Машины Шенхаге могут быть уязвимыми

С.А. Жуков

Аннотация **— В статье рассматривается модель вторжения в виде разрушения памяти одного типа абстрактного исполнителя алгоритмов, введенного Шенхаге. Определяются два сценария вторжения (атака без последействия и атака с последействием) и предложенными в статье макрокомандами программируются варианты разрушений памяти этого исполнителя. Показывается, что даже ограниченные возможности машин Шенхаге (злоумышленника) позволяют выполнить как частичное, так и полное разрушение памяти атакованной машины Шенхаге (жертвы). Под разрушением памяти понимается ее деструктуризация, при которой доступ к части ее элементов или всех элементов становится невозможным. Однако, для определенного класса машин Шенхаге (машин с регулярной структурой памяти) можно выполнить детекцию их памяти, чтобы установить факт ее повреждения. Регулярность структуры памяти понимается как возможность описания этой структуры графовой грамматикой, задающей конечное множество шаблонов структур, тиражируемых при конструировании порождаемого графа определенным в грамматике способом. Приводятся примеры машин Шенхаге, структура памяти которых описывается отмеченными выше грамматиками. В статье не рассматриваются источники угроз. Изучаются лишь некоторые формы угроз и возможность их осуществления средствами машин Шенхаге.**

Ключевые слова**— графовая грамматика, макрос, модель вторжения, поиск в глубину, указательная машина.**

I. Введение

Выявленные и изученные уязвимости в обеспечении компьютерной безопасности относятся к аппаратуре, системным и прикладным программам, а также протоколам взаимодействия вычислительных систем с традиционной высокоразвитой архитектурой [1, 2]. Если вторжения и атаки на информационную безопасность осуществимы и многообразны, то этот феномен должен проявляться и на базовом уровне фундаментальных алгоритмических моделей. В теории вычислений известен ряд абстрактных вычислительных устройств, возможности и свойства которых изучались с точки зрения их вычислительной мощи или сложности. В таких моделях вычисления выполняются в “дружественной” обстановке, когда вычислителя не атакуют. Является ли такой режим вычислений безальтернативным, и абстрактные исполнители алгоритмов исключают возможность атаки на результаты вычислений в силу самой их природы? Возможно, что ответ на этот вопрос будет разным для разных исполнителей алгоритмов.

В статье рассматривается модель атаки и разные варианты атак на память одного типа абстрактного исполнителя алгоритмов, введенного А. Шенхаге. Изложение описания и результатов исследования вычислительных возможностей этого класса абстрактных устройств, которые для краткости назовем машинами Шенхаге, дано в [3].

В данной работе показывается, что даже ограниченные возможности машин Шенхаге (злоумышленника) позволяют выполнить как частичное, так и полное разрушение памяти атакованной машины Шенхаге (жертвы). Однако, для определенного класса машин Шенхаге (машин с регулярной структурой памяти) можно выполнить детекцию их памяти, чтобы установить факт ее повреждения. Чтобы схема работы машины Шенхаге, которая сначала атакуется, а затем проверяется на наличие возможной атаки, была логически возможной, в статье рассматриваются два сценария проведения атаки: без выполнения машиной (жертвой) действий после атаки и с продолжением действий для выявления факта атаки. В статье не рассматриваются источники угроз. Изучаются лишь некоторые формы угроз и возможность их осуществления средствами машин Шенхаге.

Для замкнутости изложения дадим краткое введение в формализм машин Шенхаге, опираясь на авторское определение из [3].

Статья получена 15.10.2020.

Жуков Сергей Александрович, кандидат физико-математических наук, доцент кафедры вычислительных технологий Кубанского госуниверситета (e-mail: szhucov@fpm.kubsu.ru)

Машины Шенхаге не имеют памяти в традиционном понимании. Информация, введенная в машину, запоминается в форме структуры связей между узлами, а ее обработка выражается изменением конфигурации узлов и межузловых связей. Машина имеет входную и выходную ленты, свойства которых аналогичны свойствам ленты машин Тьюринга (МТ). Кроме того, есть внутренняя среда хранения – Δ-структура, которая является обобщением многомерной ленты МТ. Так же, как и головка МТ может сместиться за такт вдоль определенного направления (для 3-мерной ленты таких направлений 6), Δ-структура характеризуется алфавитом направлений Δ. Конечное множество Δ состоит из абстрактных попарно различимых символов. Помимо алфавита направлений Δ-структура обладает конечным, но растущим множеством узлов X. Смысл и назначение отдельного узла определяется общим проектом конкретной машины Шенхаге, но узел ничего не "хранит". Точнее, Δ-структура – это St = (X, a, p), где X – конечное множество узлов, a∈X, a – центр St, p = {p_α | α∈Δ} – семейство указательных отображений p_α: X→ X таких, что $p_\alpha(x) = y$ тогда и только тогда, когда из x идет стрелка в y, помеченная символом α (назовем ее α–стрелкой). Как обычно, Δ* − множество всех конечных цепочек из символов Δ, при этом ε–пустая цепочка. Отображение p^* : Δ* →X определяется рекурсивно согласно правилам: $p^*(\varepsilon) = a$, $p^*(W\alpha) = p_\alpha(p^*(W))$, где W∈Δ*, α∈Δ. Содержательно, $p^*(W) = x$ означает, что из центра a путь по стрелкам, пометки которых образуют цепочку W(назовем ее цепочкой доступа), ведет в узел x. Память машины Шенхаге на момент времени t – это совокупность тех узлов, к каждому из которых в момент t есть путь (с подходящей цепочкой доступа) из центра a.

Управление машины задается в виде алголо-подобной программы, которая может включать следующие, возможно помеченные, команды. Пусть U, V, W∈Δ*, тогда:

new W – текущая Δ-структура St = (X, a, p) изменяется путем создания нового узла y; новая Δ-структура St′ = (X′, a′, p′) такова, что X′ = X∪ {y}, путь доступа к новому узлу y определяется цепочкой W, от вида которой зависит p′;если W = ε, то a′ = y и $p'_\delta(y) = a$ для всех δ∈Δ; если W = Uα, где α∈Δ, то a′ = a, $p'_\alpha(p^*(U)) = y$ и $p'_\delta(y) = p^*(W)$ для всех δ∈Δ, все другие стрелки Δ-структуры не меняются;

set W **to** V − текущая Δ-структура St = (X, a, p) изменяется на St′ = (X′, a′, p′) путем смены центра или перенаправления одной стрелки; точнее, если W = ε, то a′ = $p^*(V)$ и X′ = {x| x∈X и существует W′∈Δ^* так, что x= $p^*(W')$, если a′ = $p^*(\varepsilon)$} и p′ = {p'_α| α∈Δ}, где p'_α= (X′×X′) ∩ p_α ; если W = Uα, где α∈Δ, то X′ = X, a′ = a, $p'_\alpha(p^*(U)) = p^*(V)$, все другие стрелки Δ-структуры не меняются;

goto λ – передача управления на команду с меткой λ;

if U = V **then** σ − если в текущей Δ-структуре $p^*(U) = p^*(V)$, то выполняется команда σ, где σ – одна из ранее рассмотренных команд; если $p^*(U) \neq p^*(V)$, то σ пропускается;

if U ≠ V **then** σ – команда σ выполняется, если $p^*(U) \neq p^*(V)$, иначе σ пропускается;

input λ_0, λ_1– читает символ β с входной ленты с последующей передачей управления на команду с меткой λ_β , где β∈{0, 1}; если входная строка исчерпана, выполняется следующая команда;

output β − β посылается на выходную ленту, где β∈{0, 1}.

halt − команда останова, который происходит и когда выполнятся все команды программы. Если **halt** соответствует нормальному и запланированному прекращению вычислений, то аварийный отказ от дальнейших вычислений могла бы обеспечить вводимая здесь команда **failure**. Одним из ее результатов является уничтожение данных на выходной ленте, т.е. неопределенность результата. Все команды завершаются точкой с запятой.

Машина стартует выполнением первой команды программы при пустой памяти, т.е. когда Δ-структура состоит из одного узла-центра, все стрелки которого ведут в него же.

II. Атаки на память

До описания возможных атак введем, по аналогии с машинами Тьюринга, композицию машин Шенхаге. Пусть M_1 и M_2 – машины Шенхаге с одинаковым алфавитом Δ. Тогда $M_1 \circ M_2$ – это такая машина Шенхаге, программа которой получается соединением программ M_1 и M_2 так, что вместо команды **halt** в коде M_1

выполняется переход на первую команду кода M_2. Метки команд программы M_2 подходящим образом модифицируются. При этом M_2 стартует на Δ-структуре, сформированной к моменту завершения M_1, и продолжает как чтение символов с входной ленты, так и печать символов на выходную ленту. С другой стороны, команда **failure** в M_1 прекращает выполнение всей композиции $M_1°M_2$. Композиция машин M_1, M_2, M_3 определяется как $(M_1°M_2)°M_3$. Для большего числа машин их композиция определяется аналогично.

Схема атаки такова. Машина Шенхаге работает по своей программе до некоторого момента t, формируя некоторую конфигурацию памяти − Δ-структуру St. После момента t запускается некая атакующая программа M, имеющая полный доступ к St и которая может вносить какие-то разрушения St. В зависимости от того, предполагал ли план запуска атакованной машины продолжение вычислений или нет, можно говорить об атаке без последействия, когда факт атаки M остается неопознанным, и завершение M не предполагает возобновления действий атакованной машины. Атака с последействием осуществляется, когда атакованная машина является композицией $M_1°M_2°…°M_i°…°M_n$ машин и $M = M_i$, $1 < i < n$. Разрушительный функционал M_i может быть следствием ее подмены, неполного знания о ней, например, M_i– троянец, или некоторого “зомбирования”. Очевидно, можно полагать, что $i = 2$. В дальнейшем рассматривается случай, когда в композиции лишь одна машина осуществляет атаку. Машины M_j, $j > i$, могут учитывать возможность атаки и строиться на принципах безопасного программирования, например, выполняя проверку целостности своей Δ-структуры [2].

Далее рассматриваются возможные варианты разрушений St при атаке без последействия, а при атаке с последействием – детекция разрушений в предположении целостности содержимого всех лент машины.

Машина Шенхаге не имеет команд, осуществляющих уничтожение узла или стрелки. Стрелки можно лишь перенаправить. Соответственно узел x становится гарантированно недоступным, если все заходящие в него стрелки перенаправить на другие узлы. Это действие можно назвать изоляцией узла x. Действие, в результате которого все исходящие из x стрелки направляются в него же, назовем терминацией x. После терминации x становится висячим узлом. Если отдельная стрелка, исходящая из узла, перенаправляется в него же, становясь петлей, то это действие назовем терминацией данной стрелки.

Терминация центра Δ-структуры St. Она обеспечивается кодом в виде произвольно упорядоченной последовательности команд **set** α **to** ε для каждого $\alpha \in \Delta$.

В этом случае фактически все узлы, кроме центра, становятся недоступными. Однако частичное восстановление St – это переход к Δ-структуре $St' = (X', a', p')$, где $a' \neq a$, $X' = \{x \mid x \in X$ и существует $W \in \Delta^*$ так, что $x = p^*(W)$, если $a' = p^*(\varepsilon)\}$ и $p' = \{p'_\alpha \mid \alpha \in \Delta\}$, где $p'_\alpha = (X' \times X') \cap p_\alpha$. При удачном выборе a' можно восстановить определенную часть St.

Терминация всех узлов Δ-структуры St. Это наиболее опасный вариант деградации заданной Δ-структуры, когда она преобразуется во вполне несвязный граф. Осуществление такой атаки рассмотрим далее для двухэлементного алфавита $\Delta = \{A, B\}$, поскольку показано, что этим случаем можно ограничиться [3]. Код, реализующий атаку, удобнее задавать более выразительными макрокомандами нежели вышеперечисленными командами машины Шенхаге. Каждую из следующих макрокоманд можно выразить в терминах базовых команд с использованием подходящих меток, условных и безусловных переходов:

1. **if** (cmp) $\{S_1\}$ **else** $\{S_2\}$, где cmp – это сравнение вида $U = V$ или $U \neq V$, используемое в командах базового набора, S_1, S_2 – последовательности из базовых команд машины Шенхаге или макрокоманды;

2. **if** (cmp){S},

3. **while**(cmp){S}, где S – последовательность из базовых команд машины Шенхаге или макрокоманды.

Используя указанные макрокоманды, определим макрокоманду, которую применим в атакующем коде. Условие isTerminal(W) означает проверку – является ли узел с цепочкой доступа $W \in \Delta^*$ терминальным. Тогда макрокоманда **if** (isTerminal(W)){S_1}**else** {S_2} выражается, например, как **if** ($W \neq WA$){S_2}**else** {**if**($W = WB$){S_1} **else** { S_2}}.

Атаки на Δ-структуру могут быть избирательными. Например, чтобы скрыть следы вторжения, атакующий код может выполнить

терминацию лишь части узлов или терминировать определенные стрелки. В этой связи рассмотрим терминацию всех узлов {A, B}-структуры, расположенных на определенной цепочке доступа W. Пусть цепочка доступа имеет вид $W = s_1s_2\ldots s_n$, где $s_i \in$ {A, B}.

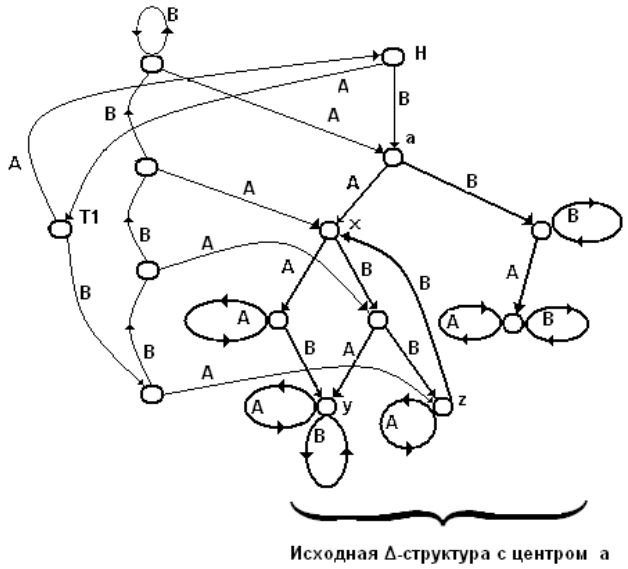

Рис. 1. Атака на память справа кодом, использующим память слева

Программа терминации узлов, расположенных на W, – TermNodesOnString использует макрокоманды: InitCntrlStruct, AccountForNode(s), TermAllNodesOnString. Она имеет вид:

TermNodesOnString = InitCntrlStruct AccountForNode(s_1) AccountForNode(s_2) … AccountForNode(s_n) TermAllNodesOnString

InitCntrlStruct = **new** ε; **new** A; **new** AB; **set** AA **to** ε; **set** ABB **to** AB; **set** ABA **to** B;

Макрокоманда InitCntrlStruct создает новый центр – узел H (см. Рис. 1). Создается узел T1, доступный из H по A-стрелке. B-стрелка узла H ссылается на центр a исходной {A, B}–структуры. Создается первый узел v вспомогательного стека для отслеживания узлов, расположенных на пути W. Этот узел доступен из T1 по B-стрелке. A-стрелка узла T1 устанавливается на новый центр – узел H. У узла v нет предшественников, поэтому его B-стрелка указывает на него же. Наконец, A-стрелка узла v ссылается на центр a исходной {A, B}–структуры.

AccountForNode(s) = **new** AB; **set** ABA **to** ABBAs;

Макрокоманда AccountForNode(s) создает новую верхушку стека – узел v′, доступный по B-стрелке из узла T1. Старая верхушка стека – узел v доступен из v′ по B-стрелке (это обеспечивается действием команды **new** AB). A-стрелка узла v′ устанавливается на тот узел исходной {A, B}–структуры, который доступен по s-стрелке из узла u, достигнутого шагом ранее на пути, определенном цепочкой доступа W. Сам узел u доступен по A-стрелке из узла v.

isNotStackBottom = ABB ≠ AB

TermAllNodesOnString = **while** (isNotStackBottom){

set ABAA **to** ABA; **set** ABAB **to** ABA; **set** ABA **to** AB; **set** B **to** AB;

set AB **to** ABB; set BB **to** B;

}

set ABAA **to** ABA; **set** ABAB **to** ABA;

set B **to** ABA; //B-стрелка узла H теперь ссылается на //центр a исходной {A, B}-структуры

set ABA **to** AB;

set BA **to** AA; //A-стрелка центра a исходной {A, B}-//структуры теперь ссылается на узел H

set AB **to** A; **set** AA **to** A; **set** A **to** ε;

set ε **to** B; //узел a снова становится центром {A, B}-//структуры

set AB **to** A; // B-стрелка узла H замыкается на H

set A **to** ε; //A-стрелка узла a замыкается на a

Макрокоманда TermAllNodesOnString выполняет проход по узлам исходной {A, B}-структуры, расположенным на цепочке доступа W, но в обратном порядке: сначала все стрелки узла $p^*(W)$ замыкаются на нем же, затем, используя B-стрелки вспомогательного стека, происходит доступ к узлу стека, ссылающегося на узел $p^*(s_1s_2\ldots s_{n-1})$. Стрелки этого узла также замыкаются на нем же. Этот процесс сопровождается замыканием стрелок каждого узла – элемента вспомогательного стека на этом же узле. После повторения указанных действий – до выборки всех узлов вспомогательного стека (а через них – терминации всех узлов, расположенных на цепочке доступа W), происходит терминация узла T1, назначение узла a вновь центром

измененной {A, B}-структуры и терминация узлов H и a. Представленная конструкция – макрокоманда TermNodesOnString, служит обоснованием следующего утверждения.

Теорема 1. Какова бы ни была цепочка доступа W в произвольной {A, B}-структуре, возможна терминация всех узлов, расположенных на W, подходящей машиной Шенхаге.

Еще одной атакой на доступность {A, B}–структуры может быть терминация всех стрелок с определенной пометкой. Это могут быть все A-стрелки или все B-стрелки. Назовем такие стрелки S-стрелками. Осуществить терминацию всех S-стрелок можно следующим алгоритмом TermAllArrows(S), комбинирующим поиск в глубину – dfs на {A, B}–структуре St с регистрацией локаций узлов, проходимых при dfs. TermAllArrows(S) использует макрокоманды: InitAccountStructs, MakeDfs, TermSetArrows(S), DeInitAccountStructs. Он имеет вид:

TermAllArrows(S) = InitAccountStructs MakeDfs TermSetArrows(S) DeInitAccountStructs(S)

Макрокоманда InitAccountStructs создает новый центр – узел H (см. Рис. 2) и узлы T1, T2, обеспечивающие доступ к двум вспомогательным стекам, используемым для регистрации пройденных согласно dfs узлов и организации самого dfs соответственно.

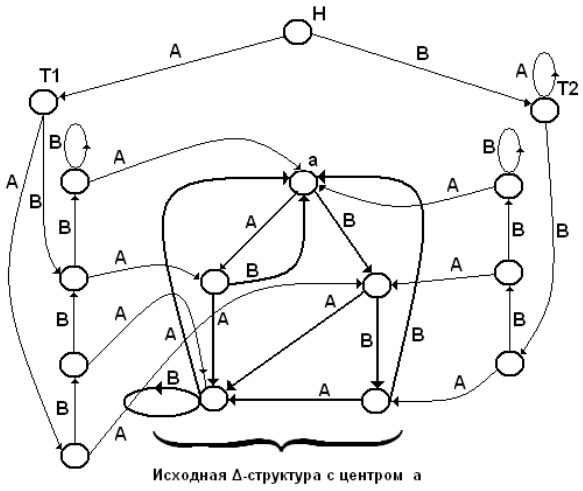

Рис. 2. Структура памяти при атаке, повреждающей определенные связи

InitAccountStructs = **new** ε; **new** A; **new** B; **new** AA; **set** AAB **to** AA; **set** AB **to** AA; **new** BB; **set** BBB **to** BB; **set** BA **to** B;

Макрокоманда MakeDfs осуществляет dfs на {A, B}-структуре St, используя список, представленный на Рис. 2 справа, в качестве стека. B-стрелка узла T2 ссылается на верхушку этого стека. A-стрелки узлов стека ссылаются на открытые, но незавершенные узлы St. Интерфейс стека – это макрокоманды PushS(X), PopS, isNotStackSBottom.

PushS(X) = **new** BA; **set** BAB **to** BB; **set** BAA **to** BBAX; **set** BB **to** BA; **set** BA **to** B;

PopS = **set** BA **to** BB; **set** BB **to** BBB; **set** BAA **to** BA; **set** BAB **to** BA; **set** BA **to** B;

isNotStackSBottom = BBB≠BB

Для регистрации локаций узлов из St, которые при dfs открываются впервые, используется список, показанный на Рис. 2 слева. Он также используется как стек, A-стрелка узла T1 которого ссылается на его верхушку, а B-стрелка того же узла используется для прохода по этому стеку без его разгрузки. Макрокоманда Add(X) добавляет к стеку новый узел, A-стрелка которого будет ссылаться на узел BBAX, если этот узел dfs открывает впервые.

Add(X) = **set** AB **to** AA; **new** AB; **set** ABA **to** BBAX; **set** AA **to** AB;

Макрокоманда Check проверяет – верно ли, что узел, выбираемый dfs для посещения, уже зарегистрирован в левом списке. Если уже зарегистрирован, то B-стрелка узла T1 замыкается на нем. В остальном, макрокоманда Check обеспечивает проход по узлам левого списка последовательной перекоммутацией B-стрелки узла T1 на эти узлы.

Check(X, L) = L: **if** (BBAX ≠ ABA){**if**(ABB ≠ AB){**set** AB **to** ABB; **goto** L;}}

else {**set** AB **to** A;}// AB = A – признак того, что узел //BBAX уже был зарегистрирован в левом списке ранее

Макрокоманда MakeDfs определяет поиск в глубину на {A, B}-структуре St, отличающийся от стандартного использованием левого списка для отметки уже посещенных узлов из St и регистрации их локации и правого списка, в котором регистрируются открытые, но незавершенные узлы из St [4].

MakeDfs = Again: **if** (isTerminal(BBA)){**goto** GoBack}

else { **if** (BBA ≠ BBAA){ Check(A, L1)

if (AB = A) {**set** AB **to** AA; **goto** Next;}

else {Add(A) PushS(A) **goto** Again;}

}

else {Next: **if** (BBA ≠ BBAB){ Check(B, L2)

if (AB = A) {**set** AB **to** AA; **goto** GoBack;}

else {Add(B) PushS(B) **goto** Again;}

}

}

}

GoBack: **if** (isNotStackSBottom) {PopS **goto** Again;}

TermSetArrows(S) = **while** (ABB ≠ AB) {

set ABAS **to** ABA; **set** AB **to** ABB;

}

set ABAS **to** ABA; **set** AB **to** AA;

Макрокоманда TermSetArrows(S) определяет проход по левому списку, последовательно перекоммутируя B-стрелку узла T1 на узлы этого списка. При этом S-стрелка очередного узла {A, B}-структуры St (он имеет цепочку доступа ABA из узла T1) замыкается.

DeInitAccountStructs(S) = **while** (AAB≠AA) {//терминация узлов левого списка

set AAA **to** AA; **set** AA **to** AAB;

set ABB **to** AB; **set** AB **to** AA;

}

set AAAS **to** ε; //S-стрелка узла a направляется на узел H

set ε **to** AAA; //центром St снова становится узел a

set SAAA **to** SAA; **set** SAA **to** SA;

set SAB **to** SA; **set** SA **to** S;

set SBA **to** SBB;

while (SBBB≠SBB) {//терминация узлов правого списка

set SBBA **to** SBB; **set** SBB **to** SBBB;

set SBAB **to** SBA; **set** SBA **to** SBB;

}

set SBBA **to** SBB; **set** SBA **to** SB; **set** SBB **to** SB;

set SB **to** S; **set** S **to** ε;

Макрокоманда DeInitAccountStructs(S) сначала в цикле выполняет терминацию всех узлов левого списка. Когда стрелки узла T1 переведутся на узел a, его S-стрелка (к тому моменту она уже замкнута на a) устанавливается на узел H. После задания узла a снова центром всей Δ-структуры, терминация узлов правого списка, узлов T1, T2 и H выполняется через S-стрелку узла a. Представленная конструкция – макрокоманда TermAllArrows(S), где S может быть A-стрелкой или B-стрелкой, служит обоснованием следующего утверждения.

Теорема 2. Какова бы ни была {A, B}-структура машины Шенхаге, стрелки с определенной пометкой всех узлов этой структуры можно терминировать подходящей машиной Шенхаге.

Следствие. Какова бы ни была исходная {A, B}-структура St машины Шенхаге, существует машина Шенхаге, преобразующая St во вполне несвязный орграф.

Можно выполнить терминацию всех узлов {A, B}-структуры, превращая ее во вполне несвязный граф, например, таким кодом:

InitAccountStructs MakeDfs TermSetArrows(A) TermSetArrows(B) DeInitAccountStructs(A)

Следствие показывает, что алгоритмически полный класс машин Шенхаге над структурой памяти с двухэлементным алфавитом может быть подвергнут атаке в произвольный момент времени в виде полной терминации структуры памяти, сформированной к этому моменту.

III. Машины Шенхаге с регулярной Δ-структурой

Существует класс машин Шенхаге, Δ-структура которых в основном формируется повторением некоторого орграфа. Например, язык $L = \{0^n10^n \mid n \in N\}$ распознается следующей машиной ML, память которой (Δ-структура) изображена на Рис. 3. Память получается итерацией одного узла с фиксированной коммутацией его стрелок. Исключение составляют стрелки из центра a, помеченные A и C, и B-стрелка последнего узла. Работа ML состоит из двух фаз. На первой фазе ML

регистрирует считанные до первой единицы нули созданием новых узлов так, что на последний такой узел ссылается A-стрелка центра a. По прочтении единицы ML начинает продвигать C-стрелку узла a, используя B-стрелки узлов. Входная последовательность отвергается, если количества нулей до и после единицы – разные, или же на входе было более одной единицы, иначе последовательность допускается.

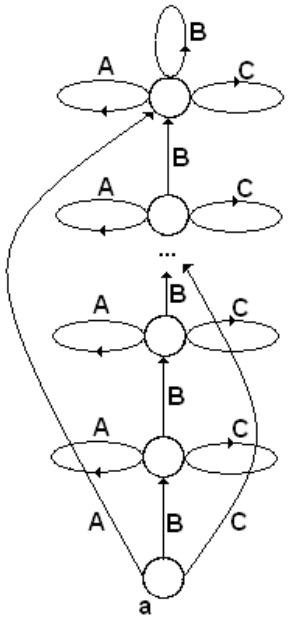

Рис. 3. Структура памяти машины ML

Код машины ML:

Phase1: **input** Z1, Phase2;

goto NoOnes;

Z1: **new** AB; **set** A **to** AB; **set** AA **to** A; **set** AC **to** A;

set AB **to** A; **goto** Phase1;

NoOnes: **output** 0; **halt**;

Phase2: **input** Z2, MoreOnes;

goto Check;

Z2: **if** C = CB **then goto** MoreOnes;

set C to CB;

goto Phase2;

MoreOnes: **output** 0; **halt**;

Check: **if** A ≠ C **then goto** MoreOnes;

output 1; **halt**;

Машинами с регулярной Δ-структурой назовем такие машины, у которых St может быть определена подходящей графовой грамматикой за исключением заранее заданного конечного числа ее стрелок, выходящих из центра. При определении такой грамматики, назовем ее стрелочной, учитывается следующее требование к разметке стрелок, выходящих из произвольного узла Δ-структуры. Все выходящие из узла стрелки метятся разными символами из Δ, нет такого символа из Δ, который бы не метил какую-то выходящую из этого узла стрелку. Назовем это требованием полноты пометки стрелок. Учитывая это, стрелки порождаемого стрелочной грамматикой орграфа, который в итоге становится регулярной Δ-структурой, будут метится терминалами, представляющими символы из Δ. При этом каждый символ a∈Δ задается в стрелочной грамматике в двух видах: терминальном, обозначим его ta, и нетерминальном, обозначим его na. Это связано с тем, что роль нетерминалов выполняют не вершины, которые обычно заменяются на графы из правых частей продукций обычной графовой грамматики, а дуги, которые в порождаемом стрелочной грамматикой графе могут быть “окончательными” (они метятся терминалами вида ta) или подлежащими дальнейшему “развертыванию” (они метятся нетерминалами вида na). “Развертывать” можно лишь дуги, являющиеся петлями.

Поскольку стрелочная грамматика предназначена для описания Δ-структур машин Шенхаге, ее определение задается в терминах этих структур. Точнее, пусть N = {na| a∈Δ} – множество нетерминальных символов, T = {ta | a∈Δ} –множество терминальных символов. Орграф D(w) понимается как частично-заданная структура памяти машин Шенхаге с центром – узлом w, т.е. D(w) = (X_D, w, p_D), где X_D – множество узлов орграфа D, $p_D = \{p_\alpha \mid \alpha \in N \cup T\}$ –семейство частично-определенных указательных отображений p_α: $X_D \rightarrow X_D$, удовлетворяющее требованию полноты пометки стрелок элементами из $N \cup T$, так что для любого узла $x \in X_D$ и любого $a \in \Delta$ значение $p_{ta}(x)$ определено ⇔ значение $p_{na}(x)$ неопределено. Стрелку p_α, помеченную $\alpha \in N \cup T$, для узла x назовем петлей, если $p_\alpha(x) = x$. При этом, для любого $a \in N$, если $p_a(x)$ определено при некотором $x \in X_D$, то $p_a(x) = x$, т.е. если стрелка в D(w) помечается нетерминалом, то она должна быть петлей.

Под стрелочной грамматикой понимается AG = (I, N, T, P), где I =A(a) – орграф, представленный как указано выше, аксиома грамматики с центром a, являющийся стартовым при порождении AG регулярных Δ-структур, P – конечное множество продукций вида nα→ D(w), где α∈Δ, D(w)– орграф с центром w. Продукция r: nα→ D(w) применима к петле h орграфа G = (X_G, v, p_G), если h∈$p_{nα}$ (т.е. в G есть петля h, помеченная нетерминалом nа для некоторого а из Δ). Результатом применения r является новый орграф G', который получается из G в результате перенаправления конца x петли h на узел w орграфа D(w), который, таким образом, добавляется к G, кроме того, пометка nα дуги h заменяется на tα, т.е. отображение $p_{tα}$ заменяется отображением $p_{tα}$ ∪ (x, w), а отображение $p_{nα}$ – отображением $p_{nα}$ \ (x, x).

Описанное преобразование G в G' путем применения продукции r определяет отношение непосредственной выводимости G' из G по продукции r. Обозначим его как G $^{r}\Rightarrow$ G'. Орграф G' непосредственно выводим из G, G ⇒ G', если в P найдется продукция r такая, что G $^{r}\Rightarrow$ G'. Транзитивное замыкание отношения ⇒ обозначим как $^{*}\Rightarrow$. Многообразием регулярных Δ-структур, определенным грамматикой AG, назовем множество D(AG) = {G| I $^{*}\Rightarrow$ G, G = (X_G, {v}, p_G) и каково бы ни было отображение $p_α$∈p_G, $p_α$: X_G→X_G, $p_α$ – всюду определено на X_G и α∈T}.

Скажем, что машина Шенхаге M имеет регулярную Δ-структуру, если существует натуральное число g и стрелочная грамматика AG, такие что:

1. для любого орграфа G ∈ D(AG) можно указать такую Δ-структуру St = (X, a, p) машины M, полученную из стартовой на некотором шаге работы M, что St, как орграф, отличается от G не более чем g стрелками, выходящими из a;
2. какова бы ни была Δ-структура St = (X, a, p) машины M, полученная из стартовой на некотором шаге работы M, существует орграф G ∈ D(AG) такой, что St может быть преобразована в Δ-структуру St′ за не более чем g команд M, продолжающих вычисление от Δ-структуры St, при этом St′, как орграф, отличается от G не более чем g стрелками, выходящими из a.

Например, совокупность Δ-структур, аналогичных данной на Рис. 3 для машины ML, описывается стрелочной грамматикой AML, показанной на Рис. 4.

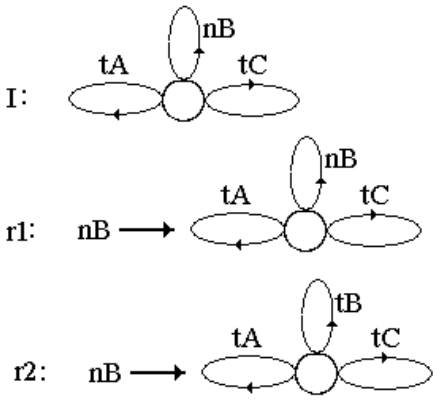

Рис. 4. Продукции стрелочной грамматики AML

Вершина, определенная в аксиоме I, является центром a в порождаемых этой грамматикой Δ-структурах, а ее стрелки, помеченные tA и tC (для упрощения, t опущено на Рис. 3), являются петлями центра a. Рост такой структуры обеспечивается благодаря "развертыванию" петли, помеченной нетерминалом nB, согласно продукциям r1 или r2.

С другой стороны, Δ-структура, назовем ее St, которую формирует машина ML после циклического выполнения блока команд с меткой Z1, имеет такой же вид что и Δ-структура St′ из D(AML) с таким же числом узлов и стрелок. Отличие между St и St′ лишь в том, что A-стрелка и C-стрелка, выходящие из a в St′ являются петлями, а такие же стрелки в St ссылаются на узлы с цепочками доступа вида B*. При этом A-стрелка из центра St ссылается на узел, имеющий цепочку доступа указанного вида максимальной длины. Таким образом, согласно определению, ML является машиной с регулярной структурой.

Машинами с регулярной Δ-структурой можно определять вычисления с фиксированной задержкой во времени. Скажем, что машина выполняет вычисление с фиксированной задержкой во времени, если существует такая константа c, что для любого момента времени t, при котором выполнялась команда **input**, команда **output** выполнится не позже, чем в момент времени t+c. Если машина M всегда выполняет **output** сразу после команды **input**, то машина M′, моделирующая M в реальное время, выполняет это моделирование с фиксированной задержкой во времени [3]. Не каждая машина с регулярной Δ-структурой работает с фиксированной задержкой, например, такова ML. Машина BC (ограниченный счетчик), приведенная ниже, работает с фиксированной задержкой. Ее регулярная Δ-структура описывается стрелочной

грамматикой, имеющей стартовый орграф, показанный на Рис. 5, и пустое множество продукций.

Пусть m (m> 1) – фиксированное натуральное число. Для удобства положим $m = 2^k$ при некотором k. Машина BC читает входную строку из единиц и по прочтении i-го символа печатает i mod m в виде k-разрядного блока. Для простоты рассмотрим k=2, поскольку в общем случае BC имеет аналогичную архитектуру. Пусть Δ = {0, 1, 2, 3, 4}. Память BC показана на Рис. 5.

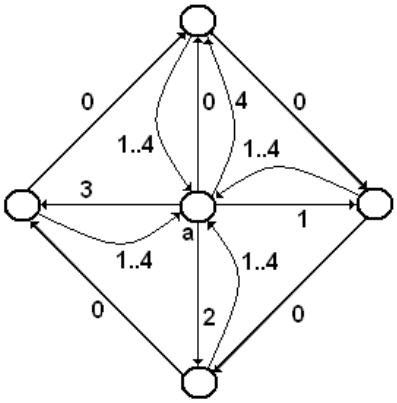

Рис. 5. Структура памяти машины BC

Узел, помеченный a, является центром Δ-структуры. Стрелка с пометкой 1..4 представляет пучок стрелок, помеченных цифрами от 1 до 4. Код машины BC:

new 0; **new** 1; **new** 2; **new** 3;

set 00 **to** 1; **set** 10 **to** 2; **set** 20 **to** 3; **set** 30 **to** 0; **set** 4 **to** 0;

Read: **input** Tick, Tick;

halt;

Tick: **set** 4 **to** 40;

if 0 = 4 **then goto** Zero;

if 1 = 4 **then goto** One;

if 2 = 4 **then goto** Two;

if 3 = 4 **then goto** Three;

Zero: **output** 0; **output** 0; **goto** Read;

One: **output** 0; **output** 1; **goto** Read;

Two: **output** 1; **output** 0; **goto** Read;

Three: **output** 1; **output** 1; **goto** Read;

Возможность атаки на Δ-структуру делает необходимой проверку ее правильности, которая выражается в соблюдении всех инвариантов и ограничений, составляющих условие регулярности Δ-структуры. Например, Δ-структура машины ML – это цепочка узлов, в которой две стрелки узла являются петлями, а одна стрелка ссылается на следующий такой же узел. A-стрелка центра всегда указывает на последний созданный узел, C-стрелка центра может указывать на любой созданный узел. Используя нотацию в стиле Лиспа, эти ограничения можно выразить рекурсивными предикатами RestrictionAB, RestrictionC:

RestrictionAB(X) = ((X≠XB) →(X=XA) & (X=XC) & RestrictionAB(XB), ((X=XA) & (X=XC) & ((A=X) ∨ (AB=X))))

RestrictionC(X) = ((X≠XB) → (C=X) ∨ RestrictionC(XB), C=X). Тогда инвариантом для Δ-структуры машины ML является RestrictionAB(B) & RestrictionC(B). Следующие макрокоманды определяют нерекурсивную проверку указанных ограничений.

NonRecRestrAB(X) = **while**(X≠XB) {

if X≠ XA **then failure**;

if X≠ XC **then failure**; **set** X **to** XB;

}

if X ≠ XA **then failure**; **if** X ≠ XC **then failure**;

if A = X **then goto** OK; **if** AB = X **then goto** OK;

failure;

OK: **set** X **to** X;

NonRecRestrC(X) = **while** (X ≠ XB) {

if C = X **then goto** OKK; **set** X **to** XB;

}

OKK: **if** C≠ X **then failure**;

Итоговая проверка инварианта для ML: NonRecRestrAB(B) **set** B **to** ε; NonRecRestrC(B)

Для Δ-структуры машины BC, используя следующие предикаты: Wheel = (00=1) & (10=2) & (20=3) & (30=0)

Hand = (4=0) ∨ (4=1) ∨ (4=2) ∨ (4=3)

toCenter(X) = (X1=X2) & (X2=X3) & (X3=X4) & (X4=ε), можно определить инвариант как Wheel & Hand & toCenter(0) & toCenter(1) & toCenter(2) & toCenter(3). Полученный инвариант нетрудно преобразовать в код, проверяющий правильность Δ-структуры ВС.

IV. Заключение

В статье представлено два сценария атаки на память одного вида указательных машин, предложенного Шенхаге. Показано, что при первом сценарии такие машины, получая доступ к памяти машины-жертвы, способны выполнять как определенные выборочные повреждения, так и полное разрушение структуры ее памяти. При втором сценарии возможна регистрация проведенной атаки посредством проверки инвариантов структуры памяти атакованной машины. Для машин с регулярной Δ-структурой они понимаются как соответствие образцам, порождаемым определенным в статье классом графовых грамматик.

Функционирование машин Шенхаге, память которых работает благодаря созданию новых узлов или управляемой перекоммутации связей между ними, может быть аналогичным работе органов или сообществ естественного или искусственного происхождения: мозг, сеть акторов, имеющих сетевую организацию [5, 6]. В этом случае изучение, классификация, нейтрализация угроз, подобных рассмотренных в статье, представляется оправданным.

Библиография

1. Landwehr C. E., Bull A. R., McDermott J. P., Choi W. S. A taxnomy of computer programs security flaws // ACM Computing Surveys. 1994. v. 26. № 3. p. 211-254.
2. Howard M., Leblanc D., Viega J. 19 deadly sins of software security: programming flaws and how to fix them. NY.: McGraw-Hill, 2005. 288 p.
3. Schönhage A. Storage modification machines // SIAM J. Computing. 1980. v. 9. №3. p. 490-508.
4. Cormen T. H., Leiserson C. E., Rivest R. L., Stein C. Introduction to algorithms, third edition. Cam.: MIT Press, 2009. 1292 p.
5. Baev K. A new conceptual understanding of brain function: basic mechnisms of brain-initiated normal and pathological behaviors // Critical reviews in neurobiology, 2007, v. 19, № 2-3, p. 119-202.
6. Agha G. Actors: a model of concurrent computation in distributed systems. Cam.: MIT Press, 1986. 190 p.

Информация об авторе: Сергей Александрович Жуков, кандидат физико-математических наук, доцент кафедры вычислительных технологий ФГБОУ ВО “Кубанский государственный университет”. Интересы: модели вычислений, модельная проверка, информационная безопасность. e-mail: szhucov@fpm.kubsu.ru, Краснодар, ул. Тюляева 18, кв. 65, тел. +79180801951.

Schönhage machines can be vulnerable

S. A. Zhucov

Abstract – The article discusses an invasion model in the form of memory destruction of one type of abstract algorithm executor introduced by Schönhage. Two invasion scenarios are defined (attack without aftereffect and attack with aftereffect), and the macros proposed in the article are used to program variants of destroying the memory of this performer. It is shown that even the limited capabilities of the Schönhage machines (the attacker) make it possible to perform both partial and complete destruction of the memory of the attacked Schönhage machine (the victim). Destruction of memory is understood as its destructuring, in which access to some of its elements or all elements becomes impossible. However, for a certain class of Schönhage machines (machines with a regular memory structure), it is possible to detect their memory in order to establish the fact of its damage. The regularity of the memory structure is understood as the possibility of describing this structure by a graph grammar, which specifies a finite set of patterns of structures, replicated when constructing the generated graph in a way defined in the grammar. Examples of Schönhage machines are given, the memory structure of which is described by the grammars noted above. Sources of threats are not considered in this article. Only some forms of threats and the possibility of their implementation by means of Schönhage machines are being studied.

Keywords – graph grammar, macro, intrusion model, depth first search, pointer machine

References

1. Landwehr C. E., Bull A. R., McDermott J. P., Choi W. S. A taxnomy of computer programs security flaws // ACM Computing Surveys. 1994. v. 26. № 3. p. 211-254.
2. Howard M., Leblanc D., Viega J. 19 deadly sins of software security: programming flaws and how to fix them. NY.: McGraw-Hill, 2005. 288 p.
3. Schönhage A. Storage modification machines // SIAM J. Computing. 1980. v. 9. №3. p. 490-508.
4. Cormen T. H., Leiserson C. E., Rivest R. L., Stein C. Introduction to algorithms, third edition. Cam.: MIT Press, 2009. 1292 p.
5. Baev K. A new conceptual understanding of brain function: basic mechnisms of brain-initiated normal and pathological behaviors // Critical reviews in neurobiology, 2007, v. 19, № 2-3, p. 119-202.
6. Agha G. Actors: a model of concurrent computation in distributed systems. Cam.: MIT Press, 1986. 190 p.