УДК 004.056

М.О. Шудрак, В.В. Золотарев

МОДЕЛЬ, АЛГОРИТМЫ И ПРОГРАММНЫЙ КОМПЛЕКС АВТОМАТИЗИРОВАННОГО ПОИСКА УЯЗВИМОСТЕЙ В ИСПОЛНЯЕМОМ КОДЕ

Шудрак М.О., Золотарев В.В. **Модель, алгоритмы и программный комплекс автоматизированного поиска уязвимостей в исполняемом коде.**

Аннотация. В настоящей работе рассматривается проблема автоматизированного поиска уязвимостей в исполняемом коде. В работе проводится анализ проблематики, и выделяются недостатки существующих решений, в части отсутствия возможности обнаружения уязвимостей с учётом тех угроз, которые они несут для защищаемой информации, которая обрабатывается в ПО. Для решения этой проблемы предлагается оригинальная модель автоматизированного поиска уязвимостей в трассе программы, её алгоритмическое обеспечение и программная реализация. В рамках модели приводятся формальные критерии отнесения ошибки к уязвимости с учётом распределения защищаемой информации в памяти программы. Для выделения участков памяти с защищаемой информацией в работе используется методика анализа помеченных данных. Кроме того приводится экспериментальная оценка эффективности разработанного программного комплекса, которая показала, что разработанное решение позволяет детектировать на 5 типов уязвимостей больше в ОС Windows и на 4 типа уязвимостей больше в Linux по сравнению с существующими аналогами. Все модули разработанного комплекса были опубликованы как ПО с открытым исходным кодом, могут свободно использоваться в других проектах и доступны для скачивания в Интернете.

Ключевые слова: уязвимость, исполняемый код, динамическое тестирование, ошибка, критерий.

Shudrak M.O., Zolotarev V.V. **A Model, Algorithms and Software Tool for Vulnerabilities Detection in Machine Code.**

Abstract. In the article we consider the problem of vulnerabilities detection in machine code. In this paper, disadvantages of current solutions in case of possibility to detect vulnerabilities in view of threats to confidential information that is processed in vulnerable software are highlighted. To solve this problem, we propose original model of vulnerabilities detection in program trace, its algorithmic support and software implementation. The model provides formal criteria to distinct bug from vulnerability taking into account distribution of protected information in the memory of software under test. We use tainted data analysis technique to highlight such memory regions. In addition, we conduct experimental evaluation of developed system efficiency which demonstrates that our solution allows detecting 5 types of Windows software vulnerabilities more and 4 types Linux software vulnerabilities more than existing analogs.

Keywords: vulnerability; machine code; dynamic analysis; bug; criteria.

1. Введение. Процесс разработки программного обеспечения невозможен без тех или иных ошибок, допускаемых программистами или архитекторами в ходе своей работы. В свою очередь такие ошибки могут приводить к уязвимостям в программном обеспечении, которые могут использоваться злоумышленниками для компрометации конфиденциальной информации обрабатываемой уязвимым ПО. Ошибки в

программном обеспечении (ПО) в свою очередь являются следствием развития современных информационных технологий в части увеличения объема, сложности и требований к масштабируемости программного кода.

В данной работе вопрос анализа безопасности ПО будет рассматриваться в условии отсутствия исходного текста программы. Актуальность анализа программ без исходного кода во многом обусловлена высоким уровнем распространения проприетарного ПО, исходный код которого представляет собой интеллектуальную собственность той или иной компании, которые, как правило, не заинтересованы в публикации или передачи своих разработок третьим лицам. Также даже при наличии исходного кода существует проблема того, что преобразования выполняемые компилятором и различными оптимизаторами могут значительно изменить поведение приложения, сделав результаты аудита исходного кода недостоверными. В зарубежной литературе эта проблема носит название «What You See Is Not What You eXecute» [1].

Структурно данная работа будет построена следующим образом: во втором разделе будет рассмотрена проблема поиска уязвимостей в исполняемом коде. Будут выделены критерии отнесения ошибки к уязвимости, рассмотрена нормативно-методическая база, а также выделены недостатки существующих методик и программных систем. В третьем разделе будет описана новая модель автоматизированного поиска уязвимостей в трассе программы, будут формально определены критерии возникновения ошибки в исполняемом коде и их отнесения у уязвимости с учётом распределения защищаемой информации в памяти программы. Кроме того будет приведено алгоритмическое обеспечение модели (раздел 4), а также программная реализация этих алгоритмов (раздел 5). В 6 разделе будут приведены результаты сравнительной оценки эффективности разработанного комплекса с аналогами, а в заключительном разделе будут сделаны выводы и дальнейшие перспективы работы.

2. Описание проблемы. Согласно ГОСТам и другим нормативно – правовым документам РФ отдельного понятия для *программной ошибки*, *программного дефекта* или *уязвимости* в *программном коде* не существует, однако в [2] *дефект* определяется как каждое отдельное несоответствие продукции установленным к ней требованиям. В [3] автор подчеркивает, что дефект является более общим понятием и объединяет в себе понятия ошибки и уязвимости. Таким образом, под *уязвимостью в ПО* будем понимать дефект безопасности ПО, обуславливающий возможность реализации угроз безопасности обрабатываемой в ПО защищаемой информации.

Выше было отмечено, что причиной уязвимости является ошибка, приводящая к нарушению безопасности обрабатываемой в ПО информации. Опишем в таблице 1 все возможные типы ошибок, приводящие к нарушению безопасности защищаемой информации. Отметим, что там, где в классе угроз находится нарушение безопасности, следует понимать, что тип ошибок потенциально может приводить к нарушению целостности, доступности или конфиденциальности защищаемой информации.

Таблица 1. Описание классов угроз нарушения безопасности в зависимости от типа ошибки и критерии отнесения ошибки к уязвимости

Тип ошибки	Класс угроз	Критерии отнесения ошибки к уязвимости
Переполнение кучи	Нарушение целостности, доступности или конфиденциальности обрабатываемой в ПО защищаемой информации	Есть возможность перезаписать другой фрагмент кучи, содержащий защищаемую информацию или перезаписи информации, управляющей потоком управления программы. Есть возможность вызова необработанного исключения.
Переполнение стека		Есть возможность перезаписать другой фрагмент стека, содержащий защищаемую информацию или перезаписи информации, управляющей потоком управления программы. Есть возможность вызова необработанного исключения.
Ошибки форматирования строк		Есть возможность перезаписать другой фрагмент стека, содержащий защищаемую информацию или перезаписи информации, управляющей потоком управления программы. Есть возможность вызова необработанного исключения.
Использование неинициализированной памяти	Нарушение доступности обрабатываемой информации	Есть возможность вызова необработанного исключения в тестируемой программе
Использование освобожденной памяти	Нарушение конфиденциальности или доступности обрабатываемой в ПО защищаемой информации	Есть возможность вызова необработанного исключения или чтения защищаемой информации, которая ранее была освобождена
Обращение по недопустимому адресу	Нарушение доступности обрабатываемой в ПО защищаемой информации	Есть возможность вызова необработанного исключения в тестируемой программе

Отдельно необходимо остановиться на понятии защищаемой информации. Согласно [4] к защищаемой информации относится информация подлежащая защите в соответствии с требованиями правовых документов или требованиями, устанавливаемые собственником этой информации. В рамках тестирования часто нет информации о том, какие конкретно защищаемые данные будут обрабатываться в программе, однако для выполнения поиска уязвимостей достаточно иметь адреса памяти, в которых эта информация будет храниться. В этом случае, до начала тестирования достаточно сформировать и пометить тестовые защищаемые данные (которые будут обрабатываться в ПО, как защищаемая информация), а затем проанализировать, то, как эти данные будут обрабатываться, и сохранить информацию о том, какие участки памяти участвовали в этом процессе. Подробный алгоритм поиска участков памяти с защищаемой информацией будет приведен далее в работе. Кроме того к защищаемым участкам памяти могут относиться те участки памяти, которые отвечают за управление программой и её бесперебойное функционирование.

На сегодняшний день для поиска уязвимостей в условиях отсутствия исходного кода используется динамический и статический анализ бинарного кода. Так в рамках автоматизированного статического анализа могут использоваться методики поиска потенциальных уязвимостей по шаблонам [5] или символическое исполнение программы [6]. Однако фундаментальные проблемы, связанные с декомпиляцией и анализом бинарного кода не позволяют полностью восстановить высокоуровневый аналог скомпилированного модуля, что создает серьезные трудности для статического анализа, а рост числа потенциальных путей исполнения в рамках символьного исполнения ведёт к проблеме «экспоненциального взрыва» [6].

Для решения этих проблем применяются методики динамического анализа, отчасти решающие проблемы статического анализа, однако не решающие вопрос с покрытием кода при тестировании, а также порождающие проблемы, связанные с влиянием анализатора на тестируемое приложение. При этом для некоторых инструментов тестирования может быть характерен высокий уровень потребления вычислительных ресурсов [7]. Также важно отметить, что многие методики динамического анализа характеризуется направленностью на обнаружение уязвимостей, связанных с нарушением доступности обрабатываемой информации, которые не позволяют детектировать некоторые типы уязвимости, не приводящие к вызову исключения в тестируемом приложении. Примером может служить ситуация, продемонстрированная в листинге 1.

```
1    int main(){
2        char buffer[10];
3        char overflow_string[9];
4        int a = 0;
5        fgets(buffer, sizeof buffer, stdin);
6        strcpy(overflow_string, buffer);
7        if (a == 0)
8                printf("expected behavior");
9        else
10               printf("overflow occurred");
11       return 0;
12   }
```

Листинг 1. Пример переполнения 1 байта в стеке

Согласно листингу 1, в случае передачи этому приложению строки длиной в 10 байт, произойдёт переполнение (строка 6), так как принимающий буфер на 1 байт меньше, чем исходящий буфер. Произойдёт переполнение одного байта, что может привести к перезаписи следующих за ним данных. Такая ситуация может и не привести к отказу в обслуживании, и приложение продолжит свою работу штатно. Например, в вышеизложенном примере может произойти вывод строки «overflow occurred» в зависимости от того, как будут располагаться данные в стеке.

Отметим, что для отслеживания ситуаций таких, как в вышеизложенном примере, необходимо отслеживать ошибочные обращения к памяти в тестируемом приложении. Для решения этой проблемы существует методика, в рамках которой осуществляется поиск уязвимостей в программе путем учета взаимодействия программы с памятью в ходе обработки тестовых данных (таким образом, генерация тестовых данных является задачей эксперта или может выполняться автоматически с помощью фаззера) [8, 9]. Для реализации этого подхода учитывается каждый байт памяти тестируемого приложения, который может иметь три состояния:

– недоступная память. Память, которая недоступна приложению;

– неинициализированная память. Память, которая адресуема, но доступ к которой не должен быть доступен приложению до тех пор, пока эта память не будет выделена;

– инициализированная память. Память, которая адресуема и доступна на запись или чтение приложению.

Информация о состоянии памяти сохраняется в так называемой «*теневой памяти*» - специальная область в памяти, которая описывает совокупность состояний памяти в тестируемом приложении. Тестирование программы заключается в проверке корректности работы программы с каждым байтом памяти и сигнализации об ошибках в случае

некорректного обращения к ней. На сегодняшний день существует только два инструмента, реализующих эту методику без доступа к исходному коду. Это системы DrMemory [10] в основе, которой лежит среда динамической бинарной инструментации (ДБИ) DynamoRIO [8] и Memcheck в основе которой лежит среда ДБИ Valgrind [9].

Крайне важно отметить, что данные системы направлены преимущественно на обнаружение ошибок в программном обеспечении и не проводят оценку тех угроз, которые они несут для защищаемой информации, обрабатываемой в программе (отметим, что данное утверждение будет подтверждено экспериментальными данными далее в работе). Эти недостатки порождают необходимость поиска новых подходов и решений в этой области.

3. Модель автоматизированного поиска уязвимостей. Рассмотрим работу программы как систему с конечным множеством состояний, в которые она последовательно переходит в ходе своей работы. Будем выполнять поиск уязвимостей в трассе по завершению работы программы. Под трассой программы следует понимать упорядоченную последовательность выполненных инструкций, значений регистров и состояний памяти после выполнения каждой инструкции.

Представим состояние регистров программы после выполнения каждой инструкции в трассе в виде вектора $s = (eax, ebx, ecx, edx, esi, edi, ebp, esp, eip)$, описывающего значения регистров процессора, где *eax, ebx, ecx, edx, esi, edi, ebp, esp, eip* – рассматриваемые в работе регистры процессора. Представим трассу программы в виде системы упорядоченных векторов: $s_0, s_1, \dots, s_n$, где n – количество выполненных инструкций, s_0 – начальное состояние, s_n – заключительное состояние.

Для поиска уязвимостей определим L как множество всех адресов в памяти доступных программе для обращения. Также определим $P_i = f(s_i)$ как множество фактических указателей в память из состояния s_i и $P_i \subseteq L, i \in [0; n]$. Отметим, что функция f будет задана алгоритмически в разделе 3.

В свою очередь пусть имеется M_i - множество доступных адресов в памяти из состояния s_i и $M_i \subseteq L$. Алгоритм определения этого множества будет описан позднее в разделе 4.

Пусть $Lprot_i \subseteq L$ некоторое подмножество фрагментов памяти, хранящих защищаемую информацию в рамках состояния s_i. Алгоритм автоматизированного поиска фрагментов памяти, хранящих защищаемую информацию, будет приведен позднее в разделе 4.1.

Определив $Lprot_i$, M_i и P_i, можно записать вектор состояния s_i следующим образом:

$$s_i = (eax_i, ebx_i, ecx_i, edx_i, esi_i, edi_i, ebp_i, esp_i, eip_i, P_i, M_i, Lprot_i)$$

Таким образом, трасса программы также может быть представлена с помощью матрицы состояний в следующем виде:

$$SM = \begin{bmatrix} eax_0 & ebx_0 & ecx_0 & edx_0 & edi_0 & esi_0 & esp_0 & ebp_0 & eip_0 & P_0 & M_0 & Lprot_0 \\ eax_1 & ebx_1 & ecx_1 & edx_1 & edi_1 & esi_1 & esp_1 & ebp_1 & eip_1 & P_1 & M_1 & Lprot_1 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ eax_i & ebx_i & ecx_i & edx_i & edi_i & esi_i & esp_i & ebp_i & eip_i & P_i & M_i & Lprot_i \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ eax_n & ebx_n & ecx_n & edx_n & edi_n & esi_n & esp_n & ebp_n & eip_n & P_n & M_n & Lprot_n \end{bmatrix}$$

Обозначим через γ функцию, которая позволяет определить, является ли состояние s_i допустимым или не допустимым следующим образом:

$$\gamma(s_i, L) = \begin{cases} 2, \text{если } P_i \cap (Lprot_i \backslash M_i) \neq \emptyset \\ 1, \text{если } ((P_i \cap Lprot_i = \emptyset) \wedge (\mathrm{P_i} \cap (\mathrm{L} \backslash \mathrm{M_i}) \neq \emptyset)) \vee (P_i \not\subseteq L) \\ 0, \text{во всех других случаях} \end{cases} \quad (1)$$

2 – недопустимое состояние и есть уязвимость;

1 – недопустимое состояние и есть ошибка;

0 – допустимое состояние.

Рассмотрим детально каждый критерий из выражения 1 в таблице 2.

Таблица 2. Описание каждого критерия из выражения 1.

Условие	Описание критерия	Тип ошибки/уязвимости
$P_i \cap (Lprot_i \backslash M_i) \neq \emptyset$	Как минимум один из элементов множества фактических указателей принадлежит множеству защищаемых участков памяти и не принадлежит множеству адресов памяти доступных из состояния s_i	Переполнение буфера, переполнение кучи, использование ранее освобожденной памяти, ошибки форматирования строк
$(P_i \cap Lprot_i = \emptyset) \wedge (\mathrm{P_i} \cap (\mathrm{L} \backslash \mathrm{M_i}) \neq \emptyset$	Ни один из элементов множества фактических указателей не принадлежит множеству защищаемых участков памяти и как минимум один из элементов множества фактических указателей принадлежит множеству выделенных участков памяти и этот участок не доступен из $\mathrm{s_i}$	Переполнение буфера, переполнение кучи, использование ранее освобожденной памяти, использование неинициализированной памяти, ошибки форматирования строк
$P_i \not\subseteq L$	Как минимум один из элементов множества фактических указателей не принадлежит доступному для обращения множеству адресов памяти	Обращение по недопустимому адресу

4. Алгоритм автоматизированного поиска уязвимостей в трассе. Рассмотрим алгоритм автоматизированного поиска уязвимостей в трассе на рисунке 1.

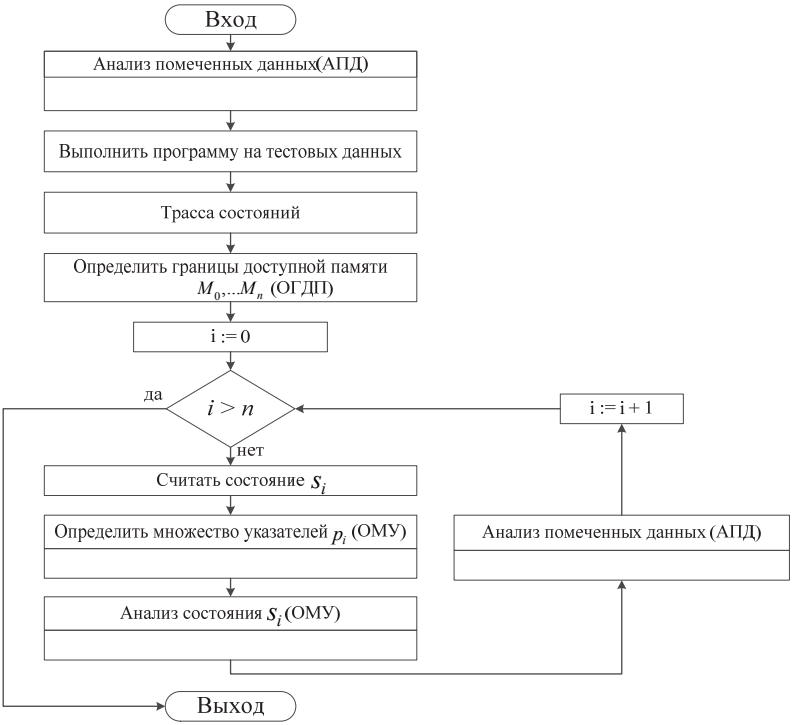

Рис. 1. Алгоритм автоматизированного поиска уязвимостей в трассе

На первом этапе алгоритма производится определение участков памяти, которые задействованы в обработке защищаемой информации (алгоритм определения таких участков будет приведен далее в работе). Затем программа выполняется на тестовых данных, в результате чего имеем трассу размером n инструкций.

На следующем этапе вступает в работу алгоритм определения границ доступной памяти (ОГДП). Рассмотрим его детально на рисунке 2. В качестве параметра передается итератор i равный нулю. На первом шаге производится вход в цикл считывания инструкций и выполняется дизассемблирование инструкции по адресу eip_i. Затем, если выполняется выделение или освобождение памяти, определяется множество адресов, которые затем добавляются или исключаются из M_{i+1}. Затем, если размер выделенного участка стека увеличивается или

уменьшается, то в множество M_{i+1}, добавляется или исключается множество новых доступных/не доступных адресов. В том случае если инструкция является вызовом процедуры, то производится определение множества передаваемых адресов памяти в качестве аргументов процедуры, а затем производится рекурсивный вызов алгоритма ОГДП с параметром $i = i+1$.

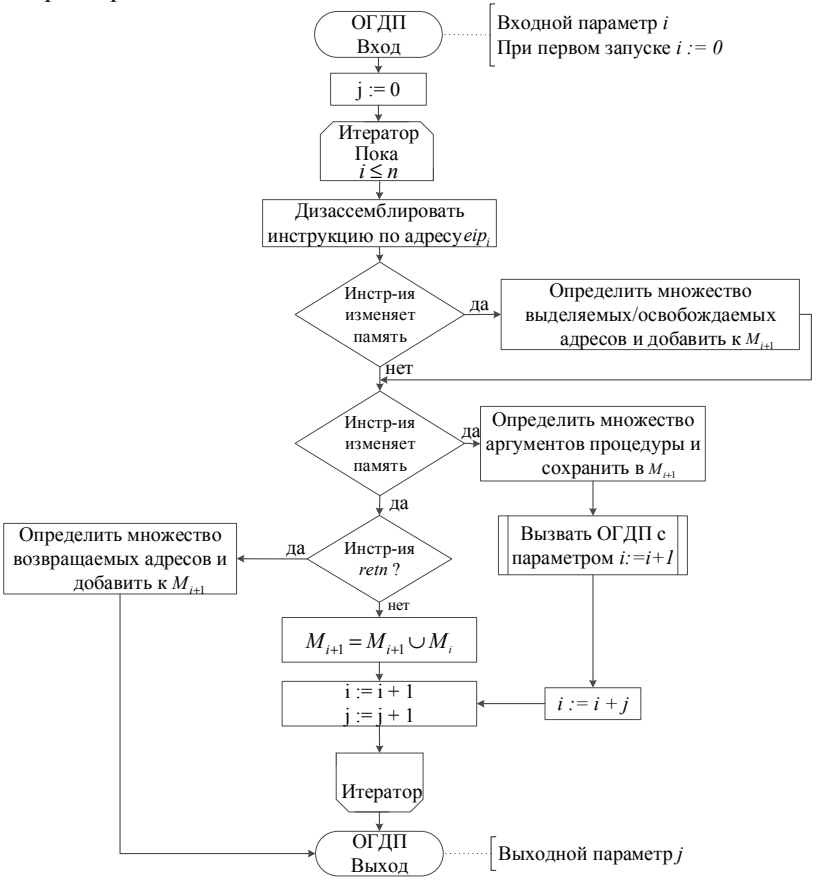

Рис. 2. Алгоритм определения границ доступной памяти (ОГДП)

Таким образом, описанный на рисунке 2 алгоритм является рекурсивным. Возврат из рекурсии происходит при достижении инструкции retn, для которой определяется множество возвращаемых адресов памяти, которые сохраняются в M_{i+1}. Переменная j определяет

количество проанализированных инструкций в рекурсии и позволяет выполнить переход к последующему после вызова call состоянию.

На следующем этапе согласно алгоритму выполняется последовательное считывание состояний s_i. Для каждого состояния последовательно выполняются два алгоритма: определение множества указателей P_i (ОМУ) и анализ состояния s_i (АС) на основе M_i и P_i с помощью выражения 1.

Для описания алгоритма определения множества фактических указателей рассмотрим блок – схему на рисунке 3.

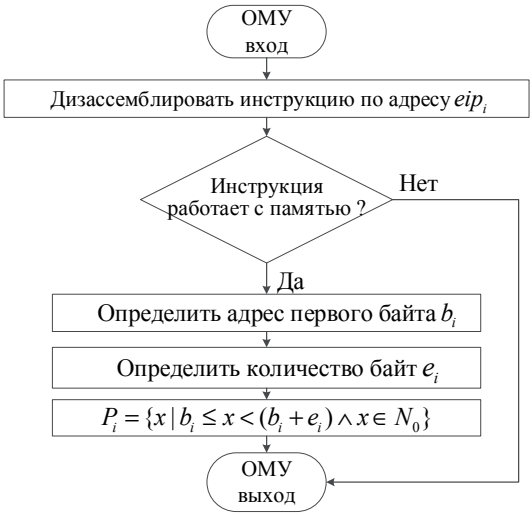

Рис. 3. Алгоритм определения множества указателей P_i (ОМУ)

На первом шаге алгоритма ОМУ выполняется считывание и дизассемблирование инструкции по адресу eip_i. Затем в том случае, если инструкция работает с памятью выполняется определение адреса в памяти первого байта, к которому осуществляется доступ (обозначим его как b_i), а затем определяется количество считываемых или записываемых байт (обозначим это количество через e_i). Тогда элементы множества P_i будут определены следующим образом: $P_i \in \{x \mid b_i \le x < (b_i + e_i) \wedge x \in N_0\}$.

Получив множества P_i и M_i, алгоритм переходит к анализу состояния (АС). Рассмотрим этот алгоритм на рисунке 4. На вход алгоритму поступают множества M_i и P_i. Затем последовательно выполняется проверка принадлежности множества фактических указателей в соответствии с критериями из выражения 1. В том случае, если один из критериев определяет факт наличия ошибки или уязвимости, инфор-

мация об этом сохраняется, и анализ продолжается для следующего состояния до тех пор, пока не будут считаны все состояния в трассе. Отметим, что после определения допустимости состояния s_i выполняется анализ помеченных данных и расчёт $Lprot_{i+1}$ (при $i = n$ не выполняется) для того, чтобы учесть возможные изменения во множестве защищаемых участков памяти при переходе в следующее состояние.

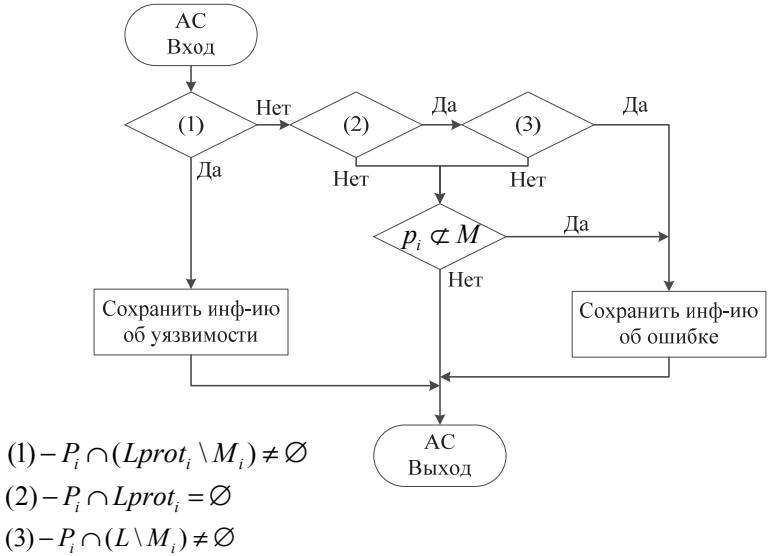

Рис. 4. Алгоритм анализа состояния s_i (АС)

4.1. Алгоритм анализа помеченных данных. Разработав модель и алгоритм автоматизированного поиска уязвимостей в трассе программы, рассмотрим алгоритм анализа помеченных данных. Согласно модели для выполнения поиска уязвимостей необходимо знать адреса памяти, в которых будет храниться защищаемая информация. Учитывая недостатки методов статического анализа, было принято решение выполнять анализ памяти в процессе исполнения программы. Для этого на подготовительном этапе тестирования необходимо подать в программу тестовые защищаемые данные, которые могут быть сформированы как разработчиком будущего программного продукта, так и владельцем защищаемой информации. В ходе обработки этих данных вступает в работу алгоритм анализа помеченных данных. Ключевым моментом для понимания работы алгоритма является понятия помеченных и не помеченных данных, а также помеченных и не помеченных участков памяти.

Под помеченными данными понимаются защищаемые данные, а под непомеченными соответственно любые другие данные используемые в программе. Рассмотрим алгоритм анализа помеченных и не помеченных данных на рисунке 5.

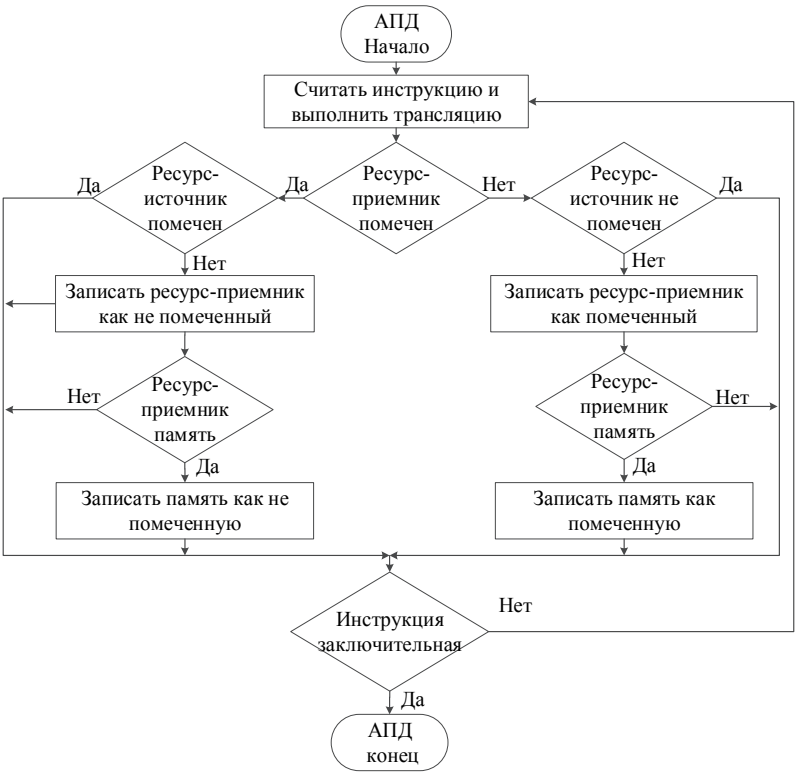

Рис. 5. Алгоритм анализа помеченных данных

В алгоритме реализуются два ключевых правила анализа продвижения помеченных данных в программе:

1) Если помеченные данные записываются или считываются в не помеченный ресурс (регистр или ячейка памяти), то этот ресурс становится помеченным.

2) Если не помеченные данные записываются или считываются в помеченный ресурс, этот ресурс также становится помеченным.

Кроме того для анализа того, какую операцию выполняет инструкция, в алгоритме производится трансляция инструкции в набор высокоуровневых операций. Методика подробно описана в данной работе [11].

5. Описание программного комплекса. Рассмотрим общую архитектуру программного комплекса на рисунке 6. Отметим, что отладчик используется для мониторинга исключительных ситуаций при работе приложения. Согласно рисунку для сбора трассы и анализа помеченных данных используется система ДБИ Intel PIN [12] ввиду её превосходящей производительности над аналогами [13]. В рамках этой системы была реализована специальная утилита, реализующая возможность выполнения анализа помеченных данных и сбора трассы для последующего анализа и поиска уязвимостей.

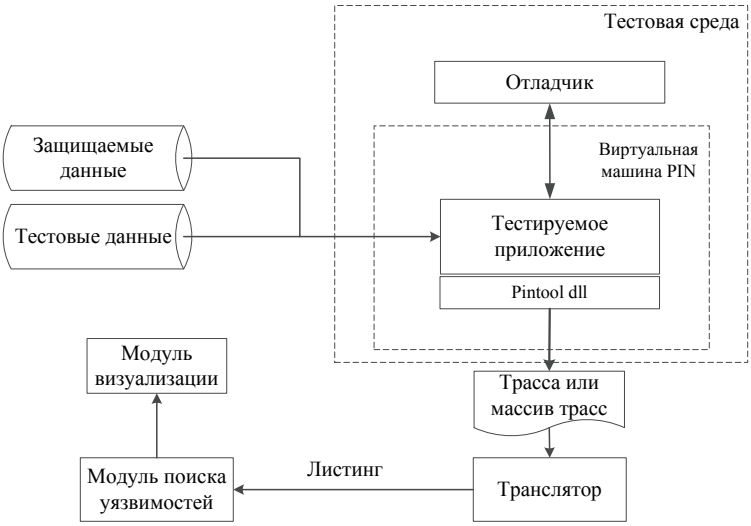

Рис. 6. Архитектура программного комплекса

Для анализа того, как помеченные данные обрабатываются в тестируемом приложении, прежде всего, необходимо учитывать потенциальные точки ввода этих данных в приложение. В качестве входных данных в работе рассматриваются сетевые данные, файлы или аргументы командной строки, а в качестве точек входа системные вызовы ОС (например, fgets или recv). Кроме того, необходимо отметить, что согласно алгоритму на рисунке 5 в качестве помеченных данных также выступают и адреса в памяти, задействованные в управлении программой (адреса возврата на стеке, таблица смещений виртуальных адресов процедур и т.п.) Программа выполняется на тестовых данных и параллельно осуществляется анализ продвижения помеченных данных в памяти и регистрах ПО.

На следующем этапе программа последовательно выполняется на каждом экземпляре тестовых данных, в результате чего имеем трассу или массив трасс (если программа требует перезапуск после каждой тестовой итерации). Затем осуществляется трансляция трассы, выделение инструкций оперирующих с памятью, определение множества фактических указателей, множества доступных участков памяти и проверка условий согласно алгоритму на рисунке 4 для каждого состояния в каждой из полученных трасс.

6. Экспериментальная оценка эффективности программного комплекса. Для сравнительной оценки эффективности системы поиска уязвимостей в трассе с аналогами, случайным образом было отобрано 24 уязвимых приложений из международной базы уязвимостей exploit-db на каждый тип уязвимости, порождающий каждый класс угроз к защищаемой информации (таблица 3). По 12 приложений для ОС Windows и Linux.

Таблица 3. Описание типов отобранных уязвимостей на каждый класс угроз

Класс угроз	Тип уязвимости	Описание уязвимости
Нарушение целостности защищаемой информации	Переполнение кучи	Запись за границами кучи
	Переполнение стека	Запись за границами стека
	Ошибка форматирования строк	Запись за границами стека
Нарушение доступности защищаемой информации	Использование неинициализированной памяти	Обращение к неинициализированной памяти
	Переполнение кучи	Перезапись управляющей информации, находящейся за границами кучи
	Переполнение стека	Перезапись адреса возврата из стека
	Ошибка форматирования строк	Перезапись управляющей информации за границами выделенной памяти
	Обращение по недопустимому адресу	Обращение по недопустимому адресу в памяти
Нарушение конфид-ти защищаемой информации	Использование освобожденной памяти	Чтение ранее освобожденной памяти в куче, содержащей защищаемую информацию
	Переполнение стека	Чтение защищаемой информации за границами стека
	Ошибка форматирования строк	Чтение защищаемой информации за границами стека
	Переполнение кучи	Чтение защищаемой информации за границами кучи

Каждый программный продукт разворачивался в собственной виртуальной среде на базе ОС Ubuntu Linux 12.10 и Windows 7 в следующей аппаратной конфигурации: x86 2 ГБ ОЗУ, Intel Core i7-3630QM, 2.4 GHz. Результаты работы систем приведены в таблице 4.

Таблица 4. Результаты экспериментального анализа систем детектирования уязвимостей (нет – уязвимость не обнаружена, да – уязвимость обнаружена)

Тип ошибки, приводящей к уязвимости	Наименование ПО и ОС. Идентификатор уязвимости	DrMemory	MemCheck	Разработ. система
Ошибки форматирования строк (конфиденциальность)	Dpkg 1.16.1.2 CVE-2014-8625	Да (Lin.)	Да (Lin.)	Да (Lin.)
	Mshtml.dll 6.0.2900.2853 Дата публикации: 03.09.2013	Нет (Win.)	Нет поддержки	Да (Win.)
Ошибки форматирования строк (целостность)	Yelp 2.23.1 CVE-2008-3533	Да (Lin.)	Да (Lin.)	Да (Lin.)
	ZipItFast 3.0 Дата публикации: 08.07.2011	Да (Win.)	Нет поддержки	Да (Win.)
Ошибки форматирования строк (доступность)	Dia 0.94 CVE-2006-2480	Да (Lin.)	Да (Lin.)	Да (Lin.)
	Opera 2.15 Дата публикации: 02.07.2013	Да (Win.)	Нет поддержки	Да (Win.)
Обращение по недопустимому адресу (доступность)	Squid 3.3.5 CVE-2013-4123	Нет (Lin.)	Да (Lin.)	Да (Lin.)
	KNeT Web Server 1.04 CVE-2005-0575	Да (Win.)	Нет поддержки	Да (Win.)
Переполнение стека (конфиденциальность)	Mozilla Firefox 3.0 CVE-2009-3382	Нет (Lin.)	Нет (Lin.)	Да (Lin.)
	Ezhometech EZ Server 6.4 CVE-2012-0618	Нет (Win.)	Нет поддержки	Да (Win.)
Переполнение стека (целостность)	Aireplay-ng 1.2 b3 CVE-2014-8322	Да (Lin.)	Да (Lin.)	Да (Lin.)
	Qualcomm Imapd 9.0.333.0 CVE-2014-10031	Да (Win.)	Нет поддержки	Да (Win.)

Продолжение таблицы 4.

Тип ошибки, приводящей к уязвимости	Наименование ПО и ОС. Идентификатор уязвимости	DrMemory	MemCheck	Разработ. система
Переполнение стека (доступность)	Unrar 3.9.3 Дата публикации: 05.08.2011	Да (Lin.)	Да (Lin.)	Да (Lin.)
	Httpdx 1.4.5 CVE-2009-4769	Да (Win.)	Нет поддержки	Да (Win.)
Переполнение кучи (конфиденциальность)	OpenSSL 1.0.1 CVE-2014-0160	Нет (Lin.)	Нет (Lin.)	Да (Lin.)
	XM Easy FTP Server 5.3.0 CVE-2007-1195	Нет (Win.)	Нет поддержки	Да (Win.)
Переполнение кучи (целостность)	inetutils 1.8.1 (ftp) Дата публикации: 07.12.2010	Нет (Lin.)	Нет (Lin.)	Да (Lin.)
	UltraVNC 1.0.1 CVE-2007-1195	Нет (Win.)	Нет поддержки	Да (Win.)
Переполнение кучи (доступность)	make 3.81. Дата публикации: 24.07.2014	Нет (Lin.)	Нет (Lin.)	Да (Lin.)
	Opera 2.12 CVE-2013-16380	Нет (Win.)	Нет поддержки	Да (Win.)
Использование освобожденной памяти (конфиденциальность)	OpenSSL 1.0.0a CVE-2010-2939	Нет (Lin.)	Нет (Lin.)	Нет (Lin.)
	ESTSoft ALPPlayer 2.0 Дата публикации: 06.07.2011	Да (Win.)	Нет поддержки	Да (Win.)
Использование неинициализированной памяти (доступность)	OpenLiteSpeed 1.3.9 Дата публикации 18.05.2015	Да (Lin.)	Да (Lin.)	Да (Lin.)
	UMPlayer 0.95 (QTGui4.dll) Дата публикации: 29.11.2012	Да (Win.)	Нет поддержки	Да (Win.)
Все	Итого	6/12 (Lin.) 7/12(Win.)	7/12 (Lin.)	11/12 (Lin.) 12/12 (Win.)

Дадим пояснения к таблице и эксперименту. В рамках эксперимента каждое приложение запускалось с использованием системы детектирования уязвимостей, а затем в приложение подавались тестовые

данные, которые приводили к срабатыванию уязвимости, после чего проводился анализ результата работы системы. Отметим, что система MemCheck не поддерживают ОС Windows, поэтому возможность её тестирования в этой ОС отсутствует.

Таким образом, разработанная система выполнила обнаружения 23 из 24 уязвимостей в приложениях для ОС Windows и Linux, что на 4 типа уязвимостей больше, чем обнаружили системы Memcheck и DrMemory в Linux и на 5 типов уязвимостей больше системы DrMemory в Windows.

Данный результат обусловлен тем, что разработанная система позволяет выполнять детектирование тех типов уязвимостей, которые не поддерживаются другими системами. Система выполнила обнаружение следующих типов уязвимостей, которые не были обнаружены другими системами:

– нарушение конфиденциальности защищаемой информации вследствие реализации уязвимости типа переполнение стека, переполнения кучи и использования освобожденной памяти (только Windows);

– переполнение кучи, приводящее к нарушению целостности и доступности защищаемой информации.

Отметим, что пропущенная уязвимость в Linux обусловлена внутренней ошибкой в среде ДБИ, а не разработанной системой.

7. Заключение. В данной работе была предложена оригинальная модель автоматизированного поиска уязвимостей в исполняемом коде и её алгоритмическое обеспечение.

Разработанная модель автоматизированного поиска уязвимостей в трассе программы позволила формально определить критерии возникновения уязвимости в исполняемом коде и осуществлять поиск уязвимостей для каждого выполненного состояния в трассе с учетом распределения защищаемой информации в памяти программы. В свою очередь алгоритмическое обеспечение модели позволило реализовать модель в виде программного комплекса.

Проведенная экспериментальная оценка программного комплекса программы показала, что разработанное решение позволяет детектировать на 5 типов уязвимостей больше в ОС Windows и на 4 типа уязвимостей больше в Linux по сравнению с существующими аналогами. Все модули разработанного комплекса были опубликованы как ПО с открытым исходным кодом, могут свободно использоваться в других проектах и доступны для скачивания в Интернете [14].

Дальнейшие перспективы видятся в использовании методики поиска уязвимостей в памяти анализируемого приложения в обход стандартных интерфейсов обработки пользовательских данных, а также расширение программного комплекса для тестирования программ отличных от x86/64 архитектур процессора.

Литература

1. *Gogul B., Reps T., Melski D., Teitelbaum T.* WYSINWYX: What you see is not what you execute // ACM Transactions on Programing Languages and Systems. 2010. vol. 32 . no. 6. pp. 202–213.
2. Исследование международной базы уязвимостей CVE. URL: http://opennet.ru/opennews/art.shtml?num=36287 (дата обращения 17.08.2015).
3. *Аветисян А.И., Белеванцев А.А., Чукляев И.И.* Технологии статического и динамического анализа уязвимостей программного обеспечения // Вопросы кибербезопасности. 2014. № 3 (4). С. 20–28.
4. ГОСТ Р 50922-2006 Защита информации. Основные термины и определения // М.: Стандартинформ. 2008.
5. *Dowd M., McDonald J., Schuh J.* The art of software security assessment: Identifying and preventing software vulnerabilities // Boston. USA: Addison-Wesley Professional. 2006. 1244 p.
6. *Каушан В.В., Маркин Ю.В., Падарян В.А., Тихонов А.Ю.* Методы поиска ошибок в бинарном коде // Технический отчет Института системного программирования РАН. 2013. № 2013-1. 42 с.
7. *Clarke T.* Fuzzing for software vulnerability discovery: technical report // Department of Mathematic. Royal Holloway University of London. 2009. 178 p.
8. *Bruening D., Zhao Q.* Practical memory checking with Dr. Memory // In Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization. 2011. pp. 213–223.
9. *Nethercote N., Seward J.* Valgrind: a framework for heavyweight dynamic binary instrumentation // ACM Sigplan notices. 2007. vol. 42(6). pp. 89–100.
10. Программный комплекс DrMemory. URL: www.drmemory.org (дата обращения 17.08.2015).
11. *Шудрак М.О., Лубкин И.А., Золотарев В.В.* Методика декомпиляции бинарного кода и её применение в сфере информационной безопасности // Безопасность информационных технологий НИЯУ МИФИ. Москва. 2012. №3. С. 75–80.
12. *Luk C.K., Cohn R., Muth R., Patil H., Klauser A., Lowney G., Hazelwood K.* Pin: building customized program analysis tools with dynamic instrumentation // In Acm Sigplan Notices. 2005. vol. 40(6). pp. 190–200.
13. *Uh G. R., Cohn R., Yadavalli B., Peri R., Ayyagari R.* Analyzing dynamic binary instrumentation overhead // In WBIA Workshop at ASPLOS. 2006.
14. Репозиторий с исходным кодом разработанной системы. URL: https://github.com/MShudrak/tvc (дата обращения 19.08.2015).

References

1. Gogul B., Reps T., Melski D., Teitelbaum T. WYSINWYX: What you see is not what you execute. ACM Transactions on Programing Languages and Systems. 2010. vol. 32. no. 6. pp. 202–213.
2. Issledovanie mezhdunarodnoj bazy uyazvimostej CVE [A taxonomy of the international vulnerabilities database CVE]. Available at: http://opennet.ru/opennews/art.shtml?num=36287 (accessed 17.08.2015). (In Russ.).
3. Avetisan A.I., Belevancev A.A., Chukliaev I.I. [The technologies of static and dynamic analyses for detecting software vulnerabilities*] Voprosy kiberbezopasnosti – CyberSecurity Issues*. 2014. vol. 3(4). pp. 20 – 28. (In. Russ.).
4. GOST R 50922-2006 [Information Security. Terms and Definitions]. M.: Standartinform. 2008. (In Russ.).
5. Dowd M., McDonald J., Schuh J. The art of software security assessment: Identifying and preventing software vulnerabilities. Boston. USA: Addison-Wesley Professional. 2006. 1244 p.
6. Kaushan V.V., Markin U.V., Padarian V.A. Tihonov A.U. [Techniques of bugs detection in machine code] *Texnicheskij otchet Tehnicheskij otchet Instituta sistemnogo*

programmirovanija RAN –Technical report of Institute of System Programming RAS. 2013. no. 2013-1. 42 p. (In Russ.).
7. Clarke T. Fuzzing for software vulnerability discovery: technical report. Department of Mathematic, Royal Holloway University of London. 2009. 178 p.
8. Bruening D., Zhao Q. Practical memory checking with Dr. Memory. In Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization. 2011. pp. 213–223.
9. Nethercote N., Seward J. Valgrind: a framework for heavyweight dynamic binary instrumentation. ACM Sigplan notices. 2007. vol. 42(6). pp. 89–100.
10. Programmnyj kompleks DrMemory [DrMemory system]. Available at: www.drmemory.org (accessed 17.08.2015). (In Russ.).
11. Shudrak M.O., Lubkin I.A., Zolotarev V.V. [The technique of binary code decompilation and its application in cyber security sphere]. *Bezopasnost' informacionnyx texnologij NIYaU MIFI – Security of Information Technologies MEPhI*. 2012. vol. 3. pp. 75–80. (In Russ.)
12. Luk C. K., Cohn R., Muth R., Patil H., Klauser A., Lowney G., Hazelwood, K. Pin: building customized program analysis tools with dynamic instrumentation. In Acm Sigplan Notices. 2005. vol. 40. no. 6. pp. 190–200.
13. Uh G. R., Cohn R., Yadavalli B., Peri R., Ayyagari R. Analyzing dynamic binary instrumentation overhead. In WBIA Workshop at ASPLOS. 2006.
14. Repozitorij s ishodnym kodom razrabotannoj sistemy [The system's source code repository]. Available at: https://github.com/MShudrak/tvc (accessed 19.08.2015). (In Russ.).

Шудрак Максим Олегович — аспирант кафедры безопасности информационных технологий, ФГБОУ ВПО Сибирский государственный аэрокосмический университет имени академика М.Ф. Решетнёва. Область научных интересов: безопасность информационных технологий, верификация программного обеспечения, тестирование безопасности программного обеспечения. Число научных публикаций — 14. slender.bit@mail.ru; пр. им. газеты «Красноярский рабочий» д,31, Красноярск, 660037; р.т.: +79233088703.

Shudrak Maksim Olegovich — Ph.D. student of information security department, Siberian State Aerospace University (SibSAU). Research interests: software verification, vulnerabilities analysis, malware analysis. The number of publications — 14. slender.bit@mail.ru; 31, Krasnoyarsky rabochiy Av., 660037, Russia; office phone: +79233088703.

Золотарев Вячеслав Владимирович — к-т техн. наук, доцент, доцент кафедры безопасности информационных технологий, ФГБОУ ВПО Сибирский государственный аэрокосмический университет имени академика М.Ф. Решетнёва. Область научных интересов: анализ рисков, верификация программного обеспечения, анализ безопасности программного кода, анализ вредоносного кода. Число научных публикаций — 29. imposer_89@mail.ru; пр. им. газеты «Красноярский рабочий» д,31, Красноярск, 660037; р.т.: +79233088703.

Zolotarev Vyacheslav Vladimirovich — Ph.D., associate professor, associated professor of information security department, Siberian State Aerospace University (SibSAU). Research interests: software verification, vulnerabilities analysis, malware analysis. The number of publications — 29. imposer_89@mail.ru; 31, Krasnoyarsky rabochiy Av., 660037, Russia; office phone: +79233088703.

Поддержка исследований. Работа выполнена при финансовой поддержке РФФИ (проект №14-07-31350-мол_а).

Acknowledgements. This research is supported by RFBR (grant №14-07-31350-mol_a).

РЕФЕРАТ

Шудрак М.О. Золотарев В.В. **Модель, алгоритмы и программный комплекс автоматизированного поиска уязвимостей в исполняемом коде.**

В статье рассматривается проблема поиска уязвимостей в исполняемом коде. В первом разделе работы рассматривается понятие ошибки и уязвимости, выделяются критерии отнесения ошибки к уязвимости, а также анализируются существующие недостатки методик динамического и статического анализа исполняемого кода. Отмечается, что существующие инструменты направлены на обнаружение ошибок без учёта тех угроз, которые они несут для защищаемой информации, которая обрабатывается в ПО.

Во втором разделе работы представлена оригинальная модель автоматизированного поиска уязвимостей в трассе программы, которая позволяет формально определить критерии возникновения ошибки в исполняемом коде и их отнесения к уязвимости с учётом распределения защищаемой информации в памяти программы на каждом шаге работы программы. Кроме того приводится алгоритмическое обеспечение модели, а также программная реализация этих алгоритмов на базе системы динамической бинарной инструментации Intel PIN.

В работе приводятся результаты экспериментального сравнение эффективности разработанной системы, которые позволяют заключить, что разработанное решение позволяет детектировать на 5 типов уязвимостей больше в ОС Windows и на 4 типа уязвимостей больше в Linux по сравнению с существующими аналогами. В заключительном разделе делаются выводы и описываются перспективные направления дальнейшей работы.

SUMMARY

Shudrak M.O., Zolotarev V.V. **A Model, Algorithms and Software Tool for Vulnerabilities Detection in Machine Code.**

In the article we consider the problem of vulnerabilities detection in machine code. In the first section of the paper we discuss the concept of bug and vulnerability, stand out criteria for distinct bug from vulnerability, as well as provide disadvantages of the existing methods for dynamic and static analysis of executable code. In addition, we describe disadvantages of current solutions in case of possibility to detect vulnerabilities in view of threats to confidential information that is processed in vulnerable software.

In the second part of the article we propose original model of vulnerabilities detection in program trace that provides formal criteria to distinct bug from vulnerability taking into account distribution of protected information in the memory of software at each step of the test. In addition, algorithmic support of the model and its software implementation based on dynamic binary instrumentation framework Intel PIN are provided.

We conduct experimental evaluation of software implementation efficiency which demonstrates that our solution allows detecting 5 types of Windows software vulnerabilities more and 4 types Linux software vulnerabilities more than existing analogs. The final section summarizes the conclusions and describes future directions of the research.