Ya. A. Bekeneva
Saint Petersburg Electrotechnical University «LETI»

ANALYSIS OF DDoS-ATTACKS TOPICAL TYPES AND PROTECTION METHODS AGAINST THEM

In the paper an analysis of popular types of DDoS-attacks based on traffic reflection and traffic amplification is introduced. Existing methods for prevention and protection against such attacks are considered. Advantages and disadvantages of these methods are described. Necessity of new protection methods development is shown.

DDoS-attack, DNS, NTP, SSDP, SNMP, DDoS-attack protection, RRL, RAD

УДК 004.052.2

Р. Хаберланд, С. А. Ивановский, К. В. Кринкин
Санкт-Петербургский государственный электротехнический университет «ЛЭТИ» им. В. И. Ульянова (Ленина)

Верификация объектно-ориентированных программ с динамической памятью на основе ссылочной модели

Предложен метод аксиоматичной верификации динамической памяти на основе логического языка программирования "Пролог" для объектно-ориентированных программ, использующих ссылочную модель памяти. Формально определены термины: куча, интерпретация кучи и рассмотрены формальные операции над кучами. Обобщенная реализация на Прологе разрешает преодолевать недостатки в выразимости. Сделаны предположения, что представленным подходом можно решить совокупность других актуальных проблем в той же области. Предполагаемая модель была проверена с помощью системы верификации [1], которая, в том числе, способна преобразовывать программы императивных языков с экземплярами классов в промежуточное представление.

Указатели, кучи, анализ псевдонимов, верификация динамической памяти

Разделяемые кучи. Реализация спецификации куч на Прологе может быть представлена различными моделями памяти, например глобальными графовыми моделями, моделями с функциями переходов или моделями с делением на кучи [2], [3]. В модели с делением на кучи используются «*распределенные кучи*» (модель согласной «Логики распределенной памяти» (ЛРП) [1]) с указанием связи вида «*а указывает на б*» или как «*а→б*», где «*а*» – идентификатор простого или объектного типа. Значением «*б*» может быть любое из предполагаемого (в том числе, объектного) домена T (см. далее). Это соответствие может быть представлено различными способами. В Прологе соответствие «*а→б*» представлено кортежами (*а*, *б*), например [(*h*2, 1), (*h*2, *h*3), (*h*3, 3)]. По определению, в модели ЛРП все кучи не пересекаются и не связаны между собой, кроме случая, когда такая связь задается дополнительным условием. Считается, что кучи могут иметь взаимосвязи. В этом случае необходимо определить связи между указателями явным образом в спецификации, как это сделано для указателей *h*1, *h*2 и *h*3 в приведенном ранее примере. Без ограничения общности можно считать, что определения куч с помощью ссылок достаточно, и ссылки на ссылки ради простоты далее не рассматриваются.

Определение кучи. Кучей является обобщенная и связанная графовая структура данных, которая располагается и меняется в динамической части памяти при запуске программы. Граф описывается типизированными ячейками памяти и связями между ними. Куча может иметь любое количество указателей (в том числе ни одного). Без ограничения общности, можно считать, что указатели определены либо стеком, либо кучей. Подкучей называем любой связный подграф.

Определение формулы для куч. Формула H для утверждения о динамической памяти (*кучи*) определяется рекурсивно:

$$H ::= \textbf{true} \mid \textbf{false} \mid \exists\, x.H \mid x \rightarrow \text{Expr} \mid H * H \mid H \vee H \mid H \wedge H \mid \textbf{emp} \mid \text{pred(params)}.$$

Здесь **true** и **false** являются константными формулами. Переменный символ x может квалифицироваться как экзистенциальный, если x не встречается как связанный символ в принадлежащей подформуле либо как всеобщный в противном случае. Оператор '→' связывает типизированный указатель 'x' с неким выражением. Кроме того, утверждения могут быть связаны между собой с помощью конъюнкции, дизъюнкции и бинарного оператора '*'; **emp** является встроенным предикатом без аргументов. Вызов предиката осуществляется с помощью pred(params), где params содержит список термов в качестве аргументов. Expr является неким значением по адресу x. Значением может являться любой экземпляр класса, целое число либо пустой указатель «nil». Формула H преобразуется явным и интуитивным образом в прологовские термы.

Определение интерпретации куч. Интерпретация формулы $H^J[h]$ для данной кучи h и некоего утверждения H является отображением на булевское множество.

Если H является формулой **true**, то $\textbf{true}^J[h]$ равно «*истина*» для любой кучи h (аналогично для формулы **false**). Предикат **emp** верен для пустой кучи h и ложен во всех остальных случаях. Формула «x→Expr» истинна, если h содержит ровно одну подходящую кучу, чье содержимое имеет значение Expr. Заранее определенный предикат pred выполним для данного списка аргументов params либо не выполним.

Бинарный оператор распределения куч '*' можно определить как деление на левостороннюю и правостороннюю кучи (см. следующее определение). Оператор '*' является коммутативным. Все разделяемые между '*' кучи всегда содержат только разовые идентификаторы на левых сторонах всех выражений «*а→б*», т. е. если имеются некие идентификаторы a_0, a_1 для $a_0 \rightarrow b_0$ и $a_1 \rightarrow b_0$, то из этого может лишь следовать при $a_0 <> a_1$, что sAddr(a_0)<>sAddr(a_1), sContent(a_0) = sContent(a_1) и hAddr(b_0) = hAddr(b_1), где sAddr выявляет адрес в статической части памяти; sContent выявляет содержание определенного идентификатора, а hAddr выявляет адрес в динамической памяти, а следовательно, псевдонимное отношение между a_0 и a_1 должно быть добавлено к '*' (см. далее).

Определение оператора '*'. Две кучи $H1$, $H2$ *существуют независимо* друг от друга в динамической памяти тогда и только тогда, когда указатели на них существуют в стеке или в кучах и все подкучи из $H1$ не зависят от всех подкуч $H2$, и наоборот. В этом случае обозначим обе кучи с бинарным оператором '*' как $H1*H2$. Если $H1$ и $H2$ связаны между собой, то (1) существует некое связывающее утверждение $H3$, (2) существует подкуча $H2'$ так, что $H2'*H1$ соблюдается и (3) аналогично существует $H1'$ для $H1'*H2$. Таким образом, утверждение $H1'*H2'*H3$ верное. По определению, кучи могут быть иерархическими, т. е. оператор '*' неассоциативен. Любая программная команда может модифицировать любое количество куч.

Фреймовая теорема. Для куч $F = F_0 * F_1 * \ldots * F_n$, где $F_j = f_j \rightarrow f_j$, с $n \geq 0$, предусловие P и постусловие Q при условии, что вызов некой процедуры S не изменяет кучи фрейма F, «*фреймовое правило*» [3] гласит, что в антецеденте правила $\{P\}S\{Q\}$ |– $\{F*P\}S\{F*Q\}$ достаточно доказать тройку Хора $\{P\}S\{Q\}$ без фрейма F.

Практически это означает, что если имеется вызов процедуры S, то нет необходимости определять все кучи полностью. Указываются лишь те кучи, которые меняются, – это P и Q. Теорема, на первый взгляд, выглядит простой, однако если имеется сеть объектов, то передача одного объекта потенциально изменяет всю сеть объектов, а следовательно, может и нарушить установленное свойство при верификации.

Определение псевдонима. Если на один элемент динамической памяти указывают 2 любых указателя, то один из них становится *псевдонимом* второго указателя. Если, например, предположить спецификацию: {h2->1*h3->3}, то {(h2->1*h3->3)^h2->_X123} не имеет утечку, а {h2->1*h1->1*h3->3} имеет утечку, так как h1 не допускается согласно спецификации. Но если h1 является псевдонимом h2 (т. е. $h1 = h2$ вместо h1->1), тогда спецификация выполнима.

В [4] вводится определение «а*бстрактного предиката*», под абстракцией подразумевается модульность предикатов для описания куч. Если расширить определение абстрактного предиката, то, например, с помощью сопоставления образцов можно решить ряд нынешних ограничений, например: предикатные символы, литералы си-

мулируют переменные, имеющие полностью вычисленное значение, как в императивных языках программирования, а вызов предикатов представляет собой лишь вызов процедур. Далее, переменные символы могут быть использованы, в отличие, например, от Пролога, в предикатах только в некоторых местах (см. далее). Возможность сопоставлять (любые термовые) объекты не предусмотрена. Утверждения объектов замкнуты и не расширяются, имеются ограничения в связи с наименованием, с местом определения и со способом их использования.

Синтаксис и семантика Пролога поддерживают символы и хорошо приспособлены к выводу SLD-унификации над термами, поэтому есть основание предполагать, что символы и переменные в Прологе ближе к логическим формулам, чем представленные действующие подходы.

Промежуточное представление объектов. Абади и Лейно [5] обходятся без классов, и используемые объекты не рассматриваются как классы в [6], а как экземпляры записи. В [5] объектные поля не содержат указателей и поэтому могут содержать внутренние объекты. Однако если не вводить дополнительные ограничения, например типизацию, то нетипизированное вычисление с объектами может оказаться неверным или неполным. По этой причине далее рассматриваются только экземпляры объектов, генерируемые классами. В [6] используются классы и вводятся типовые системы для объектов, но не рассматриваются рекурсивные определения классов, указателей или объекты с псевдонимами. Объекты не делят между собой общую часть памяти. В [7] Борна предлагает моделировать объекты как массивы фиксированного размера, однако методы преобразуются к функциям как в процедурной парадигме.

Определение типов. Типы T определяются как целочисленный тип Z либо как построенный из уже существующих типов с указанием имени:

$$T ::= Z \mid \text{class id } [\ f_i\text{: } T,\ m_j\text{: } \underline{T}\ ],$$

где $i, j \geq 0$, $\underline{T} ::= T_0 \rightarrow \ldots \rightarrow T_k$, $k \geq 0$.

Класс состоит из произвольного числа различных полей f_i и методов m_j. Объект имеет локальное наименование id вместе с экземпляром типа Z или class, в случае чего id является указателем. Проверка принадлежности объекта к определенному классу или подклассу осуществляется с помощью транзитивного отношения наследования из соответствующего предиката в Прологе. При размещении объектов в динамической памяти представление и реализация методов не предполагают их размещение в динамической памяти. При порождении экземпляра типа T всем полям объекта в случайном порядке присваиваются соответствующие типы значения. Без ограничения общности считается, что все поля должны иметь начальное определенное значение.

Определение прологовских категорий. Классовые типы добавляются как прологовские факты. Методы связываются с соответствующим классом. В случае, когда поле из подкласса совпадает с полем из родительского класса, унаследованные поля переименовываются соответственно. Любой класс или метод получает глобальное и однородное наименование. Самоизменяемый код в методах запрещается.

К примеру, классовое определение прямоугольника объекта C класса cRectangle содержит поле a и метод area, который присваивается полю, и площадь исходя из данной длины h и ширины w:

```
C = class(cRectangle, [variable(a,int),
function(area,int,[variable(w,int),variable(h,int)],
[assign(a,mul(w,h)), return(a)])].
```

Примером экземпляра объекта C класса cRectangle может послужить:

```
[(c,object(cRectangle, [(x,int,5), (y,int,3),
(a,int,0)])),_,_],
```

где обе "_" определяют еще две любые посторонние кучи. Заметим, что Пролог интерпретирует термы по необходимости и что прологовская спецификация может содержать переменные, например N123, которые могут являться ссылками на любые объекты или любые другие термовые представления.

Утверждение. Выбранная выше классовая модель объектов для Пролога является полной, т. е. объекты полностью могут быть представлены как прологовские термы.

Это следует из тотального отображения модели. Если имеется рекурсивное определение класса, то преобразуется лишь его имя, которое будет найдено в классовой среде. Если такого имени нет, считается, что правильное имя класса неизвестно. Возможные конфликты в связи с наследованием не нужно рассматривать по отдельности из-за универсального преобразования указанного ранее. Если объект содержит псевдонимы, прологовский терм будет содержать переменную, которая используется также в иных частях прологовских термов. Однако любые содержания принципиально

не допускаются, например A = object(A, A) – что, кстати, не унифицируемо из-за рекурсивного содержания переменного символа A самого себя, но может быть опущено при дальнейшем рассмотрении без ограничения общности. Будем считать, что объектные поля, по определению, никогда не пересекаются, но указанное содержимое полей, конечно, может содержать псевдонимы. Далее '*' распространяется на объекты без ограничения.

Правила доказательств.

Определение абстрактного предиката. Аксиоматичные правила верификации $a_0, ..., a_n \vdash b$ можно преобразовать в прологовские правила с некими прологовскими подцелями $a_0, ..., a_n$ и головой b. Порядок вычисления $0 \le i < n$ для всех последовательных a_{i+1} должен соблюдаться слева направо.

Более того, для a_i и a_{i+k}, $\forall i. \exists k: 0 \le i \le n-k$, $0 \le k \le (n-i)$ аргументы параметров подцелей содержат переменные, которые должны передаваться для всех последующих $a_0, ..., a_n$. Во избежание возможности неопределенности переменных при вычислении подцелей условимся для простоты, но без ограничения общности, что после достижения каждой подцели все переменные, ее содержащие, определены полностью. Будем считать, что прологовские правила приводятся сначала в соответствующую форму. Предположим, что такое преобразование невозможно и имеются две неупорядочивающиеся подцели. Отсюда следует, что существует двусторонняя зависимость между двумя подцелями, т. е. этот случай содержит неразрешаемый цикл. Следовательно, соответствующее доказательство недостижимо. Обратим внимание на то, что предикаты подцелей a_j коммутируют над '*' и предикаты могут вызывать предикаты далее.

Определение суждения. Верификация над динамической памятью имеет следующий тип: Env ***x*** Stm → Env, где Stm является программным оператором, а Env = (Stack, Heap). Здесь Stack – состояние стека, а Heap – состояние кучи.

В начале верификации Env = $(\varnothing, \varnothing)$. Среда обновляется при выполнении программы, например при выполнении нового блока, которое добавляет к актуальному стеку все переменные блока, а оператор выделения новой памяти добавляет новый элемент в кучу. Доказательство в Прологе задается поиском: если доказательство как вызов некой интерпретации предикатов, т. е. подцелей, завершается как fail, тогда доказана противоречивость некой подцели, доказательство продолжается поиском альтернативы – если такая осталась. При доказательстве локальные переменные хранятся в стеке и существуют лишь в определенном блоке управления. Примеры команды условного перехода выглядят так:

```
01 proof(ite(COND, IFBLOCK, ELSEBLOCK),
                    Env, Env2):-
02  Env = (Stack, Heap),
03  tolist(IFBLOCK, IFBLOCK2),
04  proofs(IFBLOCK2, ([COND | Stack], Heap),
                    (_, Heap1)),
05  tolist(ELSEBLOCK, ELSEBLOCK2),
06  proofs(ELSEBLOCK2,
      ([ne(COND)|Stack],Heap), (_,Heap2)) …
```

Ради простоты в верхнем примере proof предполагается, что COND содержит выражения, которые ссылаются только на символы из статической памяти, иначе необходимо прежде всего применить фильтр, который отделит динамические ячейки памяти от статических и продолжит верификацию. В строке 06 ne определяет функтор неравенства. В строке 01 определяется заголовок предиката, далее следуют цели, которые необходимо соблюдать, прежде чем proof становится выполним. Строки 04 и 06 анализируют случаи условной команды, когда COND верно и неверно. Env2 содержит стек и кучу, как они должны выглядеть, если proof выполнимо. Вспомогательный предикат proofs/3 совершает левостороннюю свертку, при этом первый параметр является списком команды, второй параметр описывает начальный стек и кучу, а последний параметр описывает окончательный стек и кучу; proofs является экземпляром инициальной алгебры с proof/3 в качестве эндофунктора. Предикат tolist/2 преобразует любой прологовский терм в список, кроме списков, которые остаются без изменений. Проверка согласно тройке Хора проводится анализом Env и Env2 с утверждениями, встроенными в программу, т. е. искусственным программным оператором assert.

В начале статьи были введены основные положения о верификации динамической памяти на основе ссылочной логики. Были определены понятия об интерпретации формулы куч и фреймовое правило как основные подходы для решения выявленных проблем в динамической памяти с классовым типом вычисления в Прологе. Объектно-ориентированные программы преобразовались на примерах в прологовские термы и рассматривались ограничения. В дальнейшим планируется представить программную систему верификации и изучить вопросы, связанные с абстрактными предикатами, кото-

рые дают преимущество расширимости предикатов – добавление новых встроенных и пользовательских определений в модульном виде.

Авторы статьи выражают свое глубокое уважение Сергею Алексеевичу Ивановскому, который внес большой вклад в подготовку и написание данной статьи, но, к огромному сожалению и печали своих коллег и учеников, не дожил до сегодняшнего дня.

СПИСОК ЛИТЕРАТУРЫ

1. Haberland R., Ivanovskiy S. Dynamically Allocated Memory Verification in Object-Oriented Programs using Prolog. Spring // Summer Young Research Colloquium on Software Engineering, Saint Petersburg, 2014. P. 46–50.

2. Burstall R. M. Some Techniques for Proving Correctness of Programs which Alter Data // Machine Intelligence. 1972. № 7. P. 23–50.

3. Reynolds J. C. Separation Logic: A Logic for Shared Mutable Data Structures // Proc. of the 17th Annual IEEE Symp. on Logic in Computer Science (LICS'02), Washington, 2002. P. 55–74.

4. Parkinson M. Local Reasoning for Java. PhD thesis. Cambridge University, 2005.

5. Abadi M., Leino K. R. M. A Logic of Object-Oriented Programs // Lecture Notes in Computer Science. 1997. Vol. 1214. P. 682–696.

6. Abadi M., Cardelli L. A Theory of Objects. New York: Springer, 1996.

7. Bornat R. Proving Pointer Programs in Hoare Logic // Lecture Notes in Computer Science. 2000. Vol. 1837. P. 102–126.

R. Haberland, S. A. Ivanovskiy, K. V. Krinkin
Saint Petersburg Electrotechnical University «LETI»

HEAP VERIFICATION OF OBJECT-ORIENTED PROGRAMS IN A POINTS-TO LOGIC

In this paper a verification technique is presented based on the logical programming language "PROLOG" for object-oriented programs based on the points-to heap model. The terms "heap", "heap interpretation" and a heap theory are introduced, and a generalised approach in PROLOG allows to bridge expressibility gaps as well as other research problems in this field as discussed in the paper. The chosen model was validated against [1], which also provides facilities to transform imperative programming languages with objects into an intermediate representation in PROLOG.

Pointers, heaps, alias analysis, dynamic memory verification

УДК 159.9 (303.732)

С. А. Колмаков, А. В. Леонов
Санкт-Петербургский государственный электротехнический университет «ЛЭТИ» им. В. И. Ульянова (Ленина)

Методика выявления ошибок человека-оператора с учетом его психологического типа на основании игровых программ, эмулирующих процесс работы на АРМ

Представлено описание методики выявления ошибок человека-оператора с учетом его психологического типа на основании игровых программ, эмулирующих процесс работы на АРМ. Приведенная методика поможет усовершенствовать методы профотбора за счет того, что ошибки перестанут носить случайный характер, а станут прогнозируемыми и устранимыми.

Человек-оператор, психологический тип, ошибки человека-оператора, MBTI, игровые программы, профессиональный отбор

В настоящее время количество и сложность человекомашинных систем (СЧМ) увеличиваются, при этом ответственность человека-оператора (Ч-О) растет пропорционально сложности СЧМ.