В. С. Горбатов, А. А. Станкевичус

ЗАЩИТА ГЕТЕРОГЕННОЙ СРЕДЫ РАСПРЕДЕЛЕННЫХ ВЫЧИСЛЕНИЙ ОТ ВРЕДОНОСНОГО КОДА ЗАДАНИЯ

1. Язык реализации верификатора

Язык программирования Си является известным инструментальных средств, для которых существуют коммерческие и свободно распространяемые компиляторы практически под любую аппаратную и программную платформу, что является важнейшим фактором выбора для реализации в гетерогенной среде распределенных вычислений (СРВ). Кроме того, язык Си позволяет выполнять достаточно низкоуровневые операции над кодом и данными в памяти и реализовывать высокопроизводительные вычислительные приложения, в том числе защиту СРВ от вредоносного кода на тех же ЭВМ, для которых он предназначен. Безопасная библиотека и верификатор также реализованы на языке программирования Си. При необходимости безопасность верификатора для вычислительного узла СРВ может быть обеспечена путем применения к нему разработанной системы защиты.

2. Язык реализации пользовательского интерфейса

В последнее время для разработки пользовательского интерфейса все чаще применяется платформа .NET [1] и библиотека Windows Forms [2]. Существует реализация с открытым исходным кодом Mono [3], позволяющая исполнять разработанное на платформе .NET программное обеспечение (ПО) на ЭВМ с различными аппаратными платформами под управлением разных операционных систем. Платформа .NET предоставляет обширный набор средств для разработки и совместного использования отдельных программных компонентов на различных языках программирования, во многом похожа на платформу Java [4] и развивается компанией Microsoft с 2002 г. Она имеет множество преимуществ, в числе которых значится определенная гарантия безопасности для ЭВМ, применяющих разработанные на ней ПО.

Платформа .NET включает в себя две основные составляющие [5,6]: общеязыковую исполняющую среду (CLR) и библиотеку классов (FCL). Первая отвечает за исполнение кода и управление им, вторая содержит систему основных классов, наиболее часто используемых в ПО [7].

При исполнении кода CLR обеспечивает «сборку мусора», контролирует версии и выполнение программы в рамках привилегий, предоставленных программе. Предоставляемые привилегии зависят в том числе и от источника исполняемого кода. Для недоверенных источников эти привилегии будут ограничены. Такой подход к предоставлению прав исполняемому файлу мог бы решить многие вопросы безопасности, но существует так называемый небезопасный код, предоставляющий возможность самомодификации программы, с помощью которой возможен обход контроля привилегий. В рамках данной работы признано целесообразным отказаться от использования небезопасного кода, а проверка его отсутствия при этом возлагается на компилятор языка C#, позволяющий при включении соответствующего режима предотвратить компиляцию программ, содержащих небезопасный код.

Использование платформы .NET налагает ряд ограничений на ЭВМ, которые могут быть использованы при разработке прикладного ПО [3]. В частности, для версии 2.0 защищаемая ЭВМ должна обладать следующими показателями [8]: центральный процессор с частотой 400 МГц, ОЗУ объемом 96 Мб, 280 Мб места на жестком диске.

На сегодняшний день рассматриваемая технология активно развивается и на текущий момент последним продуктом является версия 4.0.NET Framework.

Основой .NET Framework является CLR, выполняющая следующие основные функции: управление кодом, управление памятью, управление потоками. В библиотеке классов FCL содержится коллекция типов, тесно интегрированная с CLR.

Процесс компиляции программы для платформы .NET происходит в два этапа [9]: первый этап — компиляция исходного кода в промежуточный язык CIL, создание метаданных для каждого управляемого модуля; второй этап — компиляция CIL в исполняемый код с использованием CLR.

CLR работает не с отдельными модулями (CIL-код и метаданные), а со сборками, представляющими собой группировку одного или нескольких модулей и файлов ресурсов [9]. Платформа .NET предполагает модель безопасности на основе идентификации сборки по строгому имени и по месту, откуда загружается сборка [9]. При использовании такого подхода возможно установление таких ограничений, при которых код, загружаемый из недоверенного источника, будет ограничен в правах и не будет представлять опасность для ЭВМ. Хотя при этом не учитывается возможность самомодификации небезопасного кода, которая может привести, например, к переполнению стека и выполнению произвольного кода.

Система безопасности приложений для платформы .NET базируется на нескольких основных механизмах [10,11]: безопасность типов, аутентификация, авторизация, разрешения.

Безопасность типов подразумевает использование строгой типизации данных. При использовании строгой типизации можно гарантировать, что типобезопасный код обращается только к тем данным, которые разрешены ему для доступа. Типобезопасность позволяет гарантировать, что объекты не смогут нанести друг другу непредумышленные или злонамеренные повреждения. Непосредственно во время JIT-компиляции проводится проверка безопасности с точки зрения типизации исполняемого CIL кода [12], т. е. подтверждения того, что код может получать доступ к памяти и вызывать методы только через правильно определенные типы [13]. Если проверяемый код не проходит проверку, то генерируется исключение. Проверка не производится, если код имеет разрешение на пропуск проверки.

Приложения, разработанные для аутентификации на платформе .NET Framework [14], могут использовать некоторые из доступных на настоящий момент механизмов, например, действующих на основе служб, предоставляемых операционной системой (NTLM или Kerberos).

Для авторизации используется информация, содержащаяся в объекте-участнике и связанном с ним объекте-удостоверении. Объект-участник представляет собой контекст безопасности, в котором работает код [15]. Доступ предоставляется на основании принадлежности объекта-участника определенной роли, которая представляет собой набор объектов-участников с равными полномочиями.

На основании разрешений, выданных коду, CLR позволяет коду выполнять определенные операции. Существует три вида разрешений [16]: разрешения для доступа к коду, разрешения идентификации, разрешения безопасности на основе ролей.

Разрешения для доступа к коду предоставляют право доступа к некоторому защищенному ресурсу или право выполнения защищенной операции (например, доступ к неуправляемому коду) [17]. Разрешения идентификации предоставляются CLR на основе информации, полученной о сборке, и используются для защиты кода от НСД. Например, можно установить разрешения идентификации таким образом, что код можно будет вызывать только из сборки с определенным строгим именем и загруженной из определенного места [18]. Разрешения безопасности на основе ролей предоставляют механизм для проверки: является ли пользователь участником роли или обладает ли пользователь указанным удостоверением [19].

В версии 4 .NET Framework, в отличие от ранних версий, CLR не использует политики безопасности для предоставления полномочий коду [20]. При исполнении управляемого кода CLR во время JIT-компиляции может удостовериться, что код работает в рамках предоставленных ему полномочий и получает доступ только к выделенным объектам в памяти. При исполнении небезопасного кода такая проверка не может быть проведена, следовательно, необходимо верифицировать небезопасный код для платформы .NET.

Язык C# является языком с контролем безопасности типов [21], что может позволить снизить количество ошибок и уязвимостей в ПО, разработанном на этом языке. Переносимость программ, разработанных на языке C#, ниже, чем у программ, разработанных на языке Си. Производительность многих программ, разработанных на языке C#, также ниже, чем производительность аналогичных программ, разработанных на языке Си.

Интерфейс разработчика программного обеспечения реализован на языке программирования C#. Производительность графического интерфейса разработчика программного обеспечения не так важна, как, например, производительность верификатора исходного кода или безопасной библиотеки. Переносимость и производительность разработанного средства защиты от вредоносного кода вычислительного задания не зависит от производительности интерфейса разработчика. Интерфейс разработчика не является необходимым для обеспечения безопасности вычислительного узла СРВ компонентом системы защиты от вредоносного кода выполняемого задания и служит лишь для повышения удобства разработки и отладки безопасного вычислительного задания.

3. Выбор препроцессора Си

В качестве препроцессора языка Си в реализованной системе защиты используется модифицированная версия препроцессора MCPP [22], который является платформо-независимым и свободно распространяется в исходных кодах. Лицензия, по которой производится распространение данного препроцессора, допускает модификацию его исходных кодов, что позволяет внести в него изменения, призванные обеспечить защиту ЭВМ от вредоносного воздействия со стороны препроцессора. Кроме того, безопасность препроцессора для вычислительного узла может быть обеспечена путем применения к нему разработанной системы защиты гетерогенной СРВ от вредоносного кода выполняемого задания.

4. Реализация лексического анализатора

Лексический анализ используется в компиляторах, интерпретаторах, верификаторах и многих других программах для выполнения ввода данных, заключается в распознавании во входном потоке символов лексем и пометке лексемы ее типом. Для наших целей использовался генератор лексических анализаторов Flex [23], который позволяет частично или полностью автоматизировать процесс написания лексического анализатора. Он является генератором программ, принимающим на вход высокоуровневую спецификацию, содержащую фрагменты кода на языке Си, и выдающим исходный код программы, разделяющей выходные данные на лексемы в соответствии со спецификацией, использованной при генерации программы. При этом после считывания очередной лексемы выполняется соответствующий фрагмент кода, включенный в спецификацию. Flex позволяет генерировать лексические анализаторы на языке Си.

5. Реализация синтаксического анализатора

Синтаксический анализатор производит построение абстрактного синтаксического дерева для анализируемой программы. Для решения нашей задачи использовался генератор синтаксических анализаторов Bison, инструментарий которого [24] позволяет генерировать синтаксические анализаторы, получающие на вход лексемы из входного потока, и выявлять соответствие между последовательностью лексем и одним из правил грамматики, задаваемых при генерации синтаксического анализатора. При выявлении соответствия правила последовательности лексем вызывается пользовательский код, выполняющий обработку данного правила.

6. Структуры данных

Информация о выделенных областях памяти хранится в виде списка структур, содержащих адреса и размеры выделенных областей оперативной памяти. Кроме этого, поддерживается дерево

структур содержащих начальные и конечные адреса выделенных диапазонов для осуществления быстрой проверки попадания заданного адреса в один из выделенных диапазонов.

Для представления элементов каждого из используемых абстрактных доменов разработаны специальные структуры данных и функции, реализующие операции над ними. Для представления элементов абстрактного домена двоичных n-мерных векторов использованы структуры, состоящие из одного байта, указывающего длину n вектора и следующего за ним массива из n байт, причем значение элемента этого массива с индексом i кодирует значение i+1-й координаты n-мерного вектора. При этом значения элемента массива 0,1,2,3 кодируют соответственно значения 0, 1, $\perp$, $\top$. Для представления элементов абстрактного домена n-мультиинтервалов используется массив структур, представляющих информацию об интервале. Для хранения границ интервалов используются знаковые 64-битные целые числа.

Для представления элементов абстрактного домена восьмиугольников используются массивы из 8 элементов, представленных 64-битными целыми числами, содержащих правые части каждого из 8 неравенств, описывающих восьмиугольник.

7. Методика оценки производительности защищенного кода

Методика оценки производительности защищенного кода определяет процесс тестирования разработанной системы защиты гетерогенной СРВ от вредоносного кода выполняемого задания. Предложено провести две фазы тестирования: модульное тестирование и системное тестирование, что позволило проконтролировать полноту функциональности испытываемой системы, а также отсутствие скрытых дефектов.

Процесс тестирования автоматизирован при помощи скриптов с целью обеспечения более высокой скорости выполнения тестов по сравнению с «ручным» вариантом.

Модульное тестирование. **На этом этапе проводится проверка выполнения функций каждого модуля системы защиты гетерогенной СРВ от вредоносного кода выполняемого задания, которая состоит в выявлении ошибок в реализации алгоритмов, локализованных в той или иной функции, таких как неправильная работа с условиями и счетчиками циклов, а также при использовании локальных переменных и ресурсов.**

В качестве показателя тестирования используется критерий путей. Этот критерий, как правило, предполагает использование большего числа тестов по сравнению с другими критериями, применяемыми на этапе модульного тестирования, но при этом обеспечивает более тщательное тестирование кода. Значение приемочной оценки успешности тестирования [25] принимается равным 80 %.

Системное тестирование. Этот процесс рассматривает испытываемую систему в целом, оперирует на уровне пользовательских интерфейсов [25] и позволяет выявлять системные дефекты, такие как неверное использование ресурсов, непредусмотренные комбинации данных пользовательского уровня, несовместимость с окружением, непредусмотренные сценарии использования, отсутствующая или неверная функциональность, неудобство в применении.

Используются следующие категории тестов фазы системного тестирования: полноты решения функциональных задач и стрессовое тестирование на предельных объемах нагрузки входного потока. По достижении для всех критериев приемочной оценки 80 % процесс системного тестирования считается успешно завершенным.

8. Экспериментальное исследование системы

Экспериментальное исследование разработанной системы проведено с целью сравнения с производительностью существующих аналогичных систем. Оценивалась производительность статического верификатора и производительность безопасного кода вычислительного задания.

Эксперименты проводились на ЭВМ с центральным процессором Intel Core 2 Duo с тактовой частотой 2 ГГц, 2 Гб ОЗУ под управлением Mac OS X версии 10.5.6.

Оценка производительности статического верификатора.

Для ее реализации были использованы специально разработанные тестовые программы, а также существующее свободно распространяемое ПО. Оценка времени верификации производилась при использовании различных политик безопасности, таких как «Проверка записи в ОЗУ», «Проверка чтения и записи в ОЗУ», «Обнуление выделяемых областей ОЗУ».

Результаты оценки производительности приведены в таблице 1, которая показывает, что время верификации существенно зависит от характера и объема кода и от применяемой политики безопасности.

Таблица 1. Результаты оценки производительности статического верификатора

Наименование ПО	Количество строк кода, шт.	Описание политики безопасности	Время анализа, сек.
Prime	591	Проверка записи	12
Prime	591	Проверка чтения, записи и обнуление выделяемых областей ОЗУ	17
Mst	1127	Проверка записи	57
Mst	1127	Проверка чтения, записи и обнуление выделяемых областей ОЗУ	193
RandAlloc	192	Проверка записи	125
RandAlloc	192	Проверка чтения, записи и обнуление выделяемых областей ОЗУ	149

Оценка производительности защищенного кода.

В этом случае для оценки производительности использовались тесты, которые могут быть условно разделены на 3 типа: А — интенсивно работающие с указателями, Б — интенсивно выделяющие и освобождающие оперативную память, В — производящие интенсивные математические вычисления.

Полученные результаты сравнивались с аналогичными исследованиями, проведенными с помощью специального программного средства CCured [26], которое также позволяет повысить уровень безопасности программ. Результаты экспериментов приведены в таблице 2, из которой можно сделать следующие выводы:

- производительность защищенного кода существенно зависит от характера кода и от применяемой политики безопасности;
- при использовании политики безопасности, разрешающей чтение произвольных областей памяти, производительность защищенных нашей системой программ превышает производительность программ, обработанных таким средством, как CCured.

Таблица 2. Результаты оценки производительности безопасного кода

Наименование измеряемой характеристики	Тип программ		
	А	Б	В
Падение производительности небезопасной версии (раз)	1,00	1,00	1,00

Падение производительности при выделении памяти в безопасном пуле без дополнительных проверок (раз)	1,07	0,97	1,03
Падение производительности при проверке записи (раз)	1,61	1,57	1,02
Падение производительности при проверке чтения и записи (раз)	2,36	1,72	1,03
Падение производительности при проверке чтения, записи и обнулении памяти (раз)	3,18	1,93	1,05
Падение производительности при защите с использованием только CCured (раз)	2,04	1,83	1,21

СПИСОК ЛИТЕРАТУРЫ:

1. *Просиз Дж.* Программирование для Microsoft .NET / Пер. с англ. М.: Издательско-торговый дом «Русская редакция», 2003. — 704 с.: илл.
2. MSDN: Windows.Forms reference documentation. URL: http://msdn.microsoft.com/ en-us/library/dd30h2yb.aspx.
3. Mono project. URL: http://www.mono-project.com.
4. *Gosling J., Joy B., Guy S., Bracha G.* The Java Language Specification. Second edition. 2000. URL: http://java.sun.com/docs/books/jls/ second_edition/html/j.title.doc.html.
5. *Нейгел К., Иввен Б., Глинн Д., Скиннер М., Уотсон К.* C# и платформа .NET 3.0 для профессионалов. М.: Диалектика, 2008.
6. Библиотека MSDN. Общие сведения о платформе .NET. URL: http://msdn.microsoft.com/library/zw4w595w.aspx.
7. Microsoft. .NET Framework overview. URL: http://www.microsoft.com/net/overview.aspx.
8. Требования к системе для .NET Framework. URL: http://msdn.microsoft.com/ru-ru/library/8z6watww.aspx.
9. *Рихтер Д.* CLR via C#. Программирование на платформе Microsoft .NET Framework 2.0 на языке C#. СПб.: Питер, 2008.
10. *Казаков Т.* Механизмы безопасности в .NET. URL: http://www.rsdn.ru/article/dotnet/netsecurity.xml.
11. Основные понятия безопасности. URL: http://msdn.microsoft.com/ru-ru/library/z164t8hs.aspx.
12. *Merrill Br., Drayton P., Albahari B.* C# Essentials. 2nd Edition. O'Relly. 2002. —216 p.
Строгая типизация и безопасность. URL: http://msdn.microsoft.com/ru-ru/library/hbzz1a9a.aspx.
13. Mcpp — a portable C preprocessor with Validation Suite. URL: http://mcpp.sourceforge.net. Процесс управляемого выполнения. URL: http://msdn.microsoft.com/ru/library/k5532s8a.aspx#compiling_msil_to_native_code.
14. Flex: The Fast Lexical Analyzer. URL: http://flex.sourceforge.net. Руководящий документ. Защита от несанкционированного доступа к информации. Термины и определения.
15. Bison — GNU parser generator. URL: http://www.gnu.org/software/bison/. Объекты Principal и Identity. URL: http://msdn.microsoft.com/ru-ru/library/ftx85f8x.aspx.
16. Разрешения безопасности. URL: http://msdn.microsoft.com/ru-ru/library/5ba4k1c5.aspx.
17. Разрешения для доступа к коду. URL: http://msdn.microsoft.com/ru-ru/library/h846e9b3.aspx.
18. Разрешения идентификации. URL: http://msdn.microsoft.com/ru-ru/library/d3wktt6a.aspx.
19. Разрешения безопасности на основе ролей. URL: http://msdn.microsoft.com/ru-ru/library/7sxk9k2h.aspx.
20. Изменения системы безопасности в платформе .NET Framework 4. URL: http://msdn.microsoft.com/ru-ru/library/dd233103.aspx.
30. *Котляров В. П.* Основы тестирования программного обеспечения. М.: Интернет-университет информационных технологий. ИНТУИТ.ру. 2006. — 288 с.
21. Строгая типизация и безопасность. URL: http://msdn.microsoft.com/ru-ru/library/hbzz1a9a.aspx.
22. Процесс управляемого выполнения. URL: http://msdn.microsoft.com/ru/library/k5532s8a.aspx#compiling_msil_to_native_code.
23. Руководящий документ. Защита от несанкционированного доступа к информации. Термины и определения.
24. Объекты Principal и Identity. URL: http://msdn.microsoft.com/ru-ru/library/ftx85f8x.aspx.
25. *Котляров В. П.* Основы тестирования программного обеспечения. М.: Интернет-университет информационных технологий. ИНТУИТ.ру. 2006. — 288 с.
26. *Necula G. C., McPeak S., Weimer W.* CCured: type-safe retrofitting of legacy code // Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. New York, USA: ACM, 2002. P. 128—139.