2009 Математические основы информатики и программирования №3(5)

DOI 10.17223/20710410/5/14

УДК 004.94

МЕТОД АВТОМАТИЗИРОВАННОГО ПОИСКА ПРОГРАММНЫХ ОШИБОК В АЛГОРИТМАХ ОБРАБОТКИ СЛОЖНОСТРУКТУРИРОВАННЫХ ДАННЫХ[1]

А. Н. Макаров

ФГНУ «ГНТЦ „Наука“», г. Москва, Россия

E-mail: byalex@mail.ru

Рассмотрен разработанный автором метод поиска программных ошибок в программном обеспечении при отсутствии исходных текстов. Метод основан на стрессовом тестировании совместно с автоматической трассировкой программного обеспечения. При этом наборы тестовых данных могут формироваться статически или динамически на основе результатов трассировки.

Ключевые слова: *компьютерная безопасность, тестирование, программная ошибка.*

Введение

При разработке программных комплексов часто решается задача по интеграции в них программных модулей, для которых отсутствуют исходные коды и сопроводительная техническая документация. Для обеспечения надежности функционирования комплекса в целом может потребоваться выполнить анализ бинарного кода используемых модулей.

В данном случае под анализом бинарного кода подразумевается проверка корректности работы программного обеспечения (ПО) и отсутствия программных ошибок. Существуют различные классификации ошибок, встречающихся в ПО [1, 2]. Далее считаем, что *программная ошибка* — это ошибка реализации ПО, допущенная разработчиками на этапе кодирования. Проявлением программной ошибки является аварийное завершение процесса, связанного с соответствующим ПО.

Применяют несколько основных подходов, которым отдается предпочтение при решении задачи тестирования бинарного кода [3, 4]:

— обратная инженерия (*reverse engineering*) — применяется с целью получения ПО на языке ассемблера или на языке высокого уровня;
— анализ двоичного кода — предполагает наличие анализирующего приложения, которое читает собранное ПО и просматривает его с применением некоторых эвристических правил;
— тестирование нагрузкой или стрессовое тестирование — используется набор файлов сценариев, которые посылают ПО разнообразные входные данные различного размера и структуры.

В последнее время получил распространение один из видов стрессового тестирования — фазинг (*fuzzing*) [1, 5, 6]. Его преимущество — возможность быстрого получения результатов. Выделяют два вида фазинга — мутационный (*mutation fuzzing*) и генерационный (*generation fuzzing*). В мутационном фазинге предполагается внесение

[1] Результаты работы докладывались на Международной конференции с элементами научной школы для молодёжи, г. Омск, 7–12 сентября 2009 г.

искажений в обрабатываемые входные данные, тогда как генерационный фазинг предполагает создание входных данных на основе определенных правил.

На основе мутационного фазинга автором был разработан метод автоматизированного поиска программных ошибок (АППО), обладающий следующими свойствами:

— метод ориентирован на применение в условиях отсутствия исходного кода исследуемого ПО;
— метод предназначен для выявления ошибок кодирования («программистских» ошибок) ПО, которые, как правило, трудно выявить, основываясь на формальных методах верификации ПО;
— метод является динамическим (анализ исследуемого ПО проводится во время его работы, т. е. исследованию подвергается не само ПО, а процесс, порожденный ПО);
— метод применим для тестирования широкого спектра ПО, например ПО или программных компонент, обрабатывающих файлы с нетривиальной внутренней структурой;
— метод не ориентирован на применение в рамках какой-либо конкретной аппаратно-программной платформы.

Предлагаемый метод представляет сложноструктурированные данные в виде приведенной ниже модели структур сущностей-данных.

1. Модель структур сущностей-данных

Опишем модель структур сущностей-данных, которую используем при описании метода АППО. Дадим определение.

Определение 1. ПО, для которого проводится анализ корректности реализации с помощью метода АППО, будем называть исследуемым ПО.

Сущности-данные, обрабатываемые исследуемым ПО, как правило, имеют внутреннюю структуру (структуру данных). При этом даже в случае, когда ПО обрабатывает явно неструктурированные данные (поток данных), можно считать, что данные имеют структуру вида последовательности байт (бит или слов). С использованием [7] дадим определения и сделаем предположение.

Определение 2. *Структура* — иерархическая организация взаимосвязанных данных, обладающая свойствами, которые позволяют выделить структуру среди прочих и отнести ее к определенному типу. Структуры, относящиеся к одному типу, по определению обладают одинаковой организацией взаимосвязи данных.

Определение 3. *Элементарная структура* — один байт. По определению каждая структура, кроме элементарной, в соответствии с ее иерархической организацией состоит из вложенных в нее структур. Значение каждой структуры представляется последовательностью (строкой) байт. При этом длину данной последовательности назовем размером структуры.

Определение 4. *Тип структуры* — атрибут структуры, однозначно задающий множество значений, которые могут принимать структуры данного типа, и набор операций, выполнимых над ними.

Определение 5. *Обработчик структуры* — совокупность кода исследуемого ПО, осуществляющего обработку структуры. Обработчик однозначно идентифицирует

структуру. Структуру назовем *известной* в случае, когда известен ее обработчик (полностью описаны алгоритмы, реализованные в исследуемом ПО для обработки данной структуры).

Замечание 1. Если у двух структур одинаковый обработчик, то эти структуры имеют одинаковый тип. Две однотипные структуры могут иметь различные обработчики, и, следовательно, структуры следует считать различными.

Предположение 1. Каждой сущности-данным, обрабатываемой исследуемым ПО, соответствует некоторая структура.

Используем следующие обозначения:

Ω — множество структур, обрабатываемых исследуемым ПО;

Ω^* — множество всех конечных последовательностей структур из множества Ω;

$\Omega_{\text{know}} \subset \Omega$ — множество известных структур;

T — множество типов структур;

$byte \in T$ — тип элементарной структуры;

$\Lambda \notin T$ — специальный символ, означающий, что тип структуры не может быть определен (исследуемое ПО может обрабатывать сущности-данные, содержащие структуры, определение типов которых затруднено);

$\Omega_{\text{type}} \subset \Omega$ — множество структур, тип которых определен;

Θ — множество обработчиков структур в исследуемом ПО (точное описание элементов данного множества не является существенным; потенциально оно может быть получено с использованием методов обратной инженерии);

$\Delta \notin \Theta$ — специальный символ, означающий, что обработчик структуры неизвестен (отсутствует полное описание алгоритмов, реализованных в исследуемом ПО для обработки некоторой структуры);

$B = \{0, \ldots, 255\}$ — множество значений элементарной структуры;

$B^n = \{(b_1, \ldots, b_n)\}$ — множество всех последовательностей длины $n \geqslant 1$, где $b_i \in B$;

$B^* = \bigcup_{n \geqslant 1} B^n$ — множество всех последовательностей конечной длины из элементов в B.

Определение 6. Определим $\lambda : \Omega \to \mathbb{N}$ — функцию, задающую размер структур, где $\mathbb{N}$ — множество натуральных чисел, и $\upsilon : \Omega \to B^*$ — функцию, задающую значение структур. При этом по определению каждое значение структуры $\alpha \in \Omega$ задается последовательностью из множества $B^{\lambda(\alpha)}$.

Определение 7. Определим $\tau : \Omega \to T \cup \{\Lambda\}$ — функцию, задающую типы структур, $\omega : \Omega \to \Theta \cup \{\Delta\}$ — функцию, задающую для каждой структуры ее обработчик; $\sigma : \Theta \to \mathbb{N}$ — функцию, задающую для каждого обработчика объем кода (в тысячах строк кода, ТСК). По определению выполняются следующие условия:

— для двух структур $\alpha_1, \alpha_2 \in \Omega$ справедливо равенство $\alpha_1 = \alpha_2$ тогда и только тогда, когда справедливо равенство $\omega(\alpha_1) = \omega(\alpha_2)$;
— структура $\alpha \in \Omega$ является известной ($\alpha \in \Omega_{\text{know}}$) тогда и только тогда, когда выполняется условие $\omega(\alpha) \neq \Delta$;
— если у структуры $\alpha \in \Omega$ не определен тип ($\tau(\alpha) = \Lambda$), то она является неизвестной ($\alpha \notin \Omega_{\text{know}}$, $\omega(\alpha) = \Delta$);
— если у структуры $\alpha \in \Omega$ определен тип ($\alpha \in \Omega_{\text{type}}$), то выполняется условие $\tau(\alpha) \neq \Lambda$;
— у известных структур определен их тип; таким образом, выполняется условие $\Omega_{\text{know}} \subset \Omega_{\text{type}}$.

Определение 8. Зададим на множестве типов структур T отношение частичного порядка «$\leqslant$» (отношение «быть подтипом»). При этом тип $t_1 \in T$ является подтипом типа $t_2 \in T$ ($t_1 \leqslant t_2$) тогда и только тогда, когда структура типа t_2 содержит структуру типа t_1. Если $t_1 \leqslant t_2$ и $t_1 \neq t_2$, то будем записывать $t_1 < t_2$.

Определение 9. *Множество базовых типов* $T_0 \subset T$ — подмножество множества типов T, такое, что выполняется условие $byte \in T_0$ и для любого $t \in T_0$ не существует типа $s \in T \backslash \{byte\}$, такого, что справедливо неравенство $s < t$. Зададим *множество производных типов* $T_1 = T \backslash T_0$.

Замечание 2. Примерами базовых типов могут являться тип «строка» или тип «числовая переменная».

Определение 10. Будем говорить, что структура $\alpha \in \Omega$ является структурой с *базовым типом*, если выполняется условие $\tau(\alpha) \in T_0$. Будем говорить, что структура α является структурой с *производным типом*, если выполняется условие $\tau(\alpha) \in T_1$.

Определение 11. Определим $H_T : T \to 2^T$ — функцию иерархии типов, сопоставляющую каждому типу $t \in T \backslash \{byte\}$ множество типов $H_T(t) \subset T$ и удовлетворяющую условию: если $r \in H_T(t)$, то $r < t$ и не существует типа $s \in T$, такого, что $r < s < t$. Положим $H_T(byte) = \varnothing$.

Определение 12. Зададим на множестве структур Ω отношение частичного порядка «$\leqslant$» (отношение «быть подструктурой»). При этом структура $\alpha \in \Omega$ является подструктурой $\beta \in \Omega$ ($\alpha \leqslant \beta$) тогда и только тогда, когда структура β содержит структуру α. Если $\alpha \leqslant \beta$ и $\alpha \neq \beta$, то будем записывать $\alpha < \beta$. По определению выполняется условие: если $\alpha \leqslant \beta$, то $\tau(\alpha) \leqslant \tau(\beta)$ и $\lambda(\alpha) \leqslant \lambda(\beta)$.

Замечание 3. Возможны случаи, когда две различные структуры $\alpha_1, \alpha_2 \in \Omega$, $\alpha_1 \neq \alpha_2$, имеют одинаковый тип ($\tau(\alpha_1) = \tau(\alpha_2)$), но при этом их обработчики различны ($\omega(\alpha_1) \neq \omega(\alpha_2)$). Например, это возможно, когда структура — строка α_1, входящая в состав структуры «массив строк», имеет обработчик, отличный от обработчика структуры — строки α_2, входящей в состав структуры «список строк». Таким образом, для рассмотренных структур выполняются условия $\tau(\alpha_1) = \tau(\alpha_2)$ и $\omega(\alpha_1) \neq \omega(\alpha_2)$. При этом любая из этих структур может являться либо известной, либо неизвестной.

Определение 13. Определим $H_\Omega : \Omega \to \Omega^*$ — функцию иерархии структур, сопоставляющую каждой структуре $\alpha \in \Omega$, такой, что $\tau(\alpha) \neq byte$, конечную последовательность структур $H_\Omega(\alpha) = (\alpha_1, \ldots, \alpha_m) \in \Omega^m$, $m \geqslant 1$, удовлетворяющих условиям: $\alpha_i < \alpha$ и не существует структуры $\beta \in \Omega$, такой, что $\alpha_i < \beta < \alpha$ ($i = 1, \ldots, m$). Если $\tau(\alpha) = byte$, то $H_\Omega(\alpha)$ — пустая последовательность (длины 0).

Замечание 4. Если для структуры $\alpha \in \Omega$ справедливо равенство $H_\Omega(\alpha) = (\alpha_1, \ldots, \alpha_m)$, то из определений 8, 11, 12 и 13 следует, что справедливо равенство $H_T(\alpha) = \{\tau(\alpha_1), \ldots, \tau(\alpha_m)\}$.

Замечание 5. Возможен случай, когда типы и значения структуры $\alpha \in \Omega$ и входящих в нее структур не определены. Тогда можно считать, что $\tau(\alpha) = \Lambda$ и $H_\Omega(\alpha) = (\alpha_1, \ldots, \alpha_{\lambda(\alpha)})$, где для $1 \leqslant i \leqslant \lambda(\alpha)$ справедливо равенство $\tau(\alpha_i) = byte$ (структура α состоит из последовательности элементарных структур, длина которой равна размеру структуры α). В то же время возможен случай, когда тип структуры α

не определен ($\tau(\alpha) = \Lambda$) и она состоит из последовательности структур известных типов, не совпадающих с элементарным. Например, структура α может состоять только из структур — строк и структур — числовых констант. При этом в совокупности назначение или взаимосвязь входящих в структуру α структур будут оставаться неизвестными и, следовательно, тип структуры α будет не определен. Другой случай: тип структуры α определен ($\alpha \in \Omega_{\text{type}}$), а множество $H_\Omega(\alpha)$ неизвестно. Например, структура α представляет собой *doc*-файл, тогда как внутренние структуры, составляющие *doc*-файл, неизвестны.

Замечание 6. Так как на множестве структур и на множестве типов структур заданы отношения частичного порядка, то для наглядности представления иерархии структур или иерархии типов структур можно использовать ориентированный граф. При этом вершинами графа являются структуры или их типы, а ребра соответствуют отношению частичного порядка между вершинами.

Часто при исследовании структур данных не анализируются форматы и порядок их представления непосредственно в сущностях-данных (файлах, потоках данных), обрабатываемых исследуемым ПО. Например, как правило, предполагается, что данные структур (последовательности соответствующих им байт) не могут перемешиваться. В рассматриваемом методе АППО осуществляется поиск программных ошибок исследуемого ПО, являющихся следствием, в том числе некорректной обработки ПО структур в сущностях-данных. Значит, целесообразно осуществить анализ структур с учетом возможного их представления в сущностях-данных. Дадим определение.

Определение 14. Пусть структуры $\alpha, \beta \in \Omega$ такие, что $\alpha < \beta$ (по определению 12 выполняется условие $\lambda(\alpha) \leqslant \lambda(\beta)$). Представлением структуры α в структуре β назовем подпоследовательность последовательности $\upsilon(\beta)$, соответствующую значению структуры $\upsilon(\alpha)$. При этом по определению задана инъективная функция $\psi : \{1, \ldots, \lambda(\alpha)\} \to \{1, \ldots, \lambda(\beta)\}$ (которую назовем функцией отображения структуры α в структуру β), такая, что для любых значений структур $\upsilon(\alpha) = (a_1, \ldots, a_{\lambda(\alpha)})$ и $\upsilon(\beta) = (b_1, \ldots, b_{\lambda(\beta)})$ справедливы равенства $a_i = b_{\psi(i)}$, где $1 \leqslant i \leqslant \lambda(\alpha)$.

Замечание 7. В соответствии с определением 14 для структур $\alpha, \beta \in \Omega$, таких, что $\alpha < \beta$, элементы последовательности $\upsilon(\beta)$, соответствующие представлению структуры α в структуре β, могут следовать в произвольном порядке и между ними могут содержаться элементы, не принадлежащие значению $\upsilon(\alpha)$ структуры α.

В ПО, как правило, структуры базовых типов являются наиболее распространенными. В связи с этим производители ПО стремятся упростить представление этих структур в сущностях-данных, что упрощает разработку обработчиков структур базовых типов. Таким образом, целесообразно использовать следующее предположение.

Предположение 2. Пусть структуры $\alpha, \beta \in \Omega$ такие, что $\alpha < \beta$ и структура α имеет базовый тип ($\tau(\alpha) \in T_0$). Тогда функция ψ отображения структуры α в структуру β обладает следующим свойством: для $0 \leqslant i < \lambda(\alpha)$ выполняется условие $\psi(i+1) = \psi(i) + 1 = \psi(1) + i$.

Таким образом, описана модель структур, содержащихся в сущностях-данных, обрабатываемых исследуемым ПО.

2. Модель процесса исследуемого ПО

Рассматриваемый метод АППО является динамическим [8], то есть он ориентирован на анализ программных ошибок в исследуемом ПО, когда оно является активным

(выполняется как процесс ОС). В связи с этим опишем модель процесса ОС, которую используем при описании метода АППО. Дадим определения.

Определение 15. Процессом ОС, реализующим исследуемое ПО (процессом исследуемого ПО), назовем автомат $P{<}S, I, p, s_0{>}$, где S — множество всех состояний процесса; $I \subset \Omega^*$ — множество входных данных; $p : I \times S \to S$ — функция переходов процесса из состояния в состояние; $s_0 \in S$ — начальное состояние процесса.

Замечание 8. Функция перехода p задается последовательностью исполняемых инструкций кода исследуемого ПО, которое реализует процесс P.

Определение 16. Для каждого процесса исследуемого ПО $P{<}S, I, p, s_0{>}$ на множестве состояний S зададим множество *аварийных состояний* $E \subset S \backslash \{s_0\}$. При этом по определению для каждого аварийного состояния $e \in E$ процесса P выполняется условие: для любых входных данных $i \in I$ справедливо равенство $p(i, e) = e$ (процесс, перейдя в аварийное состояние, не может перейти ни в какое другое состояние).

Если процесс попадает в аварийное состояние, то будем говорить, что процесс P *завершился аварийно*.

Замечание 9. Как правило, для процесса любого исследуемого ПО множество аварийных состояний E можно считать всегда известным, так как оно задается средой ОС, в которой функционирует процесс. Процесс аварийно завершается тогда и только тогда, когда выполнение очередной машинной инструкции «физически» невозможно (например, нет доступа к памяти или процессор не может декодировать машинную команду) [9].

Определение 17. *Эталонные входные данные* — входные данные, которые, как правило, созданы процессом исследуемого ПО, и их обработка по определению не приводит к его аварийному завершению. Если эталонные входные данные являются файлом, который подается на вход процессу исследуемого ПО, то назовем его *эталонным входным файлом*.

Определение 18. Пусть $P{<}S, I, p, s_0{>}$ — процесс исследуемого ПО. Будем говорить, что в процессе P содержится программная ошибка, если существуют входные данные $i \in I$, такие, что справедливо условие $p(i, s_0) \in E$. *Программная ошибка* — это ошибка в коде исследуемого ПО (как правило, допущенная его разработчиками), которая приводит к аварийному завершению процесса исследуемого ПО на соответствующих входных данных.

Используем обозначения:

$I_S \subset I$ — множество эталонных входных данных для процесса $P{<}S, I, p, s_0{>}$, при этом по определению 17 для любых эталонных входных данных $i \in I_S$ и любого состояния $s \in S \backslash E$ выполняется условие $p(i, s) \notin E$;

$I_F \subset I$ — множество входных данных, при обработке которых происходит аварийное завершение процесса $P{<}S, I, p, s_0{>}$, при этом по определениям 17 и 18 справедливо равенство $I_F \cap I_S = \varnothing$;

$(e, i_1, \ldots, i_n)$ — траектория программной ошибки процесса $P{<}S, I, p, s_0{>}$ исследуемого ПО; при этом $e \in E$ и $(i_1, \ldots, i_n)$ — последовательность входных данных длины $n \geqslant 1$, подаваемых на вход процесса $P{<}S, I, p, s_0{>}$, где для $1 \leqslant k \leqslant n$ выполняется условие $i_k \in I$ и справедливо равенство $p(i_n, p(i_{n-1}, \ldots p(i_2, p(i_1, s_0)) \ldots)) = e$;

F — множество всех траекторий программных ошибок процесса $P{<}S, I, p, s_0{>}$ исследуемого ПО.

В дальнейшем целесообразно использовать следующее предположение.

Предположение 3. В любом исследуемом ПО существуют программные ошибки.

В рамках предположения 3 для процесса $P{<}S, I, p, s_0{>}$ любого исследуемого ПО существуют входные данные $(i_1, \ldots, i_n) \in I$, где $n \geqslant 1$, такие, что справедливо равенство $p((i_1, \ldots, i_n), s_0) \in E$ (при обработке входных данных $(i_1, \ldots, i_n)$ процесс $P{<}S, I, p, s_0{>}$ завершается аварийно). При этом $(p((i_1, \ldots, i_n), s_0), i_1, \ldots, i_n) \in F$ является траекторией программной ошибки процесса $P{<}S, I, p, s_0{>}$.

Таким образом, описана модель процесса исследуемого ПО.

3. Метод автоматизированного поиска программных ошибок

Для применения метода АППО разработано программное средство (ПС), реализующее следующие функции:

— формирование входных данных для исследуемого ПО;
— проведение тестов (активизация исследуемого ПО — запуски процесса исследуемого ПО и передача ему подготовленных входных данных);
— регистрация программных ошибок, возникших в результате обработки входных данных процессом исследуемого ПО.

Опишем метод АППО.

Метод АППО. Пусть дано исследуемое ПО. Применение метода АППО состоит в выполнении следующих шести этапов.

Этап 1. Проводится анализ доступной информации об исследуемом ПО, в том числе осуществляется анализ:

— доступной документации;
— доступных спецификаций структур данных, обрабатываемых исследуемым ПО;
— доступных фрагментов исходных кодов модулей исследуемого ПО или ОС (т. е. в среде, где активизируется исследуемое ПО), которые реализуют обработчики структур, содержащихся во входных данных исследуемого ПО.

В результате выполнения этапа 1 для исследуемого ПО:

— задаются множества структур Ω, известных структур Ω_{know}, типов структур T, структур известных типов Ω_{type} и обработчиков структур Θ, функции λ, υ, τ, ω, H_T и H_Ω;
— описывается, как в среде ПС исследуемое ПО может быть активизировано, т. е. запущен процесс $P{<}S, I, p, s_0{>}$ исследуемого ПО, и как этому процессу могут передаваться входные данные (множество $I \subset \Omega^*$); описывается множество входных данных I и их вид (файл, сетевые пакеты или их последовательность, последовательность событий нажатия клавиш на клавиатуре).

Этап 2. На основе результатов выполнения этапа 1 выбираются структуры, для которых является *целесообразным* восстановление и анализ кода их обработчиков. Выбор структур может быть осуществлен с учетом следующих факторов:

— наличия ресурсов для проведения детального анализа (дизассемблирование, отладка, тестирование, эксперименты на макете) кода обработчиков структур в исследуемом ПО;
— предположений о наличии программных ошибок в обработчиках выбранных структур;
— предположений о возможности восстановления кода обработчиков структур за доступное время.

Для выбранных структур осуществляется восстановление и анализ кода их обработчиков, в результате по определению 5 данные структуры становятся известными. Таким образом, уточняются значения множеств Ω, Ω_{know}, T, Ω_{type} и Θ, функций λ, υ, τ, ω, H_T и H_Ω.

Этап 3. На основе результатов выполнения этапов 1 и 2 выбираются неизвестные структуры (структуры $\alpha \notin \Omega_{\text{know}}$), для которых определен тип (структуры из множества Ω_{type}) или определение их типа является целесообразным. Выбор структур может быть осуществлен с учетом следующих факторов:

— наличия ресурсов для проведения детального анализа и описания структур;
— предположений о наличии программных ошибок в обработчиках структур выбранных типов;
— предположений о возможности описания структур за доступное время.

Для выбранных структур осуществляется определение их типа. Таким образом, уточняются значения множеств T и Ω_{type}, функций λ, υ, τ, H_T и H_Ω.

Этап 4. ПС осуществляет автоматизированный поиск программных ошибок. Поиск заключается в том, что в среде ПС запускается процесс $P{<}S, I, p, s_0{>}$ исследуемого ПО (каждый запуск процесса назовем тестом). При выполнении одного теста процессу $P{<}S, I, p, s_0{>}$ передаются входные данные, сформированные разработанными процедурами формирования входных данных (ПФВД; более подробное описание ПФВД рассмотрено далее) из эталонных входных данных путем модификации значения структур из множества Ω_{type} (множества структур, тип которых определен). Если в результате обработки входных данных процесс $P{<}S, I, p, s_0{>}$ завершился аварийно (найдена программная ошибка), то управление возвращается ПС, которое фиксирует состояние процесса $P{<}S, I, p, s_0{>}$ и входные данные, обработка которых стала причиной его аварийного завершения. При этом ПС восстанавливает корректность процесса $P{<}S, I, p, s_0{>}$ для дальнейшего тестирования.

Этап 5. На основе результатов выполнения этапов 1, 2 и 3 выбираются структуры, тип которых не определен и которые целесообразно считать элементарными (считать, что тип таких структур равен *byte*). Выбор структур может быть осуществлен с учетом следующих факторов:

— предположений о наличии программных ошибок в обработчиках данных структур;
— наличия ресурсов для проведения за доступное время тестов исследуемого ПО при обработке входных данных, в которых модифицируются значения данных структур.

Задается Ω_{byte} — множество структур, выбранных на этапе 5.

Этап 6. ПС осуществляет тестирование исследуемого ПО. При выполнении одного теста процессу $P{<}S, I, p, s_0{>}$ передаются входные данные, сформированные ПФВД из эталонных входных данных путем модификации значения структур из множества Ω_{byte} (множества структур, которые целесообразно считать элементарными). Если в результате обработки входных данных процесс $P{<}S, I, p, s_0{>}$ завершился аварийно (найдена программная ошибка), то управление возвращается ПС, которое фиксирует состояние процесса $P{<}S, I, p, s_0{>}$ и входные данные, обработка которых стала причиной его аварийного завершения. При этом ПС восстанавливает корректность процесса $P{<}S, I, p, s_0{>}$ для дальнейшего тестирования.

Прокомментируем метод АППО.

Замечание 10. Этап 1 метода АППО состоит в предварительном исследовании ПО, и время, требуемое для его выполнения, целесообразно не учитывать при оценке

результативности метода АППО. Кроме того, выбор структур, которые целесообразно считать элементарными, на этапе 5 не требует затрат времени, так как может быть осуществлен, например, случайно из неизвестных структур входных данных, тип которых не определен.

Замечание 11. Этап 2 предполагает поиск программных ошибок в сложноструктурированных данных на основе «классического» подхода (дизассемблирование и отладка). При выполнении второго этапа следует соблюдать компромисс. С одной стороны, знание всех исходных кодов обработчиков структур позволит решить задачу более эффективно (например, на основе средств анализа исходных кодов). С другой стороны, временные затраты, требуемые для восстановления исходных кодов, могут оказаться неприемлемыми. Отметим, что этап 2 может проводиться параллельно с этапами 3–6.

Можно также предложить, что оценка результативности метода АППО должна учитывать следующие параметры:

— среднее время обнаружения программной ошибки;
— средний объем кода обработчиков структур;
— среднее число программных ошибок, выявленных на этапах 2, 4 и 6;
— среднее время проведения одного теста на этапах 4 и 6;
— относительное число аварийных завершений, являющееся усредненным отношением числа аварийно завершившихся тестов к общему числу проведенных тестов на этапах 4 и 6.

В настоящий момент для эмпирической оценки результативности метода используется относительное число аварийных завершений (обозначим его через K). Опыт применения описанного метода показал, что при значении K около 0,1% (на 1000 тестов один тест завершается аварийно) и выше результативность метода для данного ПО можно признать удовлетворительной. Поскольку время проведения одного теста, как правило, занимает не более 10 с (на типовом компьютере), то при значении K порядка 0,1% и выше программные ошибки будут найдены за приемлемое время. Эксперименты показали, что существует ПО, для которого значение K достигает 5–6%.

Замечание 12. После того как программная ошибка будет найдена, наиболее разумно связаться с разработчиками программных модулей и сообщить об ошибке. К сожалению, это не всегда возможно. Также следует учесть, что если информация о программной ошибке будет принята разработчиками к сведению, то до внесения соответствующих исправлений может пройти достаточно большой срок. Если проявления программной ошибки не допустимы, то одним из возможных решений является организация дополнительной фильтрации входных данных. Как минимум, дополнительная фильтрация будет блокировать входные данные, для которых уже были выявлены программные ошибки в результате применения метода АППО.

В заключение опишем назначение ПФВД в алгоритме АППО.

4. Процедуры формирования входных данных

Применение метода выявило существенную зависимость между параметром K и применяемыми при тестировании на 4 и 6 этапах ПФВД. Были разработаны общие ПФВД (насколько это возможно), применимые к большинству исследуемого ПО. Тем не менее для повышения значения параметра K при тестировании конкретного ПО требуется разработка (или модификация существующих) индивидуальных ПФВД.

ПФВД должны сформировать входные данные, которые процессом исследуемого ПО будут приняты к обработке.

На этапах 4 и 6 метода АППО генерируются наборы входных данных, на которых запускается процесс исследуемого ПО $P{<}S, I, p, s_0{>}$. Входные данные генерируются ПФВД из эталонных входных данных путем модификации значения структур из множеств Ω_{type} и Ω_{byte}.

Используются ПФВД двух видов:

— статические ПФВД, основанные на различных эвристических правилах, которые применяются для преобразования структур без учета особенностей их обработки процессом исследуемого ПО;
— динамические ПФВД, преобразующие эталонные входные данные по правилам, которые формируются на основе трассировки процесса ПО.

Для статических ПФВД задаются различные эвристические правила преобразования структур из множеств Ω_{type} и Ω_{byte} (этапы 4 и 6). Если структуры, содержащиеся в множествах Ω_{type} и Ω_{byte}, будут найдены во входных эталонных данных, то они будут преобразованы ПФВД.

Для повышения результативности метода было решено сочетать стрессовое тестирование и динамический анализ на основе трассировки процесса исследуемого ПО. Непосредственно перед выполнением этапов 4 и 6 для каждого эталонного файла выполняются следующие действия:

— собирается трасса процесса исследуемого ПО на эталонном входном файле;
— уточняются значения множеств T, Ω_{type} и Ω_{byte}, функций λ, υ, τ, H_T и H_Ω;
— выбирается статическая ПФВД, наиболее подходящая для проведения очередного теста.

На основе трассы процесса исследуемого ПО в ПС, реализующем метод, введены вспомогательные механизмы — это механизм оценки покрытия программного кода, задействованного при обработке входных данных, и механизм определения обработчиков структур на основе сравнения трасс.

Заключение

Разработанный автором метод применяется для анализа надежности ПО в ОС семейства *Windows* 2003/*XP*/*Vista*, а также частично опробован в ОС *Windows CE* и ОС *Linux*. Эксперименты показали хорошие результаты для ПО, обрабатывающего сложноструктурированные форматы файлов.

Реализованное на основе метода ПС работает в автоматическом режиме, не требует существенных усилий со стороны оператора ПС. В качестве результатов работы ПС предоставляет множество сигнатур для организации дополнительной фильтрации входных данных, вызывающих аварийное завершение в ПО, доступ к исходным кодам которого отсутствует. Таким образом, может быть повышена надежность функционирования разрабатываемых программных комплексов.

В настоящий момент ведутся работы по исследованию и оценке параметров результативности метода с целью анализа его возможностей и расширения границ применения.

ЛИТЕРАТУРА

1. *Козиол Д., Личфилд Д., Эйтэл Д. и др.* Искусство взлома и защиты системы. СПб.: Питер, 2006. 416 с.

2. *Ховард М., Лебланк Д.* Защищенный код. 2-е изд. М.: Издательско-торговый дом «Русская Редакция», 2005. 704 с.
3. *Хогланд Г., Мак-Гроу Г.* Взлом программного обеспечения: анализ и использование кода. М.: Издательский дом «Вильямс», 2005. 400 с.
4. *Eilam E.* Reversing: Secrets of Reverse Engineering. Wiley Publishing, 2005. 589 p.
5. *Miller C., Peterson Z. N. J.* Analysis of Mutation and Generation-Based Fuzzing — `securityevaluators.com/files/papers/analysisfuzzing.pdf` — 2007.
6. *Neystadt J.* Automated Penetration Testing with White-Box Fuzzing — `msdn.microsoft.com/en-us/library/cc162782.aspx` — Microsoft Corporation, 2008.
7. *Левитин А. В.* Алгоритмы: введение в разработку и анализ. М.: Издательский дом «Вильямс», 2006. 576 с.
8. *Калбертсон Р., Браун К., Кобб Г.* Быстрое тестирование. М.: Издательский дом «Вильямс», 2002. 384 с.
9. Intel Architecture Software Developer's Manual. V. 3: System Programming. Intel, 1999.