\Delta}{=} C_1 \wedge C_2 \wedge \ldots$, где C_n – предложение политики вида: $\forall x_1,\ldots,x_m . l_1(\cdot) \wedge l_2(\cdot) \wedge l_3(\cdot)\ldots \Rightarrow Flow(u)$ или $\forall x_1,\ldots,x_m . \neg l_1(\cdot) \vee \neg l_2(\cdot) \vee \neg l_3(\cdot)\ldots \vee Flow(u)$, где $Flow(u)$ – предикат, обозначающий поток данных к u (u – связанная переменная или константа, обозначающая пользователя), $l_1 \ldots l_n$ – условия (блокировки), выполнение которых требуется для того, чтобы $Flow(u)$ принял значение $TRUE$. Предикаты $l_1 \ldots l_n$ могут быть параметрическими или непараметрическими.

Для дальнейших рассуждений удобно использовать для предложений политик клаузальную форму: $C_n \triangleq (\neg l_1(\cdot), \neg l_2(\cdot), \neg l_3(\cdot)\ldots, Flow(u))$.

Описание политик в модели (спецификации *TLA+*) требует введения ряда констант: UU – множество допустимых имен связанных переменных, U – множество актеров (конкретных пользователей системы), E_0 – множество имен непараметрических блокировок, E_1 – множество имен параметрических (с одним параметром) блокировок.

В качестве примера рассмотрим выражение политики: “Поток к произвольному пользователю x возможен, если а) открыта блокировка *guest* – для x задана роль *guest* – и открыта блокировка *t_expire* – истек заданный интервал времени **или** б) открыта блокировка *manager* – для x задана роль *manager*. Логически оно может быть представлено как $\forall x.(texpire \wedge guest(x) \Rightarrow Flow(x)) \wedge (manager(x) \Rightarrow Flow(x))$. В спецификации, полученной с помощью *PLIF* [7], данное выражение примет вид:

{⟨*x* , ⟨[*t_expire* ↦ {}],
[*guest* ↦ {*x*}, *reviewer* ↦ {*NONE*},
manager ↦ {*NONE*}, *organizer* ↦ {*NONE*}]⟩⟩,
⟨*x* , ⟨[*t expire* ↦ {*NONE*}],
[*guest* ↦ {*NONE*}, *reviewer* ↦ {*NONE*},
manager ↦ {x}, *organizer* ↦ {*NONE*}]⟩⟩}

Здесь: $x \in U$, $t\_expire \in E_0$, $\{guest, reviewer, manager, organizer\} \subseteq E_1$, $NONE$ – специальное значение, которое обозначает, что открытие блокировки не требуется. Политика определяется как множество двумерных кортежей. Каждый кортеж – отдельное предложение политики, его первым элементом является связанная переменная или константа, обозначающие пользователя, к которому разрешен информационный поток; второй элемент – это двумерный кортеж, с его помощью описывается множество блокировок, которые должны быть открыты. Элементами этого кортежа являются записи, их поля соответствуют именам блокировок, определяемых константами модели, значения полей задают множества фактических параметров блокировок. В *TLA+* записи, кортежи, массивы могут интерпретироваться как функции. Поскольку *TLA+* не поддерживает частично определенных функций, для блокировок, которые не участвуют в выражениях политики, приходится использовать специальные значения $NONE$. Для удобства анализа трасс,

приводящих к возникновению запрещенных информационных потоков, *PLIF* включает отдельную утилиту с графическим пользовательским интерфейсом – *plifparser*. Отображения политик в ней происходит в соответствии с упрощенной нотацией, в которой рассматриваемый пример выглядит так:

```
x: manager(x)
x: guest(x), t_expire
```

Отношение частичного порядка $\sqsubseteq$ на множестве политик задается следующим образом: $P_1 \sqsubseteq P_2$, если $\forall c_2 \in P_2 : \exists c_1 \in P_1 : c_1 \sqsubseteq c_2$ (1). С точки зрения логики $P_1 \sqsubseteq P_2$ можно интерпретировать, как $P_1 \vDash P_2$ (2). В [9] показано, что условие (1) является необходимым и достаточным для истинности выражения (2). Сравнение предложений политик в [9] описывается алгоритмически, через набор правил. Однако определение частичного порядка на данном множестве может иметь формальную замкнутую форму.

Пусть $C_1 = \{\neg l_1(\cdot), \neg l_2(\cdot), \ldots, \neg l_n(\cdot), Flow(u_1)\}$, $C_2 = \{\neg m_1(\cdot), \neg m_2(\cdot), \ldots, \neg m_k(\cdot), Flow(u_2)\}$, тогда $C_1 \sqsubseteq C_2$, если $\{\neg l_1(\cdot)\tau, \neg l_2(\cdot)\tau, \ldots, \neg l_n(\cdot)\tau, Flow(u_1)\tau\}\sigma \subseteq \{\neg m_1(\cdot), \neg m_2(\cdot), \ldots, \neg m_k(\cdot), Flow(u_2)\}\sigma$, где τ – операция переименования связанных переменных в термах (блокировках) соответствующего предложения политики, σ – наиболее общий унификатор (НОУ) $Flow(u_1)$ и $Flow(u_2)$ при условии, что u_1 и u_2 – связанные переменные или u_2 – константа. Операция определения НОУ является стандартной, ее подробное описание можно найти в литературе, посвященной методу резолюций.

Максимальная нижняя грань выражений политик определяется через объединение их предложений: $GLB(P_1, P_2) \triangleq P_1 \cup P_2$.

Минимальная верхняя грань в общем виде определяется следующим образом:

$$LUB(P_1, P_2) \triangleq$$

$$\triangleq \bigcup_{\substack{(\neg l_1(\cdot),\ldots\neg l_n(\cdot), Flow(u_1),\\ \neg m_1(\cdot),\ldots\neg m_k(\cdot), Flow(u_2)) \in P_1 \times P_2:\\ \exists\sigma.\sigma=(Flow(u_1), Flow(u_2))}} (\neg l_1(\cdot)\tau, \ldots, \neg l_n(\cdot)\tau, Flow(u_1)\tau, \neg m_1(\cdot), \ldots, \neg m_k(\cdot), Flow(u_2))\sigma$$

В данном случае при поиске НОУ никаких ограничений на параметр $Flow(\cdot)$ не накладывается.

Описанные в [9–11] варианты использования *Paralocks* для кодирования политик элементов среды вычислений, преемственных к существующим механизмам управления доступом (например, ролевому), позволяют сделать вывод о возможности внесения допущений, которые упрощают описание соответствующих сущностей в спецификациях *TLA+*. Положим, что множество имен связанных переменных *UU* включает один элемент, также будем считать, что, если связанная переменная появляется в одной из блокировок некоторого предложения политики, то эта же переменная является аргументом и в литерале $Flow(\cdot)$ того же предложения, и, если связанная переменная является аргументом $Flow(\cdot)$ некоторого предложения политики, то все параметрические блокировки того же предложения также будут принимать эту переменную в качестве аргумента.

С учетом заданных предположений, а также того, что $Flow(\cdot)$ является однопараметрической блокировкой, и в качестве ее параметра может выступать либо связанная переменная, либо константа, имеем:

$$LUB(P_1, P_2) \triangleq$$

$$\cup\{(\neg l_1(\cdot), \ldots, \neg l_n(\cdot), \neg m_1(\cdot), \ldots, \neg m_k(\cdot), Flow(x)) :$$
$$\wedge(\neg l_1(\cdot), \ldots, \neg l_n(\cdot), Flow(x)) \in P_1$$
$$\wedge(\neg m_1(\cdot), \ldots, \neg m_k(\cdot), Flow(x)) \in P_2\}$$
$$\cup\{(\neg l_1(\cdot), \ldots, \neg l_n(\cdot), \neg m_1(\cdot), \ldots, \neg m_k(\cdot), Flow(a)) :$$
$$\wedge(\neg l_1(\cdot), \ldots, \neg l_n(\cdot), Flow(a)) \in P_1$$
$$\wedge(\neg m_1(\cdot), \ldots, \neg m_k(\cdot), Flow(a)) \in P_2\}$$
$$\cup\{(\neg l_1(\cdot), \ldots, \neg l_n(\cdot),$$
$$\neg m_1(\cdot)[x := a], \ldots, \neg m_k(\cdot)[x := a], Flow(a)) :$$
$$\wedge(\neg l_1(\cdot), \ldots, \neg l_n(\cdot), Flow(a)) \in P_1$$
$$\wedge(\neg m_1(\cdot), \ldots, \neg m_k(\cdot), Flow(x)) \in P_2\}$$
$$\cup\{(\neg l_1(\cdot)[x := a], \ldots,$$
$$\neg l_n(\cdot)[x := a], \neg m_1(\cdot), \ldots, \neg m_k(\cdot), Flow(a)) :$$
$$\wedge(\neg l_1(\cdot), \ldots, \neg l_n(\cdot), Flow(x)) \in P_1$$
$$\wedge(\neg m_1(\cdot), \ldots, \neg m_k(\cdot), Flow(a)) \in P_2\}$$

В табл. 2 с использованием упрощенной нотации показано несколько примеров работы оператора сравнения на множестве политик, в табл. 3 – вычисление минимальной верхней грани.

Множество возможных предложений политик *ClausesSet* и множество самих политик *PoliciesSet* с

Таблица 2. Пример работы оператора $\sqsubseteq$

P_1	P_2	$P_1 \sqsubseteq P_2$
x: manager(x)	x: reviewer(x)	FALSE
x: reviewer(x)	x: manager(x)	FALSE
x: manager(x)	x: manager(x), t_expire	TRUE
x: manager(x)	bob: manager(bob)	TRUE

Таблица 3. Пример работы оператора LUB

P_1	P_2	$P_1 \sqcup P_2$
x: manager(x)	x: reviewer(x)	x: manager(x), reviewer(x)
alice	x: reviewer(x)	alice: reviewer(alice)
x: manager(x) x: reviewer(x)	x: t_expire	x: manager(x), t_expire x: reviewer(x), t_expire
bob	alice: t_expire	$\top$

учетом изложенного выше в спецификации *Paralocks.tla* [7] определяется как:

ClausesSet $\triangleq$
{$c \in \{\langle u, \langle e0, e1\rangle\rangle$
: $u \in (U \cup UU)$,
$e0 \in [E0 \to \text{SUBSET } \{NONE\}]$,
$e1 \in [E1 \to ((\text{Subset } (U \cup UU))$
$\setminus \{\{\}\}) \cup \{\{NONE\}\}]\}$:
$\vee\, c[1] \in UU$
$\vee \wedge\, c[1] \notin UU$
$\wedge$ (UNION *Range*(c[2][2]))
$\cap\, UU = \{\}$

PoliciesSet $\triangleq$
{$p \in$ subset *ClausesSet* :
$\forall\, c1 \in p$:
$\neg\exists\, c2 \in p$:
$\wedge\, c1 \neq c2$
$\wedge \vee$ (*compareClause*($c1$, $c2$)
$\vee$ *compareClause*($c2$, $c1$))}
$\cup \{\{\}\}$

Определение операторов: *compareClause*($\cdot$), *comparePol*($\cdot$), в данной работе не разбирается. Заданное описанным образом конечное множество выражений политик образует полную решетку. Доказательство соответствующих свойств проведено с использованием формальной системы для вывода доказательств *TLAPS* в */plif_specs/plif_lattice/Paralocks.tla* (см. [7]).

THEOREM *Reflexivity* $\triangleq$
$\forall\, p \in$ *PoliciesSet* : *comparePol*(p, p)

THEOREM *Transitivity* $\triangleq$
$\forall\, p1, p2, p3 \in$ *PoliciesSet* :
$\wedge$ *comparePol*($p1$, $p2$)
$\wedge$ *comparePol*($p2$, $p3$) $\Rightarrow$ *comparePol*($p1$, $p3$)

THEOREM *Antisymmetry* $\triangleq$
$\forall\, p1, p2 \in$ *PoliciesSet* :
$\wedge$ comparePol($p1$, $p2$)
$\wedge$ *comparePol*($p2$, $p1$) $\Rightarrow p1 = p2$

THEOREM *ParalocksLattice* $\triangleq$
$\forall\, p1, p2 \in$ *PoliciesSet* :
$\wedge$ *comparePol*($p1$,*LUB*($p1$, $p2$))
$\wedge$ *comparePol*($p2$,*LUB*($p1$, $p2$))
$\wedge \forall\, y \in$ *PoliciesSet* :
$\wedge$ *comparePol*($p1$, y)
$\wedge$ *comparePol*($p2$, y)
$\Rightarrow$ *comparePol*(*LUB*($p1$, $p2$), y)

Безопасность семантики. За основу принимается понятие прогресс-зависимого информационного невлияния [12]. Данная схема имеет несколько отличительных особенностей по сравнению со строгой схемой информационного невлияния, более подробные сведения можно найти в [4]. С целью сохранить целостность изложения приведем все же формальное определение.

Определение 2.1 *Программа P удовлетворяет свойству прогресс-зависимого информационного невлияния для начального и конечного отображений множества переменных на множество меток безопасности M_1 и M_2 – ПЗИН$(P)_{M_1,M_2}$, если для любых двух состояний S_1 и S_2 среды вычислений, состоящих в отношении низкой эквивалентности относительно некоторого уровня конфиденциальности L при отображении M_1: а) каждый шаг вычислений сопровождается генерацией одинаковых наблюдаемых значений относительно уровня L или приводит к "расхождению" для обоих состояний; б) соответствующие финальные состояния – S_1' и S_2' находятся в отношении низкой эквивалентности относительно уровня L при отображении M_2* (рис. 2):

$$\text{ПЗИН}(P)_{M_1,M_2} \triangleq \forall L, S_1, S_2, o_1, o_2 : S_1 \approx_{M_1,L} S_2 \wedge$$
$$\wedge\, P(S_1) \downarrow \langle S_1', o_1\rangle \wedge P(S_2) \downarrow \langle S_2', o_2\rangle \Rightarrow \qquad (2.1)$$
$$\Rightarrow o_1 \approx_{L(d)} o_2 \wedge S_1' \approx_{M_2,L} S_2'$$

Здесь $o_1 \approx_{L(d)} o_2$ означает эквивалентность наблюдаемых серий значений (поведений) относительно уровня L (с учетом деклассификации). Подробнее понятие деклассификации данных рассматривается далее, а также в [4]. Предполагается что, два состояния среды вычислений находятся в отношении низкой эквивалентности относительно заданного уровня конфиденциальности L, если все пары одноименных элементов с меткой, не превышающей L, обладают одинаковыми значениями. Условие *б*) легко проверяется для отдельных выражений и команд *PL/SQL* с ис-

пользованием заданных для них правил абстрактной семантики и является важным для формального доказательства безопасности вычислений в целом в условиях потокочувствительности и отсутствия ограничений на количество исполняемых блоков в одном сеансе [4]. Важную роль в определении также играют начальное и конечное отображения множества элементов на множество меток (политик) безопасности M_1 и M_2, назовем их абстрактными состояниями. Например, в простейшем случае, когда в системе имеется лишь один сеанс, и множество программных блоков ограничено лишь одной процедурой вида:

```
PROCEDURE proc_1 ( hi number) IS
    var 1 number;
    f1 utl__file.file_type;
BEGIN
        select col_1 into var_1 from Tab1;
    f1 :=
    utl__file.file_open('DIR', 'file name' ,wb);
    utl_file.put(f1, var_1);
        update Tab1
    set col_1 = hi
    where id = n;
END;
```

в которой внешний объект **file_name** обладает неизменяемой политикой $\bot$, начальная политика глобальной переменной **col_1** – $\bot$[1] и политика формального параметра **hi** – $\top$, отсутствие запрещенных информационных потоков при однократном выполнении процедуры позволяет сделать вывод о безопасности вычислений лишь для начального абстрактного состояния M_1, в котором глобальная переменная **col_1** имеет политику $\bot$, но не $\top$. Поэтому безопасная композиция программного блока с собой невозможна. Предложенный механизм проверки условий безопасности вычислений учитывает это обстоятельство см. Раздел 3.

Проверка условий безопасности. Как описано в [4], механизм проверки основан на формальной верификации программных блоков методом “проигрывания моделей” – *model checking*. В качестве языковой платформы спецификаций выбран язык *TLA+*. Важными его преимуществами являются: полноценная поддержка темпоральной логики (дает возможность более точно описывать поведение системы), а также возможность проверки свойств переходов, что важно в контексте предложенного алгоритма проверки модели вычислений (см. Раздел 3). Свойство перехода CompInv, используемое в данном алгоритме, может быть выражено как:

[1] Под глобальными переменными моделируемой среды вычислений понимаются атрибуты отношений.

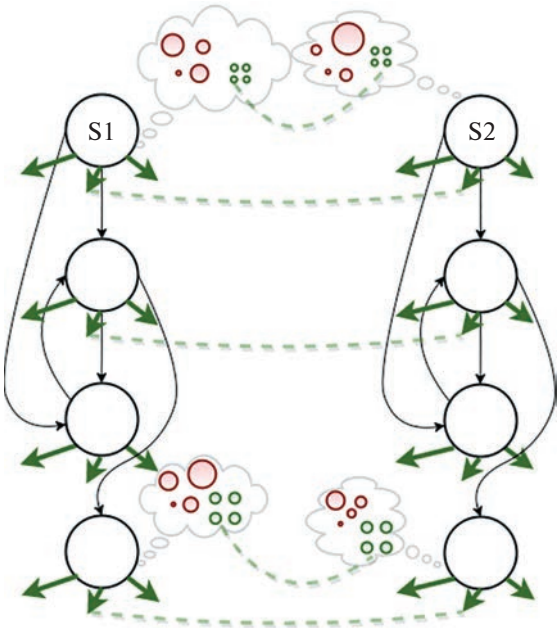


Рис. 2. Схема ПЗИН.

$$
\begin{array}{l}
VPolUnchanged \triangleq \\
\quad \text{LET } CompInv\_OP1(x, y) \triangleq \\
\qquad \wedge\ x \\
\qquad \wedge\ comparePol(VPol[y].policy,\ VPol'[y].policy) \\
\qquad \wedge\ comparePol(VPol'[y].policy,\ VPol[y].policy) \\
\quad \text{IN } FoldSet(CompInv\_\text{OP1}, \text{TRUE}, \text{DOMAIN } VPol) \\
\text{CompInv} \triangleq \Box[VPolUnchanged]_{vars}
\end{array}
$$

VPol – переменная состояния, содержащая политики глобальных переменных – столбцов. Кроме того, для проверки спецификаций *TLA+* методом “проигрывания моделей” специально разработан инструмент *TLC*.

Абстрактная семантика информационных потоков в программных блоках, выполняемых в параллельных пользовательских сеансах, представлена в [4]. Ее правила дополнены выражениями *inv*, символизирующими необходимые проверки на определенных этапах вычислений. Например, правило (C-IF):

$$
\frac{\langle e, M\rangle \Downarrow p \quad \text{inv} : e - local}{\left\langle PC, \begin{array}{l}\textbf{if } e \textbf{ then } c_1 \\ \textbf{else } c_2\end{array}, M\right\rangle \rightarrow \langle p :: PC, c_1 \vee c_2, M\rangle}
$$

говорит о том, что при выполнении условного оператора политика элемента *PC* – счетчика команд – дополняется политикой p условного выражения e, далее осуществляется равновероятный переход либо к инструкции c_1, либо к ин-

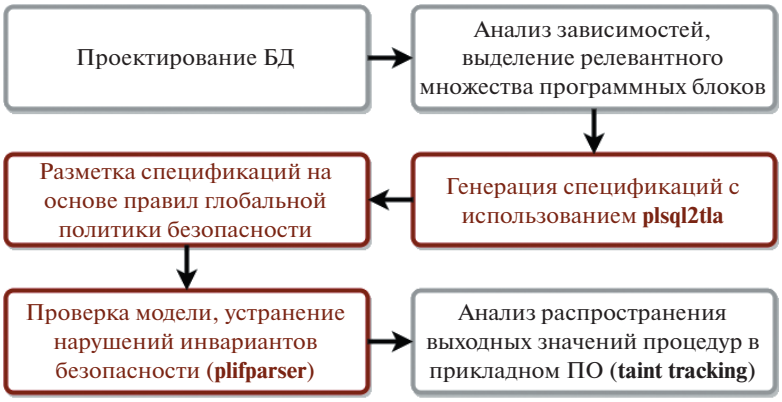


Рис. 3. Общая процедура исследования.

струкции c_2, абстрактное состояние среды вычислений при этом не изменяется. Для удобства манипулирования политикой PC с учетом возможной вложенности она представляется как кортеж отдельных значений. Очевидно, актуальное значение политики PC для каждого сеанса рассчитывается как минимальная верхняя грань входящих в соответствующий кортеж выражений. Ограничение $e - local$ (проверяется вручную) необходимо для исключения запрещенных вероятностных информационных потоков. Соответственно, правило (C-END-IF):

$$\frac{}{\langle p :: PC, \mathbf{endif}, M\rangle \rightarrow \langle PC, \mathbf{null}, M\rangle}$$

указывает лишь на ослабление политики PC при выходе из условия.

Общее описание системы переходов, моделирующей параллельные вычисления в сеансах работы с базой данных (БД), также можно найти в [4]. Реализованный анализ, как уже отмечалось, является потокочувствительным.

Важным этапом проверки является автоматическая трансформация кода PL/SQL в спецификации $TLA+$. Эта задача решается с использованием разработанной утилиты *plsql2tla*. В ее основе лежит подход, предложенный в [13].

В случае нарушения заданного инварианта безопасности или свойства перехода проблемную трассу и граф информационных потоков можно исследовать с помощью инструмента *plifparser*. Данный инструмент был разработан на основе [14]. В четвертом разделе показан пример его использования.

3. ОБЩАЯ ПРОЦЕДУРА ИССЛЕДОВАНИЯ

В общем виде этапы процедуры анализа информационных потоков в программном обеспечении промышленной информационной системы, построенной на основе СУБД, представлены на рис. 3.

Проектирование базы данных рекомендуется осуществлять таким образом, чтобы конфиденциальные данные размещались компактно – в ограниченном наборе таблиц. Данная рекомендация не противоречит общепринятым принципам безопасной разработки, и при этом позволит более эффективно изолировать критичные вычисления от общего кода PL/SQL.

Следующим этапом является анализ зависимостей и выделение релевантного множества программных блоков. На этом этапе предполагается определение тех блоков PL/SQL, которые имеют отношение к манипулированию конфиденциальными данными. Важной особенностью современных промышленных СУБД является наличие встроенных механизмов выявления прямых и косвенных зависимостей.

Ключевыми и наиболее сложными этапами процедуры являются: генерация и разметка спецификаций, проверка модели и устранение нарушений инвариантов безопасности. Данные этапы разберем далее на конкретном примере. Акцент будет сделан на алгоритме проверки модели вычислений (рис. 4). Предварительный шаг алгоритма Инициализация предполагает инициализацию констант, определяющих количество сеансов, множество конкретных пользователей, множество имен связанных переменных, множества параметрических и непараметрических блокировок и т.д. При проигрывании модели проверяются два свойства: главный инвариант безопасности *ParalocksInv* – гарантирует выполнение условия *a)* Определения 0.1 для отдельных команд и выражений – и свойство перехода *CompInv* – гарантирует эквивалентность начального и конечного отображений множества глобальных переменных на множество политик. В случае нарушения *ParalocksInv* принимается решение о деклассификации данных, в некоторых случаях требующей ис-

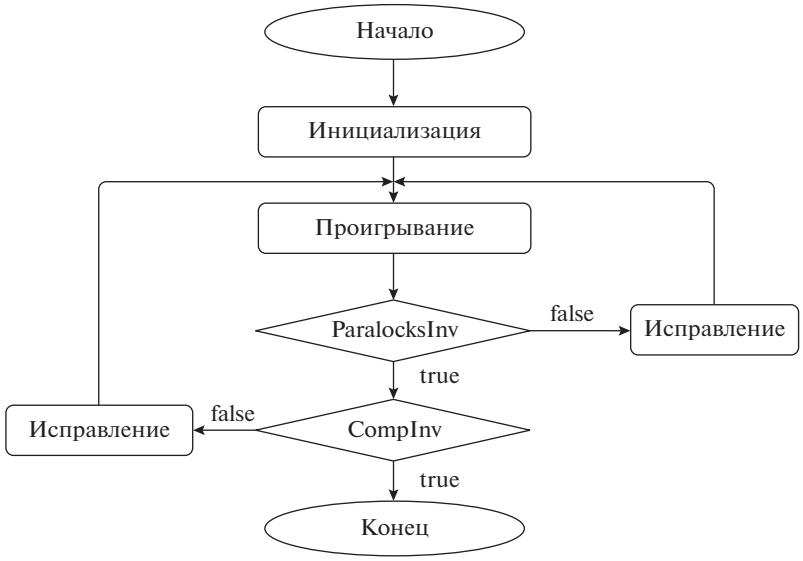


Рис. 4. Алгоритм проверки модели.

правления кода или изменения привилегий на доступ к объектам БД (рассматривается далее). При нарушении свойства *CompInv* в начальном состоянии модели повышается политика соответствующей глобальной переменной – столбца[2]. Приведение политик глобальных переменных к стационарным значениям позволяет проверить выполнение условия *a*) Определения 2.1 для отдельных команд и выражений *PL/SQL* для всех возможных начальных и конечных абстрактных состояний M_1 и M_2, что, в свою очередь, требуется для доказательства корректности алгоритма проверки модели вычислений.

В [4] доказано, что успешное завершение алгоритма гарантирует безопасность вычислений в смысле *ПЗИН* при неограниченном количестве программных блоков, выполняемых в каждом пользовательском сеансе.

Завершающим этапом процедуры является анализ распространения выходных значений критичных процедур и функций в прикладном программном обеспечении.

4. PLIF. ПРИМЕР

Для апробации *PLIF* использовался пример из [4]. Пусть имеется система управления конференциями. Пользователи системы регистрируют доклады, которые рецензируются и распределяются по секциям. На уровне СУБД *Oracle* действует ролевой механизм управления доступом. Иерархия ролей представлена на рис. 5.

Основные бизнес-функции в системе реализуются с помощью хранимых программных блоков *PL/SQL*. Общие требования по безопасности формулируются, например, следующим образом: любой зарегистрированный пользователь может подать заявку на участие в конференции с докладом. Анализ тезисов осуществляется рецензентами, которые, однако, не могут ассоциировать работы с авторами. Программа конференции формируется менеджером, ему доступен по чтению статус любого доклада (одобрен или нет), оставшимся пользователям статусы докладов могут быть доступны лишь по истечению заданного интервала времени. Программу конференции могут просматривать все пользователи. Авторы работ должны оставаться недоступными по чтению для всех пользователей до истечения определенного интервала времени.

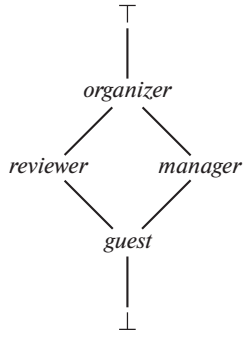


Рис. 5. Решетка ролей.

[2] Выполнения свойства CompInv можно добиться за конечное количество итераций, поскольку алфавит политик представляет собой конечную решетку, а переходные функции вычисления политик глобальных переменных являются монотонно возрастающими.

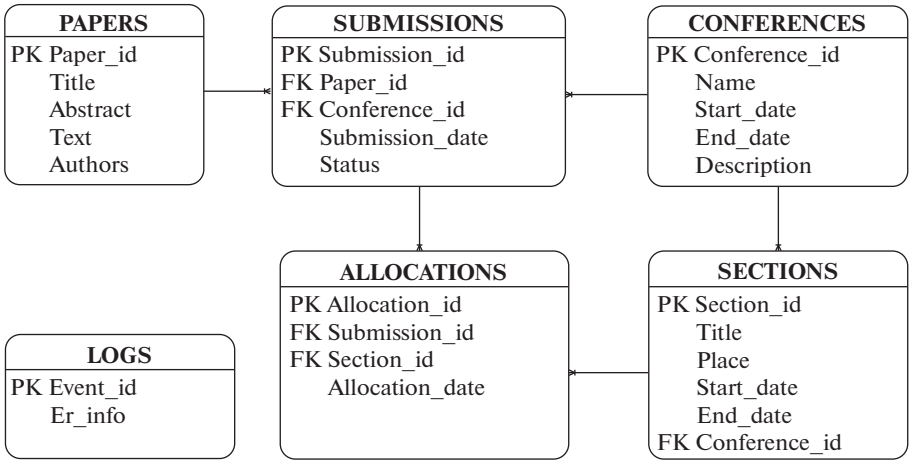


Рис. 6. Структура БД.

Пусть в системе запрещен прямой доступ к таблицам, взаимодействие возможно лишь с использованием хранимых процедур и функций. Положим также, что для поддержания описанных требований по безопасности на уровне БД заданы следующие объектные привилегии:

grant execute on p_add_paper to public;

grant execute on p_submit_paper to public;

grant execute on p_chahge_status to reviewer;

grant execute on p_allocate to manager;

grant execute on f_getsection_program to public;

grant execute on f_getpaper to public;

grant execute on f_is_accepted to public.

После завершения этапа **проектирования БД**, в соответствии с описанной процедурой (см. рис. 3), следует определить таблицы, предназначенные для хранения конфиденциальных данных, и выявить зависимые от них объекты БД. В СУБД *Oracle* для этих целей существует отдельный механизм. Например, для получения прямых и косвенных зависимостей, ассоциированных с таблицей *Submissions* – содержит статусы зарегистрированных для участия докладов – можно выполнить следующие команды:

execute deptree_fill('TABLE', 'CONFERENCE',
'SUBMISSIONS');

select * from deptree.

Положим, на **втором этапе** удалось выявить релевантное множество объектов БД. В него входят отношения: *Papers* – доклады, *Submissions* – зарегистрированные заявки на участие, *Conferences* – конференции, *Sections* – секции, *Allocations* – распределение докладов по секциям, *Logs* – журнал ошибок (рис. 6); и программные блоки: *p_add_paper* – добавляет новый доклад в БД, *p_submit_paper* – регистрирует доклад для участия в конференции (добавляет строку в таблицу *Submissions*), *p_chahge_status* – изменяет значение поля *Status* для заданной строки в таблице *Submissions* после рецензирования соответствующего доклада, *p_allocate* – добавляет сведения о докладе в таблицу *Allocations* после осуществления ряда проверок, *f_getsection_program* – возвращает программу заданной секции, для формирования отчета обращается к таблицам: *Papers* и *Allocations*, *f_get_paper* – возвращает сведения о заданном докладе, *f_is_accepted* – проверяет статус доклада и возвращает значение *TRUE*, если доклад одобрен. **Генерация спецификаций** осуществляется с помощью утилиты *plsql2tla*. В данной работе этот этап подробно не рассматривается.

Далее необходимо выполнить **разметку спецификаций** на основе правил глобальной политики безопасности. Как может выглядеть результат выполнения данного этапа, показано в табл. 4. Остановимся на этом немного подробнее.

Прежде всего, опираясь на заданные требования глобальной политики и иерархию ролей БД (рис. 5), определим множества параметрических и непараметрических блокировок E_1 и E_0. Пусть $E_0 \triangleq \{$“*t_expire*”$\}$, $E_1 \triangleq \{$“*guest*”,“*reviewer*”, “*manager*”, “*organizer*”$\}$. К процедуре *p_submit_paper* администратором БД предоставлен общий доступ, поэтому для всех формальных параметров справедлива политика *any_caller* ($\boxed{a}$) или формально:

$$any\_caller\,(a) \triangleq$$
$$\{\langle a, \langle [e1 \in E0 \mapsto \{NONE\}],$$
$$[e2 \in E1 \mapsto \{NONE\}]\rangle\rangle\}$$

Таблица 4. Разметка спецификаций

Программный блок	Входные значения	Формальные параметры	Локальные переменные	Выходные значения
create or replace procedure p_submit_paper s_id number p_id number, c_id number, sub_date_date, stat number) is begin ... end p_add_paper;	**i1:=** x **i2:=** x **i3:=** x **i4:=** x **i5:=** x	**s_id :=** a **p_id :=** a **c_id :=** a **sub_date :=** a **stat :=** a		
create or replace. procedure p_add_paper (p_id number, tit varchar2, abstr varchar2, t clob, auth varchar2) is begin ... end p_add_paper;	**i1:=** x **i2:=** x **i3:=** x **i4:=** x **i5:=** x : t_expire	**p_id:=** a **tit :=** a **abstr :=** a **t :=** a **auth :=** a		
create or replace procedure p_change_status (s_id number, stat number) is begin ...; end p_change_status;	**i1:=** x **i2:=** x : manager(x) x : t_expire	**s_id :=** a : reviewer(a) **stat :=** a : reviewer(a)		
create or replace procedure p_allocate (id number, s_id number, sec_id number, alloc_date number) is p_not_accepted exception v_p_id exception v_is_acc exception begin ... end p_allocate;	**i1:=** x **i2:=** x **i3:=** x **i4:=** x	id := a : manager(a) s_id := a : manager(a) **sec_id :=** a : manager(a) **alloc_date:=** a : manager(a)	**p_not_accepted:=** x **v_p_id :=** x **v_is_acc :=** x	

Таблица 4. Окончание

Программный блок	Входные значения	Формальные параметры	Локальные переменные	Выходные значения
create type paper_type ...; create or replace function f_get_paper (p_id number) return paper_type as v_paper paper_type begin ... return v_paper; end f_get_paper;	**i1:=** x	**p_id:=** a	**v_paper_c1:=** x **v_paper_c2:=** x **v_paper_c3:=** x **v_paper_c4:=** x **v_paper_c5:=** x	**r_rec_c1:=** a **r_rec_c2:=** a **r_rec_c3:=** a **r_rec_c4:=** a **r_rec_c5:=** a

create or replace function
f_is_accepted
(s_id number)
return boolean
is
v_status number;
begin
select status
into v_status
from submissions
where
submission_id = s_id;
if v_status = 1
then return true;
else return false;
end if;
end f_is_accepted;
	i1:= x	**s_id :=** a	**v_status:=** x	**r:=** a

create type paper_type
as object(...);
create type
program_arr_type
as varray(1000)
of paper_type;
create or replace
function
f_get_section_program
(s_id number)
return program_arr_type
as
(v_program)
program_arr_type;
begin
...
end;
	i1:= x	**s_id:=** a	**v_prog_ar_e1_c1:=** x
			v_prog_ar_e1_c2:= x
			v_prog_ar_e1_c3:= x
			v_prog_ar_e1_c4:= x
			v_prog_ar_e1_c5:= x
			v_prog_ar_e2_c1:= x
			v_prog_ar_e2_c2:= x
			v_prog_ar_e2_c3:= x
			v_prog_ar_e2_c4:= x
			v_prog_ar_e2_c5:= x
				r_ar_e1_c1:= a
				r_ar_e1_c2:= a
				r_ar_e1_c3:= a
				r_ar_e1_c4:= a
				r_ar_e1_c5:= a
				r_ar_e2_c1:= a
				r_ar_e2_c2:= a
				r_ar_e2_c3:= a
				r_ar_e2_c4:= a
				r_ar_e2_c5:= a

Здесь a – символизирует элемент множества U, представляет собой конкретного пользователя сеанса. Описание общей политики безопасности также не содержит ограничений на какие-либо данные, вводимые пользователем при вызове процедуры, поэтому все входные значения обладают минимальной политикой x:

$$min \triangleq \{\langle \text{CHOOSE } x \in UU : \text{TRUE}, \langle [e1 \in E0 \mapsto \{NONE\}], [e2 \in E1 \mapsto \{NONE\}]\rangle\rangle\}$$

Разметка программного блока *p_add_paper* во многом аналогична. Исключение составляет входное значение параметра *auth*, ему назначается политика x: t_expire, поскольку в общих требованиях сказано, что сведения об авторах работ должны оставаться недоступными до истечения заданного интервала времени. Исходя из заданных привилегий на вызов процедуры *p_change_status* политика ее формальных параметров определяется как a: reviewer(a) при этом передаваемое при ее вызове фактическое значение статуса доклада $i2$ должно быть доступно по чтению *manager* и всем остальным пользователям по истечению временного интервала, следовательно $i2 :=$ x : manager(x) / x : t_expire. Входные значения процедуры *p_allocate* не являются конфиденциальными в соответствии с представленными выше требованиями, соответственно, обладают минимальной политикой. Политика формальных параметров a: manager(a) продиктована заданными на уровне системы объектными привилегиями. Отметим, что данная процедура также включает локальные элементы. Локальным переменным всегда назначается минимальная начальная политика, политика исключения *p_not_accapted* в данном случае также минимальна. Общедоступные функциии *f_get_paper* и *f_get_section_program* представляют интерес в части описания составных и ссылочных локальных переменных и возвращаемых значений. Так переменная *v_paper* и возвращаемое значение функции *f_get_paper* обладают объектным типом, состоящим из пяти полей. Каждое поле объектного типа или записи в *PLIF* моделируется отдельным элементом. Аналогично переменная *v_program* и возвращаемое значение функции *f_get_section_program* являются коллекциями объектов. Любая коллекция, ассоциированная с набором кортежей БД, в текущей версии *PLIF* моделируется двухэлементным массивом. Потеря точности автоматического анализа, как ожидается, может быть компенсирована удобством графического интерпретатора трасс вычислений, приводящих к нарушению инварианта безопасности (см. далее).

Ключевым этапом процедуры, конечно, является **проверка модели** и устранение нарушений инвариантов безопасности. В данном примере для завершения алгоритма потребовалось 8 итераций. Показанные далее графы информационных потоков были сгенерированы с помощью утилиты *plifparser*. Проигрывание модели осуществлялось в *TLC* при следующих значениях основных параметров инициализации (констант):

$U \triangleq \{allen, bob, allex, john\}$

$UU \triangleq \{\text{x}\}$

$E0 \triangleq \{\text{“t_expire”}\}$

$E1 \triangleq \{\text{“guest”}, \text{“reviewer”}, \text{“manager”}, \text{“organizer”}\}$

$GPol \triangleq$ (“organizer” :> {“manager”, “reviewer”, “guest”})@@
(“manager” :> {“guest”})@@
(“reviewer” :> {“guest”})@@
(“guest” :> {“guest”})

$Session\_number \triangleq 1$

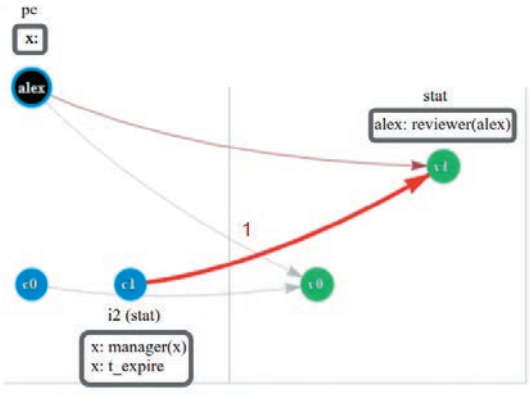
Здесь функция *GPol* задает отображение множества ролей на множество подмножеств ролей, соответствует иерархии на рис. 5 и позволяет осуществлять автоматическое открытие зависимых блокировок при проигрывании модели. *Session_number* – количество моделируемых параллельных сеансов.

ШАГ 1. Нарушение ParalocksInv

При первой попытке проигрывания модели возникает нарушение инварианта безопасности на этапе выполнения оператора *p_change_status_load* – загрузке в сеанс параметров и локальных переменных процедуры *p_change_status*[3] (рис. 7). Вычисления на данном шаге выполняются в соответствии с правилом (C-EXT-PRC) абстрактной семантики, описанной в [4]:

$$\frac{\langle e_1, M\rangle \Downarrow p_1 \ldots \langle e_n, M\rangle \Downarrow p_n \quad \langle x_1, M\rangle \Downarrow p_{x_1} \ldots \langle x_n, M\rangle \Downarrow p_{x_n} \quad \text{inv} : p_1 \sqsubseteq p_{x_1} \ldots p_n \sqsubseteq p_{x_n}}{\langle PC, x_f(x_1 \to e_1, \ldots x_n \to e_n), M\rangle \to \langle PC, \textbf{null}, M[x_1 \mapsto p_1, \ldots x_n \mapsto p_n]\rangle}$$

[3] Генерируемые *PLIF* спецификации описывают состояние каждого пользовательского сеанса, которое включает область локальных переменных и параметров – стек, указатели на текущий кадр стека и текущую выполняемую в сеансе инструкцию, а также область возвращаемых значений.



Session	TLA+	PLSQL
alex	p_change_status_load	change_status_load

Рис. 7. ParalocksInv is violated (p_change_status_load).

Причиной нарушения является несоответствие политик входного значения *i*2(*stat*) – `x : manager(x)` `x : t_expire` и формального параметра *stat* – `a : reviewer(a)`. Для устранения проблемы применяется процедура деклассификации. В литературе описано несколько принципов и способов деклассификации данных в программном обеспечении [15]. В данном случае предполагается, что рецензирование докладов осуществляется анонимно, и рецензент ограничен (на прикладном уровне) только текстом доклада (без имени автора и номера заявки), определяющим фактором является именно расположение проблемного выражения. Поэтому выявленное нарушение можно устранить, применив деклассификацию с ограничением по месту (*WHERE*). Для исправления модели достаточно в соответствующем операторе установить флаг *Ignore*, исправление кода самой процедуры не требуется.

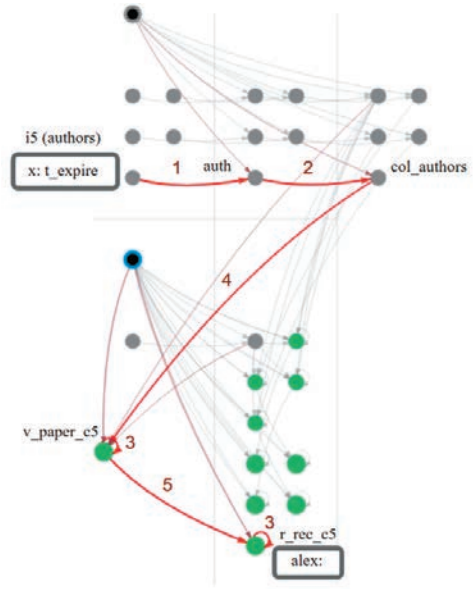
$p\_change\_status\_load(id) \triangleq \ldots \wedge Ignore' = 1$

$ParalocksInv \triangleq$ IF $Ignore \neq 1$
THEN ... ELSE TRUE

Другие допустимые в *PLIF* способы деклассификации применяются для исправления модели на следующих итерациях.

ШАГ 2. Нарушение ParalocksInv

После повторного запуска утилиты проигрывания модели инвариант безопасности нарушается при переходе в операторе *f_get_paper_8* (рис. 8). Анализ графа информационных потоков позволяет легко выявить проблемную траекторию: 1 – политика входного значения *i5 (authors)* –



bob	p_add_paper_load	add_paper_load
bob	p_add_paper4	insert into PAPERS (PAPER_ID, TITLE, ABSTRACT, TEXT, AUTHORS)values (p_id, tit, absr, t, auth)
bob	p_add_paper_exit	add_paper_exit
alex	f_get_paper_load	get_paper_load
alex	f_get_paper_5	select PAPER_ID, TITLE, ABSTRACT, TEXT, AUTHORS into v_paper from PAPERS where PAPER_ID = p_id
alex	f_get_paper_8	return v_paper

Рис. 8. ParalocksInv is violated (p_get_paper_8).

`x : t_expire` попадает в параметр *auth* процедуры *p_add_paper* (выполняется в процессе загрузки процедуры в сеансе *bob*), 2 – обновленная политика параметра *auth* передается в глобальную переменную (столбец) *col_authors*, 3 – происходит загрузка функции *f_get_paper* в сеансе *alex*, 4 – политика столбца *col_authors* передается во внутреннюю переменную *v_paper_c5*, 5 – политика переменной *v_paper_c5* передается в возвращаемое значение *r_rec_c5*, обладающее стационарной политикой `alex`. Таким образом, нарушается условие, заданное правилом (C-EXT-RET) абстрактной семантики:

$$\frac{\langle e, M\rangle \Downarrow p_1 \quad \langle x_{out}, M\rangle \Downarrow p_2 \quad \text{inv} : p_1 \sqcup pc_1 \ldots pc_n \sqsubseteq p_2}{\langle PC, \textbf{return}(x_{out} \to e), M\rangle \to \langle PC, \textbf{null}, M\rangle}$$

Для того, чтобы информационный поток в данной точке стал разрешенным, можно открыть

блокировку *t_expire*. Фактически это означает исправление кода процедуры путем добавления проверки условия:

```
create or replace function f_get_paper
...
is
   begin
      if is time expire()
         then ... else return null;
end p_get_paper;
```

Формально в этом случае применяется деклассификация *WHEN*. Исправление модели требует использования оператора *openLock*, который позволяет добавить во множество открытых блокировок сеанса новый элемент:

$f\_get\_paper\_load(id) \triangleq$

...

$\wedge\ openLock(id, \{[t\_expire \mapsto ALL]\})$

...

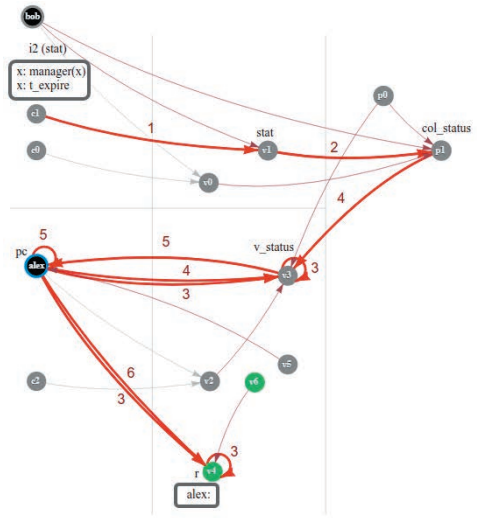
ШАГ 3. Нарушение ParalocksInv

Третье проигрывание модели также завершается нарушением инварианта безопасности, в этот раз ошибка возникает в операторе *f_is_accepted_9*. Поскольку доступ к этой функции предоставлен всем пользователям, ее возвращаемое значение имеет политику a (*any_caller*). Вариант трассы, приводящей к ошибке, показан на рис. 9.

1, 2 – политика входного значения *i2*(*stat*) `x: manager(x)` `x: t_expire` попадает в глобальную переменную *col_status* в сеансе *bob*. 3, 4 – политика глобальной переменной *col_status* передается во внутреннюю переменную *v_status* функции *f_is_accepted* в сеансе *alex*. 5 – в результате проверки условия *if* политика переменной *v_status* переходит в политику *pc_label* сеанса *alex*. 6 – политика *pc_label* передается в возвращаемое значение функции *f_is_accepted*, которая обладает стационарной политикой a. Таким образом, на последнем шаге трассы нарушается рассмотренное уже ранее условие, заданное правилом (C-EXT-RET) абстрактной семантики.

В данном случае, скорее всего, администратором БД была допущена ошибка при настройке правил разграничения доступа. Функция *f_is_accepted* не должна вызываться явно пользователями системы, или доступ к ней должен быть ограничен ролью *manager*. Корректировка объектных привилегий может выглядеть как:

```
revoke execute on f_is_accepted from PUBLIC;
grant execute on f_is_accepted to manager.
```

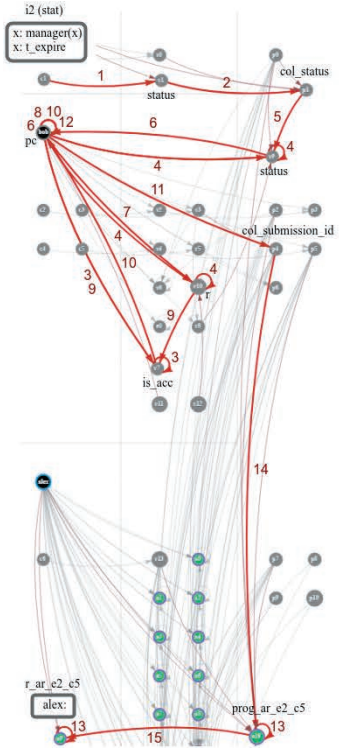


bob	p_change_status_load	change_status_load
bob	p_change_status4	update SUBMISSIONS set STATUS = stat where SUBMISSION_ID = s_id
bob	p_change_status_exit	change_status_exit
alex	f_is_accepted_load	is_accepted_load
alex	f_is_accepted5	select STATUS into v_status, from SUBMISSIONS where SUBMISSION_ID = s_id
alex	f_is_accepted8	if v_status = 1
alex	f_is_accepted9	then return TRUE

Рис. 9. ParalocksInv is violated (f_is_accepted_9).

Для исправления спецификации необходимо в формулу начального состояния *Init* внести изменения, при которых при запуске процедуры *f_is_accepted* в сеансе пользователя *s* во множество открытых блокировок сеанса *SLocks* добавлялись бы блокировки: *manager*(*s*) и *guest*(*s*):

$Init \triangleq \ldots \wedge SLocks =$
$[s \in S \mapsto$
$[e1 \in E0 \mapsto \{\}]$
$@@[e2 \in E1 \mapsto$
CASE . . .
if *f_is_accepted* is called
manager(*s*) and *guest*(*s*) are added
into the initial set of open locks
$\square \wedge Sessions[s][\text{“StateRegs”}][1][\text{“pc”}][1]$
$= \text{“ f\_is\_accepted”}$
$\wedge \vee e2 = \text{“manager”}$
$\vee e2 = \text{“guest”} \rightarrow \{s\}$
$\square \ldots$



allen	p_change_status_load	change_status_load
allen	p_change_status4	update SUBMISSIONS set STATUS = stat where SUBMISSION_ID = s_id
allen	p_change_status_exit	change_status_exit
bob	p_allocate_load	allocate_load
bob	f_is_accepted_call	is_accepted_call
bob	f_is_accepted5	select STATUS into v_status from SUBMISSIONS where SUBMISSION_ID = s_id
bob	f_is_accepted8	if v_status = 1
bob	f_is_accepted9	then return TRUE
bob	f_is_accepted11	endif
bob	f_is_accepted_exit	is_accepted_exit
bob	p_allocate7r	is_accepted_exit
bob	p_allocate_8	if v_is_acc
bob	p_allocate_10	then select paper_id into v_p_id from SUBMISSIONS where submission_id = s_id
bob	p_allocate_13	insert into ALLOCATIONS (ALLOCATION_ID, SUBMISSION_ID, SECTION_ID, ALLOCATION_DATE) values (id, s_id, sec_id, alloc_date)
bob	p_allocate16	endif
bob	p_allocate_exit	allocate_exit
alex	f_get_section_program_5	select paper_typ(PAPER_ID, TITLE, ABSTRACT, TEXT, 'UNKNOWN_AUTH' bulk collect into v_program from PAPERS from PAPERS where PAPER_ID IN (select PAPER_ID from ALLOCATIONS a join SUBMISSIONS s on a.submission_id = s.submission_id where a.SECTION_ID = s_id)
alex	f_get_section_program_9	return v_submissions

Рис. 10. ParalocksInv is violated (f_get_section_program_9).

При этом политику возвращаемого значения функции в *ParametersFS.tla* также следует изменить:

$$
\begin{aligned}
&f\_ia\_r(x) \triangleq \\
&\quad [loc \mapsto \text{“mem”}, \\
&\quad offs \mapsto 2, \\
&\quad policy \mapsto \text{replace } any\_caller(x) \text{ with} \\
&\qquad \{\langle x, \langle [t\_expire \mapsto \{NONE\}], \\
&\qquad\quad [guest \mapsto \{NONE\}, \\
&\qquad\quad reviewer \mapsto \{NONE\}, \\
&\qquad\quad manager \mapsto \{x\}, \\
&\qquad\quad organizer \mapsto \{NONE\}]\rangle\rangle\}, \\
&\quad name \mapsto \text{“ f\_ia\_r”}]
\end{aligned}
$$

ШАГ 4. Нарушение ParalocksInv

Четвертая попытка проигрывания модели также приводит к нарушению инварианта *ParalocksInv*. При выполнении оператора *f_get_section_program_9* (рис. 10) вновь нарушается условие, заданное правилом (C-EXT-RET).

Попробуем кратко описать проблемную траекторию распространения конфиденциальных данных. 1, 2 – политика входного значения $i2(stat)$ – `x: manager(x)` `x: t_expire` попадает в глобальную переменную *col_status* в сеансе *allen*. 3,4,5 – в сеансе *bob* загружается процедура *p_allocate* внутри нее происходит вызов функции *f_is_accepted*, и политика глобальной переменной *col_status* передается в локальную переменную *v_status*. 8 – политика переменной *v_status* добавляется к политике *pc_label* сеанса *bob*. 7, 8, 9 – политика `x: manager(x)` `x: t_expire` из элемента *pc_label* сначала передается в выходное значение функции *f_is_accepted*, а затем в локальную переменную *v_is_acc* процедуры *p_allocate*, при выходе из конструкции *if* указанная выше политика удаляется из общей политики элемента *pc_label*. 10 – политика `x: manager(x)` `x: t_expire` из переменной *v_is_acc* вновь попадает в *pc_label*. 11 – политика элемента *pc_label* сеанса *bob* передается в глобальную переменную *col_submission_id*. 13, 14 – в сеансе *alex* вызывается функция *f_get_section_program*, и в ее локальную переменную *v_program* (представляет собой массив объектов, состоящих из 5 полей) передается политика из глобальной переменной *col_submission_id*. 15 – политика одного из элементов массива *v_program* передается в один из элементов выходного массива функции *f_get_section_program*, который в соответствии с текущими настройками глобальной политики управления

доступом обладает стационарной политикой a. В данном случае аналитик легко заметит, что среди возвращаемых функцией значений имеются наименования статей. То есть любой пользователь независимо от выполнения заданного в описании требований по безопасности условия истечения временного интервала может получить список докладов, обладающих статусом “допущено”. Очевидно, выявленный запрещенный информационный поток не является ложным срабатыванием. Для устранения проблемы целесообразно применить ту же стратегию (деклассификация *WHEN*), что и на шаге 2 – внедрить в код процедуры проверку:

```
create or replace function f_get_section_program
...
is
begin
  if is time expire() or is manager()
    then ... else return null;
end p_get_section_program;
```

и исправить модель, добавив во множество открытых блокировок сеанса, в котором вызывается функция *f_get_section_program*, необходимые элементы:

$f\_get\_section\_program\_load(id) \triangleq$
...
$\wedge \vee openLock(id, \{[t\_expire \mapsto ALL]\})$
$\vee openLock(id, \{[manager \mapsto id]\})$
...

Нарушение инварианта не всегда свидетельствует о наличии действительно опасного информационного потока. Предложенный анализ, как уже отмечалось, не является абсолютно точным. Однако предполагается, что любую проблемную траекторию можно легко исследовать с использованием графической утилиты *plifparser*. Например, если в последнем случае (рис. 10) запрос *SELECT* вместо полных сведений о докладе возвращал бы только идентификатор *paper_id*, который по мнению аналитика невозможно ассоциировать с конкретной работой, или общее количество докладов в заданной секции, то возвращаемое значение можно было бы деклассифицировать без дополнительных проверок (деклассификация *WHAT*):

---

select(*id*, ...,
Replace the value calculated using *LUB*
LUB4Seq
(⟨*VPol*[“*col_papers_paper_id*”].*policy*,
VPol[“*col_allocations_submission_id*”].*policy*,
VPol[“*col_allocations_section_id*”].*policy*,
VPol[“*col_submissions_paper_id*”].*policy*,
VPol[“*col_submissions_submission_id*”].*policy*,

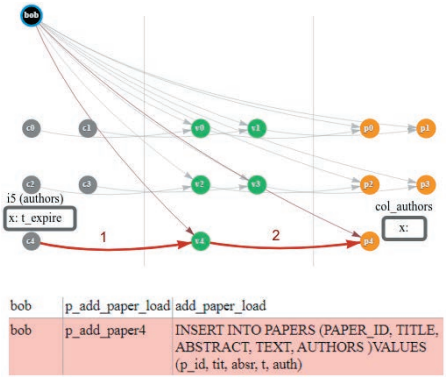


Рис. 11. Action property CompInv is violated (p_add_paper_4).

load(*id*, *f_gsp_p_s_id*(*id*))))
with *MIN* policy (WHAT declassification)
min, ...)

ШАГ 5. Нарушение свойства CompInv

После исправления модели на шаге 4 нарушений основного инварианта безопасности более не происходит. Однако включение проверки свойства перехода *CompInv* приводит к ошибке при выполнении оператора *f_is_accepted_9* (рис. 11). Стратегия исправления ошибки проста (см. Раздел 3) – обновление начальной политики соответствующей глобальной переменной:

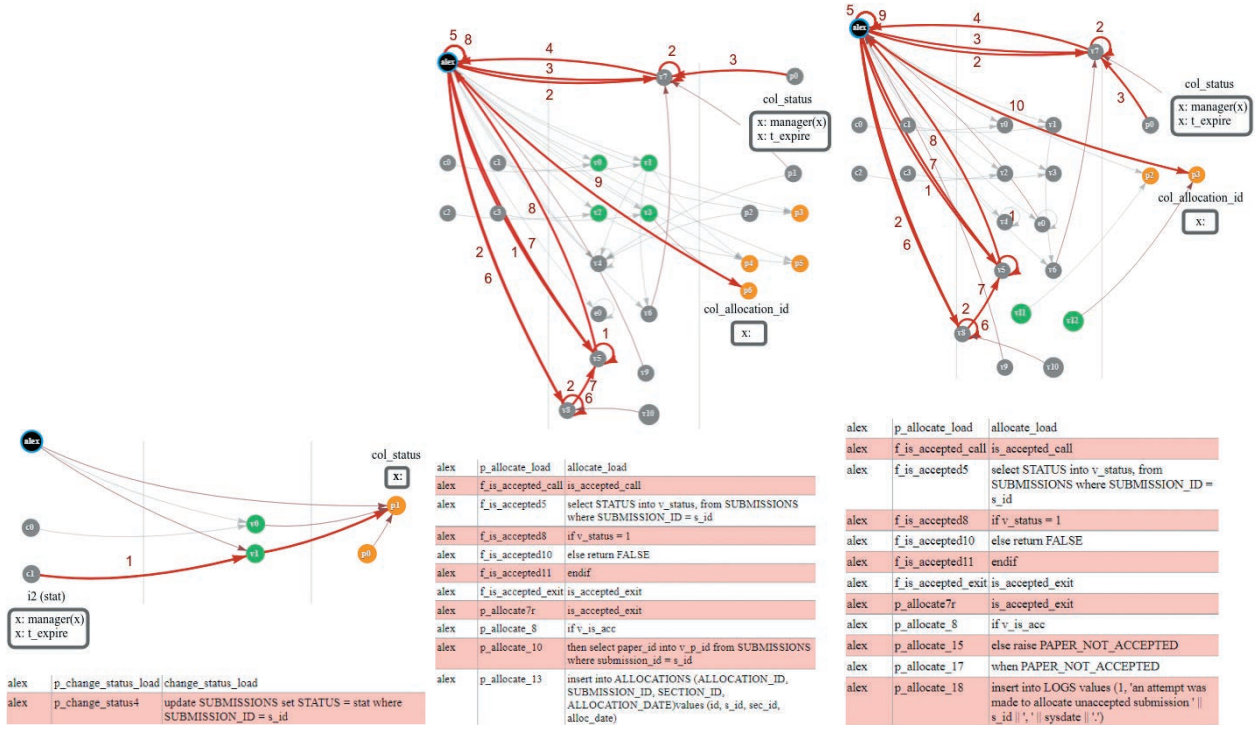
$Init \triangleq ...$
$\wedge VPol =$
[... *col_papers_authors* $\mapsto$
[*ext* $\mapsto$ 0,
policy $\mapsto$ min
{⟨*u*1, ⟨[*t_expire* $\mapsto$ {}],
[*guest* $\mapsto$ {*NONE*},
reviewer $\mapsto$ {*NONE*},
manager $\mapsto$ {*NONE*},
organizer $\mapsto$ {*NONE*}]⟩⟩},
name $\mapsto$ “`col papers authors`”], ...]

Исправление кода *PL/SQL* или глобальной политики управления доступом не требуется [4].

ШАГИ 6 – 8. Нарушение свойства CompInv

Последующие три прогона модели приводят к нарушению свойства перехода *CompInv* в операторах: *p_change_status_4*, *p_allocate_13*, *p_allocate_14* (рис. 15). Исправление модели осуществляется также как и на Шаге 5.

После исправления ошибки на восьмом шаге проигрывание модели завершается успешно.



(а) p_change_status 4 (б) p_allocate_13 (в) p_allocate_18

Рис. 12. Action property CompInv is violated.

Контроль распространения выходных значений критичных процедур и функций в прикладном программном обеспечении предполагается осуществлять с использованием стандартного анализа помеченных данных (Taint Tracking), например в среде *CodeQL* [16]. В данной работе этот этап подробно не рассматривается.

ЗАКЛЮЧЕНИЕ

В данной работе детализирована процедура выявления запрещенных информационных потоков в программных блоках *PL/SQL*. Возможно несколько избыточный иллюстративный материал, подготовленный с использованием утилиты *plifparser*, на наш взгляд имеет существенное значение для доказательства практической осуществимости предложенного в [4] подхода. В условиях некоторой потери точности анализа, за счет неизбежного для формальной верификации абстрагирования, возможность графической интерпретации проблемных трасс вычислений значительно упрощает задачу выявления “ложных” срабатываний.

Наряду с иными исследованиями логической семантики политик *Paralocks* [17], в данной работе раскрываются некоторые особенности возможной реализации языка в спецификациях *TLA+*. Существенным в этой части является наличие формальных доказательств свойств алгебраической решетки, построенной на заданном множестве политик безопасности.

В дальнейшем требуется провести отдельные исследования в отношении последнего этапа предложенной процедуры контроля информационных потоков – анализа распространения выходных значений критичных процедур и функций в прикладном программном обеспечении.

Кроме того, в работах, посвященных КИП, достаточно широко охвачена проблема *деклассификации* данных, но практически не уделяется внимание обратной процедуре – *классификации* данных. Уровень конфиденциальности используемых программой данных может не только понижаться, но и повышаться в процессе их обработки.

СПИСОК ЛИТЕРАТУРЫ

1. *Latham D.C.* Department of defense trusted computer system evaluation criteria // Department of Defense, 1986. T. 198.
2. *Infrastructure P.K., Profile T.P.* Common criteria for information technology security evaluation // National Security Agency, 2002.
3. *Девянин П.Н., Леонова М.А.* Применение подтипов и тотальных функций формального метода Event-B для описания и верификации МРОСЛ ДП-моде-

ли // Программная инженерия. 2020. Т. 11. № 4. С. 230–241.

4. *Тимаков А.А.* Контроль информационных потоков в программных блоках баз данных на основе формальной верификации // Программирование. 2022. № 4. С. 27–49
5. *Denning E. Dorothy* A lattice model of secure information flow // Communications of the ACM. 1976. № 5. P. 236–243.
6. *Шайтура С.В., Питкевич П.Н.* Методы резервирования данных для критически важных информационных систем предприятия // Российский технологический журнал. 2022. Т. 10 (1). С. 28–34.
7. *Timakov A.* PLIF. 2021. GitHub. https:// github.com/timimin/plif.
8. *Konnov I., Kukovec J., Tran T.-H.* TLA+ model checking made symbolic // Proceedings of the ACM on Programming Languages. 2019. V. 3. OOPSLA. P. 1–30.
9. *Broberg Niklas, Sands David* Paralocks: Role-based information flow control and beyond // Conference Record of the Annual ACM Symposium on Principles of Programming Languages. 2010. P. 431–444.
10. *Broberg Niklas, Sands David* Flow locks: Towards a core calculus for dynamic flow policies // Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). 2006. No March. P. 180–196.
11. *Broberg N.* Thesis for the Degree of Doctor of Engineering Practical, Flexible Programming with Information Flow Control. 2011.
12. *Hedin Daniel, Sabelfeld Andrei* A Perspective on Information-Flow Control // Software safety and securit. 2012. P. 319–347.
13. *Methni A., Lemerre M., Hedia B.B., Barkaoui K., Haddad S.* An Approach for Verifying Concurrent C Programs // 8th Junior Researcher Workshop on Real-Time Computing. 2014. P. 33–36.
14. *Fernandes A.* tlaplus-graph-explorer. 2021. GitHub. https:// github.com/afonsonf/tlaplus-graph-explorer.
15. *Hedin Daniel, Sabelfeld Andrei* A Perspective on Information-Flow Control // Software safety and security. 2012. P. 319–347.
16. *Kristensen E.* CodeQL. 2022. GitHub. https:// github.com/github/codeql.
17. *Bart V. Delfit, Broberg Niklas, Sands David* A Datalog semantics for Paralocks // Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). 2013. P. 305–320.
18. *Harrison M.A ., Ruzzo W.L., Ullman J.D.* Protection in operating systems // Communications of the ACM. 1976. V. 19. № 8. P. 461–471.