Модел. и анализ информ. систем. Т. **18**, № **4** (2011) 131–143

УДК 004.492+519.7

Тестирование безопасности программного обеспечения на языке C с использованием верификатора SPIN[1]

Кушик Н.Г.[1], Маммар А.[2], Кавалли А.[2],
Евтушенко Н.В.[1], Джиминез В.[2], Монте де Ока Е.[3]

[1] *Томский государственный университет, Томск, РФ*
[2] *Институт Телекоммуникаций и Менеджмента, Еври, Франция*
[3] *Монтимаж, Париж, Франция*

e-mail: amel.mammar@it-sudparis.eu, ninayevtushenko@yahoo.com, edgardo.montesdeoca@montimage.com

получена 20 сентября 2011 года

Ключевые слова: Программа, уязвимость, язык C, верификация, SPIN

Предлагается метод тестирования безопасности C программ с использованием широко известного верификатора SPIN. Обсуждаются классы уязвимостей в C программах, которые могут быть обнаружены с использованием предложенного подхода. Приводятся результаты компьютерных экспериментов по обнаружению уязвимостей в студенческих программных реализациях алгоритмов работы с массивами.

1. Введение

Задача безопасности программного обеспечения становится все более актуальной, особенно для программного обеспечения, используемого в критических системах (к которым относятся системы связи, медицина, ряд протоколов в системах связи и др.), где большое количество программ написано на языке программирования C. Достаточно часто под безопасным понимается программное обеспечение, в котором отсутствуют уязвимости, поэтому в настоящее время большое внимание уделяется методам поиска уязвимостей в C программах.

При поиске уязвимостей выделяют два самостоятельных направления. Первое направление связано с анализом программного кода без непосредственного исполнения программы (статический поиск уязвимостей); второе направление, в свою очередь, требует непосредственного исполнения программного кода (динамический поиск уязвимостей) [1]. Методы статического поиска уязвимостей хорошо изучены,

[1]Работа частично поддержана финансированием в рамках конкурса межвузовских проектов НИ ТПУ.

и на данный момент существует довольно много программ-анализаторов кода, распространяемых в свободном доступе [2]. Большинство известных методов динамического поиска уязвимостей, в основном, сводятся к случайной генерации входных данных программы (search based testing) [3]. Поэтому особый интерес представляют методы динамического обнаружения уязвимостей в программном обеспечении, когда на программу подаются не случайные входные данные, а только те, которые могут привести к ее некорректной работе.

Один из возможных способов автоматического получения входных данных для динамического поиска уязвимостей заключается в использовании верификаторов (model checkers) [4, 5]. Верификатор проверяет, найдутся ли такие входные данные для программы, что выполняется свойство, соответствующее наличию проверяемой уязвимости. Если такие данные найдены, то верификатор предоставляет необходимый контрпример. Далее на основе полученного контрпримера формируются входные данные, которые подаются на проверяемую программу, и, если программа ведет себя некорректно, то уязвимость найдена; в противном случае можно заключить, что верификация, которая обычно проводится по некоторой модели программы, была выполнена некорректно. В данной работе мы показываем, каким образом некоторые классы уязвимостей могут быть обнаружены с использованием хорошо известного верификатора SPIN. SPIN является программой-верификатором, распространяемой в свободном доступе, соответственно эта программа доступна на официальном сайте SPIN [5]. Экспериментальные исследования проводились со студенческими программами, реализующими алгоритмы работы с массивами (вычисление среднего значения, сортировка, поиск минимального/максимального элемента), которые широко используются в различных приложениях. Поскольку эффективность поиска контрпримера существенно зависит от алгоритма генерации входных данных, отдельный раздел настоящей статьи посвящен обзору возможных методов такой генерации для верификатора SPIN.

2. Уязвимости в C программе и методы их поиска с использованием верификатора SPIN

Как уже отмечалось, мы рассматриваем уязвимость как особое свойство C программы, и для того чтобы проверить, обладает ли программа данным свойством, мы используем программу-верификатор SPIN [5], которая позволяет осуществлять динамический поиск уязвимостей в программе. При использовании SPIN необходимо создать модель исходной программы в специальном языке PROMELA; программа в языке PROMELA подается на вход верификатора SPIN. Соответственно возникает вопрос о том, каким образом транслировать инструкции языка C в инструкции языка PROMELA. Язык PROMELA предназначен для работы с распределенными системами, и вместо операторов и функций, используемых в языке C, в данном языке присутствуют процессы, среди которых выделяется главный процесс (подобно функции main() в C) и которые выполняются параллельно. Если программа содержит уязвимость, которая сохраняется в ее PROMELA модели, то, как и любой верификатор, SPIN выдает контрпример с соответствующими значениями ло-

кальных переменных или входных данных. В этом случае возможно осуществить «инъекцию тестовых данных» (test injection), т.е. посылку «плохих» данных на программу с целью демонстрации программисту части кода, в которой имеется ошибка. Однако поскольку язык PROMELA является языком моделирования, а не языком программирования, то построенная PROMELA спецификация в общем случае не эквивалентна исходной С программе, и, вообще говоря, не всякий контрпример, сгенерированный SPIN, может сразу продемонстрировать уязвимость при инъекции. Таким образом, инъекция теста необходима также для проверки корректности трансляции инструкций языка С в инструкции языка PROMELA.

В данной работе под *уязвимостью* мы понимаем свойство программы, которое позволяет пользователю нарушить ее конфиденциальность, целостность и/или доступность. Пусть имеется множество уязвимостей С программы. Если С программа не обладает ни одним из свойств заданного множества, то она считается *безопасной* относительно заданного множества уязвимостей, в противном случае – небезопасной. В данной работе мы рассматриваем три типа уязвимостей: переполнение буфера (buffer overflow), отрицательное переполнение (arithmetic underflow) и уязвимости повторного освобождения памяти (double free vulnerability). Для переполнения буфера, которое очень часто возникает в С программах, мы рассматриваем три самостоятельных случая: переполнение типа (type overflow), переполнение приведения типа (type conversion overflow) и переполнение в массивах (array overflow). Мы также обращаем особое внимание на так называемое строковое переполнение (string overflow), которое может возникнуть при копировании одной строки в другую в С коде. Мы отмечаем, что предложенные методы поиска уязвимостей применимы к достаточно узкому классу программ, написанных на языке С. В частности, программы в языке С++, использующие объектно-ориентированное программирование, в данной статье не рассматриваются. Более того, тестируемые на безопасность программы не должны содержать вложенных последовательностей функций. С одной стороны, такие ограничения сужают класс программ, к которым применим предлагаемый метод обнаружения уязвимостей, с другой стороны, как правило, на уязвимости тестируют не все приложение целиком, а лишь некоторые его части (например, некоторую функцию или блок инструкций).

2.1. Переполнение типа

Переполнение типа может возникнуть в С программе в случае, когда значение некоторого выражения e присваивается переменной v, т.е. $v = e$. Если переменная v имеет тип t и значение выражения e превышает максимальное значение для данного типа (обозначение: max_t), то в этом случае в С программе имеется уязвимость переполнения типа. Подобная уязвимость может быть продемонстрирована на небольшом примере, когда $e = n1 + n2$. В этом случае в соответствующем С коде имеются инструкции

```
t v;
v = n1 + n2;
```

Чтобы описать данный случай в языке PROMELA (задать PROMELA спецификацию), мы описываем три процесса на языке PROMELA. Два из них используются для моделирования входных данных ($n1$ и $n2$), в то время как третий процесс используется для верификации свойства $n1 + n2 < (max\_t + 1)$. Данное свойство описывается в PROMELA как *утверждение* (assertion), которое не должно нарушаться. В случае если SPIN сигнализирует о нарушении данного утверждения (assertion violation), можно, как правило, сделать вывод о наличии уязвимости переполнения типа в исходной C программе. Ниже в таблице 1 представлен пример трансляции C кода в PROMELA код. Заметим, что в целях оптимизации в соответствующем коде переменная max_t принимает значение 256 (максимальное для типа *unsigned char*, увеличенное на единицу).

Если SPIN обнаруживает нарушение утверждения $n1 + n2 < (max\_t + 1)$, то SPIN предоставляет соответствующий контрпример (значения переменных $n1$ и $n2$). В данном примере, когда переменная v имеет тип *unsigned char*, верификатор выдал контрпример, в котором $n1 = 205$ и $n2 = 200$. Если же переменные $n1$ и $n2$ не являются непосредственными входными данными программы, то при генерации контрпримера SPIN выдает все значения переменных, от которых зависят значения $n1$ и $n2$, и, таким образом, появляется возможность наблюдать как значения внутренних переменных программы, так и значения внешних переменных, которые могут вызвать соответствующую уязвимость в исходной C программе. Для подтверждения наличия данной уязвимости данные из контрпримера подаются на вход программы (test injection).

В приведенном выше примере мы проиллюстрировали обнаружение переполнения типа в случае работы с байтами (тип *unsigned char*). В этом случае мы проверяли, может ли значение переменной v оказаться больше 255. Данный подход естественным образом распространяется на иные типы переменных языка C, такие как *int*, *short*, *unsigned short* с соответствующей границей max_t и соответствующим типом переменных языка PROMELA. Для C типов *bool*, *int* и *short* типы в PROMELA идентичны; для работы с типом *unsigned short* в PROMELA необходимо описать тип *unsigned*, под переменные которого отводится 16 бит.

2.2. Переполнение приведения типа

Аналогично уязвимости переполнения типа можно обнаружить переполнение приведения типа. Такой тип уязвимости может возникнуть в C коде, когда в переменную $v1$ типа $t1$ заносится значение переменной $v2$ типа $t2$, при этом $max\_t1 < max\_t2$. В качестве примера рассмотрим переменную $t1$ типа *int* и переменную $t2$ типа *char*. Пусть также $v2$ есть входная переменная C программы. В этом случае C программа содержит следующий набор инструкций

```
char v1;
int v2;
scanf(‘‘%d’’, &v2);
v1 = v2;
```

Таблица 1. Соответствие С и PROMELA программ для уязвимости переполнения типа

Программа в языке С	Программа в языке PROMELA

int main()
{
unsigned char n1, n2, v;
scanf(“%d”,&n1);
scanf(“%d”,&n2);
v = n1 + n2;
return 0;
}
	byte n1, n2,n3;
	int max_t;
	int done;
	proctype input1()
	{
	n1 = n1 + 100;
	do
	::
	if
	:: (true) -> {
	if
	::(n1 * 2 < 256) -> n1 = n1 * 2;
	::(true) ->n1 = n1 - 5;
	fi
	}
	:: (true) -> {break;}
	fi
	od;
	done=1;
	}
	proctype input2()
	{
	n2 = n2 + 100;
	do
	::
	if
	:: (true) -> {
	if
	::(n2 * 2 < 256) -> n2 = n2 * 2;
	::(true) ->n2 = n2 - 5;
	fi
	}
	:: (true) -> {break;}
	fi
	od;
	done=1;
	}
	active proctype main()
	{
	max_t = 256;
	run input1();
	done==1;
	done=0;
	run input2();
	done==1;
	assert((n1 + n2) < max_t);}

Для построения PROMELA модели такой программы мы, как в случае 1, описываем специальный процесс *input* для генерации значения переменной $v2$, и отдельный процесс отводится на верификацию соответствующего свойства. Последний процесс, в котором переменная max_t1 увеличена на единицу, будет иметь следующий вид:

```
init
{
  max_t1 = 256;
  run input();
  done == 1;
  assert(v2 < max_t1);
}
```

Как видно из приведенного примера, уязвимость переполнения типа и уязвимость приведения типа очень похожи, и их проверка осуществляется практически одинаково. Поэтому эти уязвимости часто объединяют и называют общим термином "переполнение типа".

2.3. Переполнение в массивах

Уязвимость переполнения в массивах может возникнуть в случае, когда программист работает с массивом a размера $size\_a$ и использует переменную $a[i]$ при $i >= size\_a$. Рассмотрим небольшой пример, когда i есть входной параметр программы. Чтобы обнаружить данную уязвимость, мы вновь описываем два PROMELA процесса, первый из которых используется для моделирования значений параметра i, а второй – для верификации соответствующего свойства. В таблице 2 представлен пример трансляции программы в языке C в программу в языке PROMELA. Для этого примера SPIN выдает нарушение свойства $i < size\_a$ при $i = 1073741824$.

2.4. Отрицательное переполнение

Уязвимость отрицательного переполнения по смыслу очень похожа на уязвимость переполнения буфера (иногда даже считается ее частным случаем). Отрицательное переполнение может возникнуть в C программе в случае, когда значение некоторого выражения e заносится в переменную v, т.е. $v = e$. Если переменная v имеет тип t и значение выражения e меньше минимального значения данного типа (обозначение: min_t), то в этом случае в C программе имеется уязвимость отрицательного переполнения. Для обнаружения данной уязвимости мы строим PROMELA спецификацию, аналогично таковой из пункта 1; различие состоит лишь в формулировке утверждения для проверки свойства. В данном случае необходимое PROMELA утверждение имеет вид $assert((min\_t - 1) < e)$.

2.5. Строковое переполнение

Уязвимость строкового переполнения может возникнуть в C программе, когда программист копирует одну строку в другую. В качестве примера для данной уяз-

Таблица 2. Соответствие С и PROMELA программ для уязвимости переполнения в массивах

Программа в языке С	Программа в языке PROMELA

const int size_a = 100;
int main()
{
int a[size_a];
scanf("%d",&i);
return 0;
}
	int i; int size_a; int done;
	proctype input()
	{ i = i + 1;
	do ::
	if :: (true) -> i = i * 2;
	:: (true) -> break;
	fi
	od;
	done=1;
	}
	init
	{
	size_a = 100;
	run input();
	done == 1;
	assert(i < size_a);
	}

вимости можно рассмотреть хорошо известную функцию *strcpy*(*char* ∗ *dest*, *const char* ∗ *src*) библиотеки *string.h*. Строковое переполнение возникает в программе, когда программист использует эту функцию, и размер строки *src* (обозначение: *size_src*) превышает размер строки *dest* (обозначение: *size_dest*). Мы отмечаем, что данная уязвимость может быть легко обнаружена верификатором SPIN. В этом случае соответствующее PROMELA утверждение имеет вид *assert*(*size_src* <= *size_dest*).

2.6. Уязвимость повторного освобождения памяти

Некорректная работа программиста с памятью может также привести к наличию уязвимости. Уязвимость повторного освобождения памяти возникает при объявлении в С программе указателя $*p$ и повторном освобождении программистом памяти, выделенной под p. Заметим, что работа с указателями при помощи операторов *new* и *delete* возможна только в языке С++, а не «чистом» языке С, однако связанная с ними уязвимость повторного освобождения достаточно легко обнаруживается предложенным методом, поэтому метод ее обнаружения также представлен в настоящей статье.

Чтобы обнаружить данную уязвимость, в PROMELA программе мы объявляем переменную v, которой вначале присваиваем значение нуль. Далее мы «сканируем» исходную С программу инструкция за инструкцией; каждый раз при встрече оператора *new* в С (С++) коде в соответствующей PROMELA программе мы увеличиваем значение v на единицу. И наоборот, каждый раз при встрече оператора

Таблица 3. Соответствие C и PROMELA программ для уязвимости повторного освобождения памяти

Программа в языке C	Программа в языке PROMELA
int main();	int v;
{	proctype v_calc()
int *p;	{
p = new int[5];	v = v + 1;
p[4] = 3;	v = v - 1;
printf("p[4] = %d",p[4]);	v = v - 1;
delete []p;	}
delete []p;	init
return 0;	{
}	run v_calc();
	assert((-1 < v) && (v < 2));
	}

delete в C (C++) программе мы уменьшаем значение переменной v на единицу. Если в исходной C программе нет уязвимости повторного освобождения памяти, то переменная v в PROMELA коде принимает значение 0 или 1, поэтому соответствующее утверждение имеет вид $assert((-1 < v)\&\&(v < 2))$. В таблице 3 представлена соответствующая трансляция программы в языке C в программу в языке PROMELA для случая, когда элементы массива p имеют тип int.

Предложенные методы обнаружения уязвимостей были реализованы программно, однако поскольку задача построения PROMELA модели по C программе является трудоемкой, при проведении компьютерных экспериментов по обнаружению уязвимостей в C программах некоторые PROMELA модели были доработаны вручную. Кроме того, поскольку модель, как обычно, не отражает полностью свойства моделируемого объекта, осуществляется проверка, что контрпример для PROMELA программы действительно соответствует «плохим» входным данным для исходной программы на C. Такая проверка осуществляется инъекцией входных данных из контрпримера в исходную C программу. В наших экспериментах все уязвимости, найденные по PROMELA модели, соответствовали уязвимостям в исходных программах. Отметим также, что генерация контрпримера является самостоятельной задачей, от которой существенно зависит результат обнаружения уязвимостей, поэтому в следующем разделе статьи мы кратко рассматриваем возможные методы генерации входных данных.

3. Генерация входных данных

Время, затраченное на генерацию контрпримера верификатором SPIN, существенно зависит от того, какой алгоритм генерации входных данных заложен в PROMELA модели, и в этом разделе кратко обсуждаются различные способы такой генерации. Заметим, что достоинством верификатора SPIN и языка моделирования PROMELA

является возможность недетерминированного исполнения инструкций, в частности, за счет параллельного выполнения PROMELA процессов. Соответственно генерация входных данных будет осуществляться не строго по заложенному алгоритму, и требуемый контрпример может быть найден за более короткое время. На очередном шаге SPIN может случайным образом выбирать процесс для исполнения. Соответственно для генерации входных данных отводится несколько процессов для использования такого недетерминизма. Ниже кратко описываются некоторые способы генерации входных данных, которые могут быть реализованы в модели PROMELA для исходной С программы. Пусть далее через n обозначается переменная типа t, значение которой необходимо сгенерировать; min_t и max_t суть нижняя и верхняя границы типа t.

3.1. Генерация входных данных с шагом

Рассмотрим, каким образом можно генерировать входные данные для верификации программ на примере программы с одним входным данным n. Метод начинает работу с переменной n, которой присваивается нижняя граница типа t, т.е. $n = min\_t$. Как правило, после предварительных экспериментов задается значение $step$, соответствующее шагу, с которым будет осуществляться генерация. Очередное значение переменной n определяется как сумма текущего значения этой переменной и шага $step$, т.е. $n = n + step$. Этот процесс повторяется до тех пор, пока не будет найден контрпример, или пока n не превысит max_t. Контрпример при такой генерации может быть найден не всегда, однако чем меньше шаг, тем более вероятно, что контрпример будет найден.

3.2. Скачкообразная генерация входных данных

Аналогично первому методу начальное значение переменной n совпадает с нижней границей min_t. Далее выбираются два шага $step1$ и $step2$. Первый шаг $step1$ является «большим» и определяет движение значения переменной n от нижней границы к верхней, т.е. $n = n+step1$. Такое движение продолжается до тех пор, пока не будет найден контрпример, либо пока очередное значение n не превысит верхнюю границу max_t. Значение второго шага $step2$ является «небольшим» и используется для движения от верхней границы к нижней, т.е. $n = n - step2$. Так, например, генерация входных данных $n1$ и $n2$ в примере раздела «Уязвимость переполнения типа» начинается с начального значения нуль. К очередному сгенерированному значению прибавляется число 100, далее значение увеличивается вдвое ($step1 = n$). Увеличение значения переменных $n1$ и $n2$ происходит до тех пор, пока сгенерированное число не выйдет за пределы максимального значения для типа $byte$, т.е. 255. Если граница 255 достигнута, то от сгенерированного числа итеративно отнимается число 5. Такой скачкообразный процесс, состоящий в сложении с «большим» (близким к границе для типа) числом и вычитании «маленького» числа, позволяет найти контрпример для уязвимости переполнения типа, равный $n1 = 205$ и $n2 = 200$, достаточно быстро (за секунду или меньше).

3.3. Генерация граничных значений

Как отмечается в литературе, в ряде случаев достаточно эффективным оказывается граничное тестирование (boundary testing) [7, 8] . В этом случае значение переменной n полагается равным min_t, и далее с «небольшим» шагом $step$ (чаще всего единица) осуществляется увеличение текущего значения n. Процесс увеличения $n = n + step$ прекращается через k итераций, причем k достаточно мало. Если контрпример не найден, то переменной n присваивается значение max_t. Далее n уменьшается не более чем k раз с шагом $step$ до тех пор, пока не будет найден контрпример (если это возможно).

Заметим, что наиболее эффективной обычно является комбинация перечисленных методов. Кроме того, при решении конкретной задачи метод генерации входных данных для функции $scanf$ в PROMELA модели, а также других функций ввода, определяется условиями задачи и/или на основе предварительных исследований. В дальнейшем авторы планируют провести экспериментальные исследования, в рамках которых предложенные методы генерации входных данных будут реализованы и использованы для тестирования безопасности студенческих C программ, реализующих различные алгоритмы.

4. Экспериментальные результаты поиска уязвимостей в C программах

В данном разделе мы представляем результаты компьютерных экспериментов по обнаружению уязвимостей в C программах. В качестве экспериментальных реализаций были выбраны студенческие программы и функции, написанные на языке C. К таким программам, например, относятся программы, содержащие различные манипуляции с массивами: поиск минимального (максимального) элемента в массиве, различные сортировки массивов, вычисление среднего значения. Мы отмечаем, что такие программы пишутся довольно легко и занимают сравнительно мало места, однако широко используются на практике, поскольку в общем случае элементами массива могут быть не стандартные типы данных, а абстрактные, на которых могут быть заданы различные отношения, в частности, отношение порядка. Для проверки наличия уязвимости в C программе методом, описанным в разделе 1, инструкции программы транслируются в PROMELA инструкции, а затем в нужные места вставляются соответствующие утверждения. Уязвимости переполнения типа были обнаружены в студенческих программах поиска среднего значения в массиве. При подаче на вход программ данных из сгенерированных контрпримеров программы не выдали никаких предупреждений о вводе некорректных данных, и до проведенного тестирования программы считались работающими правильно.

В результате проведения экспериментов уязвимость переполнения в массивах была обнаружена также в других C программах, где память под массив выделяется статически, а затем пользователем вводится реальный размер массива. Вот некоторые из них:

1. программа поиска минимального (максимального) элемента в массиве;

2. программа сортировки массива методом пузырька;

3. программа сортировки массива методом прямой вставки;

4. программа нахождения простых чисел в интервале $[1, n]$ для заданного n (решето Эратосфена).

Отметим также, что для всех тестируемых программ не требовалось повторной генерации контрпримера SPIN, что говорит как о корректной трансляции инструкций языка С в инструкции языка PROMELA, так и об эффективности предложенного метода динамического поиска уязвимостей в программном обеспечении.

Таким образом, в результате проведения экспериментов с доступными реализациями широко используемых на практике алгоритмов работы с массивами в программах были обнаружены уязвимости переполнения типа и уязвимости переполнения в массивах. Как отмечается в работе [9], данные уязвимости не всегда могут быть обнаружены статическими анализаторами кода. Соответственно подтверждается необходимость использования динамических методов поиска уязвимостей. Кроме того, во всех программах, приведенных в разделе 1, в которые были умышленно внесены все рассматриваемые уязвимости, данные уязвимости были полностью обнаружены.

5. Заключение

В статье рассмотрены методы поиска уязвимостей в программном обеспечении, написанном на языке С, с использованием верификатора SPIN. Причиной разработки нового подхода к динамическому поиску уязвимостей в программном обеспечении является недостаток средств статического анализа программного кода. Поскольку SPIN принимает на вход программы, написанные в языке PROMELA, в статье также приведен ряд правил для трансляции инструкций языка С в инструкции языка PROMELA. Проведенные со студенческими программами эксперименты подтверждают известный факт о необходимости тестирования безопасности программного обеспечения, как статическими, так и динамическими методами, и возможность использования SPIN для тестирования безопасности С программ. Поскольку трансляция программ в языке С в язык PROMELA является трудоемкой, в дальнейшем планируется усовершенствовать подход к обнаружению уязвимостей в программном обеспечении, опустив операцию трансляции. В частности, планируется оценить возможность использования некоторого промежуточного языка, например, промежуточного языка С (C Intermediate Language или CIL) [10]. Возможность использования SPIN для верификации С программ на основе CIL описана в [11]. С другой стороны, избежать трансляции инструкций языка С в инструкции языка PROMELA возможно, если анализировать безопасность программ, написанных на другом языке, например, Java и использовать соответствующий верификатор, например, JPF [4].

Авторы выражают благодарность студенту РФФ ТГУ Антону Ермакову за помощь в проведении компьютерных экспериментов.

Список литературы

1. Willy Jimenez, Amel Mammar, Ana R. Cavalli. Software Vulnerabilities, Prevention and Detection Methods. A Review, SEC-MDA workshop, Enschede, The Netherlands (24 June 2009).

2. Харитонова Е. Поиск уязвимостей в программах с помощью анализаторов кода [Электрон. статья]. Режим доступа: http://www.codenet.ru/progr/other/code-analysers.php?rss=1, свободный.

3. Phil McMinn. Search-based Software Test Data Generation: A Survey. Softw. Test., Verif. Reliab. 2004. 14(2). P. 105–156.

4. JPF. The swiss army knife of JavaTM verification [Электрон. статья]. Режим доступа: http://javapathfinder.sourceforge.net/, свободный.

5. SPIN [Электронный ресурс]. Режим доступа: http://spinroot.com, свободный.

6. Holzmann Gerard J. SPIN Model Checker: The Primer and Reference Manual. Addison-Wesley Professional. Published Sep. 2003. 4.

7. Software Testing Glossary [Электронный ресурс]. Режим доступа: www.aptest.com/glossary.htm#bvatesting.

8. Котляров В.П., Коликова Т.В. Основы современного тестирования программного обеспечения, разработанного на C#: Учебное пособие / под редакцией В.П. Котлярова. СПб., 2004. 170 с.

9. Ermakov Anton, Kushik Natalia. Detecting C Program Vulnerabilities // Proc. of the 5th Spring/Summer Young Researchers Colloquium on Software Engineering. P. 61–64.

10. George C. Necula, Scott McPeak, Shree P. Rahul, Westley Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs // International Conference on Compiler Construction. 2002. P. 213–228.

11. Alex Groce, Rajeev Joshi. Extending model checking with dynamic analysis // Proc. of the VMCAI. 2008. LNCS 4905.

A SPIN-based Approach for Detecting Vulnerabilities in C Programs

Kushik N.G., Mammar A., Cavalli A., Yevtushenko N.V., Jimenez W., Montes de Oca E.

Keywords: C Program, vulnerability, C language, model checking, SPIN

The C language is widely used for developing tools in various application areas, and a number of C software tools are used for critical systems, such as medicine, transport, etc. Correspondingly, the security of such programs should be thoroughly tested, i.e., it is important to develop techniques for detecting vulnerabilities in C programs. In this paper we present an approach for dynamic detection of software vulnerabilities using the SPIN model checker. We discuss how this approach can be implemented in order to detect automatically C code vulnerabilities and illustrate a proposed technique for a number of C programs which are widely used in a number of applications.

Сведения об авторах:
Кушик Наталья Геннадьевна,
Томский государственный университет, аспирант;
Маммар Амель,
Институт Телекоммуникаций и Менеджмента Франции, исследователь;
Кавалли Ана,
Институт Телекоммуникаций и Менеджмента Франции,
профессор, директор департамента;
Евтушенко Нина Владимировна,
Томский государственный университет,
профессор, зав. кафедрой;
Джиминез Вилли,
Институт Телекоммуникаций и Менеджмента Франции, аспирант;
Эдгардо Монте де Ока,
компания "Монтимаж", генеральный директор.