ISSN 2524-0986

iScience™

АКТУАЛЬНЫЕ НАУЧНЫЕ ИССЛЕДОВАНИЯ В СОВРЕМЕННОМ МИРЕ

ЖУРНАЛ

Выпуск 5(49)
Часть 7

Переяслав-Хмельницкий
2019

АКТУАЛЬНЫЕ НАУЧНЫЕ ИССЛЕДОВАНИЯ В СОВРЕМЕННОМ МИРЕ

ВЫПУСК 5(49)
Часть 7

Май 2019 г.

ЖУРНАЛ

Выходит – 12 раз в год (ежемесячно)
Издается с июня 2015 года

Включен в наукометрические базы:

РИНЦ http://elibrary.ru/title_about.asp?id=58411
Google Scholar

https://scholar.google.com.ua/citations?user=JP57y1kAAAAJ&hl=uk
Бібліометрика української науки
http://nbuviap.gov.ua/bpnu/index.php?page_sites=journals **Index Copernicus**
http://journals.indexcopernicus.com/++++,p24785301,3.html

Переяслав-Хмельницкий

«Актуальные научные исследования в современном мире»
Выпуск 5(49) ч. 7 ISSN 2524-0986

УДК 001.891(100) «20»
ББК 72.4
А43

Актуальные научные исследования в современном мире // Журнал - Переяслав-Хмельницкий, 2019. - Вып. 5(49), ч. 7 – 128 с.

Языки издания: українська, русский, english, polski, беларуская, казақша, o'zbek, limba română, кыргыз тили, Հայերեն

Сборник предназначен для научных работников и преподавателей высших учебных заведений. Может использоваться в учебном процессе, в том числе в процессе обучения аспирантов, подготовки магистров и бакалавров в целях углубленного рассмотрения соответствующих проблем. Все статьи сборника прошли рецензирование, сохраняют авторскую редакцию, всю ответственность за содержание несут авторы.

УДК 001.891(100) «20»
ББК 72.4
А43

СОДЕРЖАНИЕ

СЕКЦИЯ: СОВРЕМЕННЫЕ ИНФОРМАЦИОННЫЕ ТЕХНОЛОГИИ

СЕКЦИЯ: ЭКОНОМИЧЕСКИЕ НАУКИ

«Актуальные научные исследования в современном мире»
Выпуск 5(49) ч. 7 ISSN 2524-0986

УДК 004

Бурибаев Бакыт Бурибаевич, Кульмамиров Серик Алгожаевич,
Зулпыхаров Нуркен Тастанбекович
Казахский национальный университет имени аль-Фараби (Алматы, Казахстан)

ПОИСК УЯЗВИМОСТИ В ТЕКСТЕ С++ СТАТИСТИЧЕСКИМ АНАЛИЗОМ

Аннотация. *Инструмент статического анализа становится популярным средством поиска в тексте программы выявления ситуаций (ошибок стиля кодирования, нарушений об использовании библиотек или процедур языка программирования, критических ошибок, уязвимостей, закладок). Здесь представлены результаты проведенного обзора статического анализа текста программы на Си++. Описанный инструмент среди программистов особо популярна для поиска критических ошибок и уязвимостей. Рассмотренная процедура позволит оценить имеющиеся ситуаций для формирования предупреждений появления уязвимостей используемой программы.*

Ключевые слова: *статический анализ, данные, интервальный анализ, процедуры анализа, уязвимость.*

Buribaev Bakyt Buribaevich, Kulmamirov Serik Algozhaevich,
Zulpykharov Nurken Tastanbekovich
Al-Farabi Kazakh National University
(Almaty, Kazakhstan)

SEARCH FOR VULNERABILITY IN C ++ TEXT STATISTICAL ANALYSIS

Abstract: *The static analysis tool is becoming a popular means of searching in the text of a situation detection program (coding style errors, violations about the use of libraries or programming language procedures, critical errors, vulnerabilities, bookmarks). Here are the results of a review of a static analysis of the text of a C ++ program. The described tool among programmers is especially popular for searching for critical errors and*

vulnerabilities. The considered procedure will allow to assess the existing situations for the formation of warnings for the appearance of vulnerabilities of the program used.

Keywords: *static analysis, data, interval analysis, analysis procedures, vulnerability.*

Усложнение алгоритмов программы, например, в C++, влечет необходимость механизмов защиты от атак, в том числе поиска уязвимостей и ошибок, через которые могут происходить такие атаки. Одним из средств обнаружения критических ошибок является статический анализ программ - это поиск ситуаций в тексте программы, которые могут означать наличие уязвимости. Примером простых ситуаций подобного типа являются выдаваемые компилятором предупреждения, например, об использовании указателя одного типа для манипулирования объектом другого типа.

Авторами статьи проведен обзор инструмента статического анализа Svace, разработанного в Институте РАН для анализа программ на языке C++ [1-3]. Инструмент генерирует список предупреждений с указанием типа предупреждения и данных о том, как именно может произойти ситуация, описанная в предупреждении.

Такой инструмент разработан с учетом следующих требований:

- анализ производится автоматически, не требуется специально подготавливать исходный код либо вмешиваться в ходе анализа;
- анализ учитывает влияние функций C++ на ее поведение при поиске разных ситуаций;
- размер анализируемых программ может достигать миллионов строк кода и сотен тысяч функций;
- инфраструктура анализа расширяема, при разработке дополнительных алгоритмов появится возможность поиска новых видов потенциально опасных ситуаций в C++ с использованием подготовленных ими данных;
- значительная часть найденных предупреждений должна быть истинной (т.н. true positive) [7].

Для таких задач необходимостью является применение такого анализа, который не гарантирует нахождение всех ситуаций заданных типов в тексте C++ (unsound analysis). Все

известные инструменты для автоматического статического анализа, не требующего трудоемкой подготовки анализируемых программ или требований к ним (Prevent Coverity [9] и K9 Klocwork [9]), используют тот же подход.

Рассмотрим архитектуру и поведение разработанного инструмента достигнутых при анализе программ с открытым исходным кодом. Найденная ситуация в тексте программы будем называть предупреждением. Оно является истинным в случае действительного наличия описываемой данным типом предупреждения ситуации в коде и ложным в обратном случае. Детектором (checker) будем называть компонент инструмента анализа, ответственный за поиск предупреждений определенного типа (или схожих типов).

Инструмент анализа оперирует файлами с внутренним представлением (IR), сгенерированными компилятором из файлов исходного кода программы и содержащими описания всех используемых типов, глобальных и локальных переменных модуля компиляции, а также текст функций в кода для абстрактной регистровой машины.

Для построения файлов, например, с LLVM-биткодом, используется специальная утилита перехвата сборки. Она перехватывает все запуски указанного ей компилятора в ходе исполнения произвольной системы сборки анализируемой программы, разбирает командную строку запуска компилятора, формирует команду для выполнения компилятора LLVM-GCC.

Инструмент статического анализа состоит из инфраструктуры анализа, управляющей ходом анализа и реализующей набор общих алгоритмов, и набора детекторов, небольших компонент, осуществляющих поиск конкретных типов предупреждений и реализующих специальные алгоритмы анализа, необходимые для обнаружения ситуаций соответствующего вида. Помимо файлов внутреннего представления, при анализе используются спецификации некоторых стандартных библиотечных функций, входящие в состав инструмента анализа, и (в некоторых случаях) файлы с исходным кодом анализируемой программы. Результаты анализа выдаются как в текстовом виде, так и во внутреннем формате, который может быть просмотрен в среде Eclipse с помощью поставляемого с инструментом анализа встраиваемого компонента – при этом текст предупреждения автоматически показывается в окружении соответствующего исходного кода.

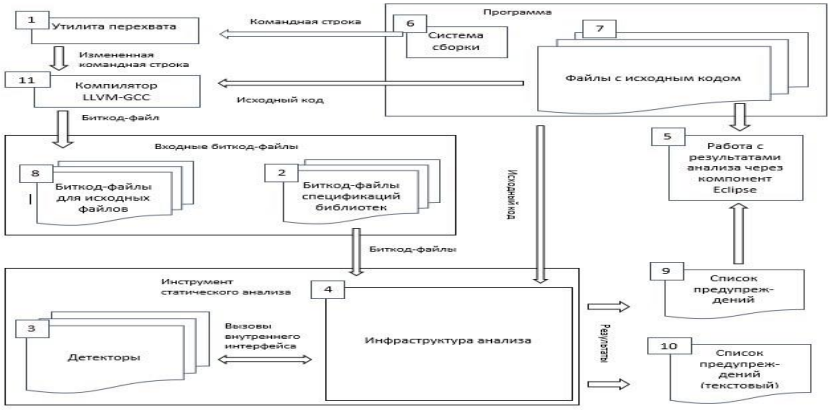

Рисунок 1. Схема работы инструмента статического анализа

Схема работы инструменты приведена на рисунке 1, где компоненты инструмента выделены цифрами 1-5, а обрабатываемые данные цифрами 610. Компилятор LLVM-GCC выделен цифрой 11, так как он является сторонним компонентом.

В начале работы утилита перехвата запуска компилятора запускает систему сборки программы обычным образом, параллельно создавая биткодфайлы. После окончания сборки над множеством сгенерированных биткодфайлов запускается инструмент анализа, который также подгружает спецификации библиотечных функций, необходимые для анализа и хранящиеся в виде биткод-файлов. Управление ходом анализа ведет инфраструктура анализа, вызывающая детекторы по мере надобности. После окончания анализа, сгенерированные предупреждения сохраняются в текстовом виде и внутреннем формате. Компонент Eclipse позволяет просматривать результаты с привязкой к исходному коду программы.

Распространяемые между функциями данные о программе хранятся в ходе анализа в виде аннотаций функций. Аннотация является структурой данных, связанной с функцией анализируемой программы, которая абстрактно описывает эффект от выполнения данной функции в произвольном контексте вызова – побочные эффекты, возвращаемые

значения, способы использования и изменения параметров и других доступных функции данных. Аннотация создается автоматически как результат анализа функции. Детекторы могут сохранять в аннотации необходимые им данные для последующего анализа и выдачи предупреждений.

Необходимые инструменту анализа спецификации стандартных библиотечных функций пишутся также на Си++ в виде заглушек, в которых указаны только полезные для анализа операции. Исходный код спецификаций транслируется компилятором LLVM-GCC, и результат поставляется в виде биткод-файлов. Эти файлы анализируются до начала анализа остальных файлов, в результате чего создаются аннотации соответствующих библиотечных функций.

Например, для функции printf достаточно указать, что ее первый аргумент всегда разыменовывается (это свойство полезно для поиска ошибок разыменования нулевого указателя; для указания этого свойства первый параметр явно разыменовывается в исходном коде спецификации) и является форматной строкой (это свойство важно для поиска уязвимостей форматной строки). Для пометки о том, что некоторый указатель есть форматная строка, используется специальная интерфейсная функция sf_use_format, предоставляемая детектором уязвимостей форматной строки. Использование этой функции перехватывается в ходе анализа. Сама аннотация на языке Си++ выглядит, как показано ниже:

```
int printf(const char *format,...) {
char d1 = *format;
sf_use_format(format);
sf_fun_printf_like(0);
sf_fun_does_not_update_vargs(1);
}
```

Статический анализ программы происходит в 4 этапа:

1 этап. Все биткод-файлы считываются по очереди в произвольном порядке, и строится общий граф вызовов программы.

2 этап. Граф вызовов обходится в обратном топологическом порядке («снизу вверх»), так что (насколько позволяет отсутствие циклов в графе вызовов) каждая функция посещается после того, как были посещены все вызываемые из

нее функции. Для каждой посещенной в этом порядке обхода функции программы выполняется внутрипроцедурный анализ.

3 этап. Выполняется специфический для Си++ анализ кода, исследующий взаимодействие методов классов.

4 этап. Принимаются решения о выдаче либо исключении некоторых предупреждений на основании собранных в ходе основного анализа статистических данных и фрагментов исходного кода, соответствующих местам потенциальной выдачи предупреждений.

В ходе анализа конкретной функции инструменту доступно внутреннее представление лишь этой функции и данные о вызываемых ей функциях, присутствующие в виде аннотаций. В результате анализа функции создается ее аннотация, которая может использоваться в дальнейшем.

Таким образом, анализ всей программы масштабируется линейно. Все аннотации функций хранятся в оперативной памяти, но при ее нехватке сериализуются на диск и считываются при необходимости. Обычного современного объема ОЗУ (до 4 ГБ) хватает для анализа программ из сотен тысяч строк кода без использования сериализации.

При анализе функции строится ее граф потока управления, после чего проводится потоково-чувствительный анализ, аналогичный анализу потока данных. При этом после нескольких проходов сверху вниз анализ завершается проходом снизу вверх. С каждой дугой графа потока управления ассоциируется контекст. Например, после выполнения строки if (x >= 4 && x <= 7) в контексте подчиненного условному выражению ребра будет отражена информация о том, что значение переменной x находится в отрезке [4 – 7].

Анализ потока данных происходит путем «продвижения» контекста по дуге, входящей в инструкцию, через эту инструкцию и построения контекста на выходе из инструкции, основываясь на том, как инструкция манипулирует данными и памятью (при этом используются компактные структуры данных, не требующие чрезмерного дублирования данных). Например, после обработки следующих инструкций контекст выглядит: s = 6; *p = s.

Ячейка памяти	p	*p	s
Идентификатор значения	vid1	vid1	vid2
Атрибуты	PT-TO: *p NOTNULL	POSITIVE	POSITIVE

Элементами этого контекста являются три ячейки памяти (p, s и *p, при этом p и s соответствуют явно указанным переменным), два идентификатора значений (один соответствует значению 6, другой соответствует адресу ячейки *p), и 3 атрибута. Ячейка p хранит значение с идентификатором vid1, адресом ячейки *p. Это отражено в атрибутах PT-TO (куда указывает p) и NOT-NULL (если бы указатель был нулевым, то присваивание *p=s привело бы к ошибке времени выполнения). Ячейки *p и s хранят одно и то же значение vid2 (этот факт устанавливается после обработки того же присваивания), которое имеет атрибут —положительный‖. Например, детектор уязвимости переполнения буфера может отметить в атрибуте значения vid2, что оно равно в точности 6.

При анализе функции не используется информация о том, откуда она могла быть вызвана. Такой вид процедурного анализа называется контекстнонечувствительным, так как разные точки вызова функции не различаются, и возможно появление информации о данных на —ложных‖ путях выполнения, которые входят в функцию в одной точке вызова и выходят в другой. Некоторая контекстная чувствительность оказывается возможной при использовании параметризованных аннотаций [17].

На рисунке 2 аннотация функции foo отражает тот факт, что ее возвращаемое значение равно аргументу, через использование одного идентификатора значения для обеих ячеек памяти, при этом идентификатор значения не привязан к конкретным атрибутам:

```
int foo(int p) {
return p;
}
@
r
e
t
:
=
p
i
n
```

t
q
=
r
e
a
d
(
…
)
;
i
n
t
s
=
f
o
o
(
q
)
;
/
/
s
п
о
л
у
ч
е
н
о
// от пользователя

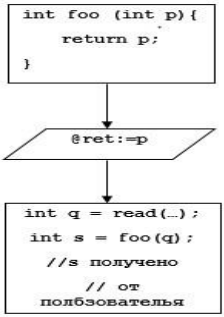

Рисунок 2. Параметризованная аннотация

При продвижении контекста через вызов функции foo в примере используется фактический параметр q, значение которого получено от ввода пользователя. В ходе продвижения идентификатор значения для ячейки s устанавливается в идентификатор значения для ячейки q, тем самым значение s будет также помечено атрибутом —получено от пользователя‖. При этом анализ вызова функции foo не влияет на ее аннотацию, и таким образом, информация об испорченности возвращаемого значения в данном вызове не будет влиять на анализ других вызовов той же функции. Такой анализ одновременно контекстно-чувствителен и позволяет распространять атрибуты через вызовы функций [9].

Процесс продвижения контекста через точку вызова функции называется трансляцией аннотаций, т.к. при этом элементы аннотации ставятся в соответствие элементам контекста в точке вызова.

На примере, показанном на рисунке 3, аннотация функции foo содержит ячейки z и *z, у ячейки z есть идентификатор значения v1, имеющий атрибут, говорящий о том, что ячейка указывает на *z, а у ячейки *z есть идентификатор v2, имеющий атрибут, показывающий, что значение идентификатора равно четырем. В общем случае ячейки, идентификаторы и атрибуты составляют лес, в котором происходит поиск соответствующих элементов контекста в точке вызова и их создание в случае неудачи поиска. Контекст на ребре, входящем в инструкцию вызова, содержит лишь ячейку x, которая связывается с z.

После того, как все идентификаторы созданы, атрибуты идентификаторов *x и *z должны быть объединены (точнее,

атрибуты идентификатора v2 должны быть перенесены с объединением на атрибуты идентификатора значения соответствующей ячейки *x). Так как ранее ячейки *x не было в контексте, то информация из аннотации просто копируется в виде атрибута, показывающего, что значение *x равно четырем:

```
void test() {
 foo(x);
}
void foo(int* z) {
 *z = 4;
}
z
(
=
v
1
)
*
z
(
=
v
2
)
4
x
*
x
```

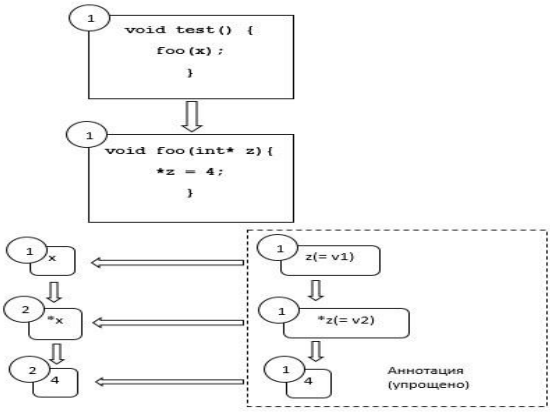

Рисунок 3. Трансляция аннотаций

Каждый детектор может вводить новые атрибуты для отслеживания необходимых свойств данных программы и выдачи предупреждений. Чтобы определить способы продвижения и слияния новых атрибутов, детекторы описывают обработчики этих событий, которые вызываются инфраструктурой анализа, и специальные функции манипуляции атрибутами, которые используются при написании спецификаций библиотечных функций. Обработчики событий имеют доступ к инструкции внутреннего представления, через которую продвигается контекст, и всем элементам контекста, на которые ссылаются используемые в инструкции переменные и константы: charbu f[10];

```
buf[i]=0;  //  i  ∈  [0,9] →
OVERFLOW  if(i>9){
// i[ 10,+INF); i∈ [0,9] → OVERFLOW
/->/ OVERFLOW
exit(1);
}
```

Выше в тексте программы описаны шаги анализа, используемые для нахождения ситуации переполнения буфера. В ходе анализа приведенного фрагмента кода вводятся 2 атрибута. Первый атрибут описывает отрезок, внутри которого должно лежать значение, чтобы в данной точке программы не

было переполнения буфера. Второй атрибут описывает отрезок, в котором гарантированно находится данное значение. Во второй строке рассматриваемого примера первый атрибут устанавливается в отрезок [0,9]. Атрибут, связанный с идентификатором значения, распространяется до вызова функции exit. В точке вызова этой функции отрезок значения для этого идентификатора будет установлен в [10,+INF) как результат продвижения через условное выражение. Так как второй атрибут не содержит значений, находящихся в первом, детектор выдает предупреждение о переполнении буфера.

В редких случаях детекторы могут обращаться к файлам с исходным кодом программы для отсечения ложных предупреждений с помощью информации, которая не сохранена в файлах с внутренним представлением. Например, некоторые предупреждения могут выдаваться в тексте макросов в местах, которые являются мертвым кодом. Знание о том, сгенерирован ли некоторый код с помощью макроса или написан программистом вручную, позволяет отсеять такие случаи.

После окончания анализа его результаты могут быть просмотрены пользователем через компонент в среде Eclipse. Для каждого предупреждения показывается сообщение и релевантные места в исходном коде программы. Для удобства пользователя реализованы следующие дополнительные возможности:

- возможность запуска анализа некоторого проекта через среду Eclipse;
- возможность размечать результаты анализа как истинные или ложные;
- поддержка истории запусков анализа.

Эта часть инструмента позволяет сохранять результаты анализа и соответствующие исходные коды в базу данных и просматривать по мере необходимости. История запусков в совокупности с разметкой истинных и ложных предупреждений позволяет не выдавать пользователю повторно предупреждения, которые им были помечены как ложные.

Ниже в тексте программы C++ приведен тест для предупреждения DEREF_OF_NULL (разыменование указателя, имеющего нулевое значение). Этот тест проверяет работу процедурного анализа с побочными эффектами, выражающимися в изменении значений глобальных переменных. Функция get_var всегда возвращает значение

глобальной переменной var. В функции test первый условный оператор выполняется, когда функция get_var возвращает (а переменная var имеет) нулевое значение, и предупреждение DEREF_OF_NULL должно быть выдано при разыменовании этого значения.

Требование о выдаче выражается вызовом функции-утверждения sf_assert_thrown, которой передается имя предупреждения. Выдача ожидается на следующей после вызова функции строке. Аналогично, второй условный оператор выполняется при ненулевом возвращаемом значении, и предупреждение не должно быть выдано, что выражается вызовом функции sf_assert_not_thrown:

```
#inclu
de
"spec
func.
h"
int*
var;
int*
get_v
ar(voi
d)    {
return
var;
}
void test(int *p, int x, int y)
{
 if(!get_var()) {
 sf_assert_thrown(DEREF_OF_NULL);
 x = *get_var();
 }
 if(get_var()) {
 sf_assert_not_thrown(DEREF_OF_NULL);
 x = *get_var();
 }
}
```

Таким образом, инструменты статического анализа являются полезными для задачи обеспечения

безопасности программного обеспечения. С помощью глубокого процедурного анализа можно достичь хорошей доли (до 80%) истинных предупреждений при приемлемом времени анализа (несколько часов для миллионов строк кода).

Залогом успеха является кропотливая работа по просмотру реальных промышленных программ с открытым исходным кодом и доработке инструмента анализа на основе его результатов на этих программах, для достижения приемлемого компромисса между качеством выдаваемых результатов и их количеством.

В планах работы над инструментом – поддержка других языков программирования (в первую очередь C#), языков Web-программирования (Javascript), развитие интерфейса пользователя (поддержка удаленного и распределенного анализа, поддержка командной работы), а также предоставление доступа к инструменту как к сервису в рамках систем облачных вычислений.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ:

1. С. С. Гайсарян, А. В. Чернов, А. А. Белеванцев, О. Р. Маликов, Д. М. Мельник, А. В. Меньшикова. О некоторых задачах анализа и трансформации программ. Труды ИСП РАН, №5, с. 7-41, 2004.
2. О. Р. Маликов, А. А. Белеванцев. Автоматическое обнаружение уязвимостей в программах. Материалы конференции «Технологии Майкрософт в теории и практике программирования», Москва, 2004.
3. О. Р. Маликов. Автоматическое обнаружение уязвимостей в исходном коде программ. Известия ТРТУ, №4, с. 48-53, 2005.
4. В. С. Несов, О. Р. Маликов. Использование информации о линейных зависимостях для обнаружения уязвимостей в коде программ. Труды ИСП РАН, № 9, с. 51-57, 2006.
5. О. Р. Маликов, В. С. Несов. Автоматический поиск уязвимостей в больших программах. Известия ТРТУ, Выпуск «Информационная безопасность», №7, с. 114-120, 2006.
6. В. С. Несов. Использование побочных эффектов функций для ускорения автоматического поиска уязвимостей в программах. Известия ЮФУ. Технические науки. Выпуск «Информационная безопасность». Таганрог: Изд-во ТТИ ЮФУ, 2007. № 1(76), с. 134-139.

7. В. С. Несов, С. С. Гайсарян. Автоматическое обнаружение дефектов в исходном коде программ. Методы и технические средства обеспечения ИБ: Материалы XVII Общероссийской научно-технической конференции. СПб.: Изд-во Политехн. ун-та, 2008, с.107.
8. В. С. Несов. Автоматическое обнаружение дефектов при помощи межпроцедурного статического анализа. Материалы XI Международной конференции «РусКрипто'2009».
9. Vladimir Nesov. Automatically Finding Bugs in Open Source Programs. Electronic Communications of the EASST 20. ISSN 1863-2122, 2009.

АКТУАЛЬНЫЕ НАУЧНЫЕ ИССЛЕДОВАНИЯ В СОВРЕМЕННОМ МИРЕ

Май 2019 г.
ВЫПУСК 5(49)
Часть 7

Ответственность за новизну и достоверность результатов научного исследования несут авторы
Ответственный за выпуск: Водяной О.
Дизайн и верстка: Вовкодав А.
Учредитель: ОО "Институт социальной трансформации"
свидетельство о государственной регистрации №1453789 от 17.02.2016 г.

Подписано к печати
5.06.2019. Формат 60x84
1/16.
Тираж 300 шт. Заказ
№042 Изготовитель: ФЛП
"Кравченко Я.О."
свидетельство о государственной регистрации В01 №560015
Адрес: 03039, Украина, Киев, просп. В. Лобановского, 119 тел. +38 (044) 561-95-31

Адрес ред. коллегии:
08400, Украина, Киевская обл., г. Переяслав-Хмельницкий, ул. Богдана Хмельницкого, 18 тел.: +38 (063) 5881858 сайт: http://iscience.in.ua e-mail: iscience.in.ua@gmail.com