ПОДХОД К ПРОВЕДЕНИЮ ДИНАМИЧЕСКОГО АНАЛИЗА ИСХОДНЫХ ТЕКСТОВ ПРОГРАММ

Мельников П.В.[1], Горюнов М.Н.[2], Анисимов Д.В.[3]

В статье представлен обзор подхода, проводимого с целью анализа и сопоставления декларированных и реализованных маршрутов выполнения функциональных объектов в исходных текстах программ с помощью штатных средств операционных систем семейства Linux. Рассматривается проблема корректной вставки датчиков и получения необходимой отладочной информации при проведении анализа сложных и объемных проектов. В рамках ее разрешения предложено применение механизма регулярных выражений и динамической трассировки SystemTap. Преимуществом представленного подхода в части использования регулярных выражений является открытость и простота модификации исходного кода вставки датчиков, что позволяет эксперту оперативно корректировать его под конкретные условия проведения анализа. Технология SystemTap кроме получения графа потока вызова позволяет просматривать параметры функций и возвращаемые ими значения, что является необходимым с точки зрения контроля информационных объектов, содержащих критически важную информацию. Предлагаемый комплексный подход является гибким, масштабируемым и может быть применен при проведении сертификационных испытаний программного обеспечения на отсутствие недекларированных возможностей.

***Ключевые слова**: сертификационные испытания, отладочная информация, регулярные выражения, трассировка, SystemTap.*

Введение

В условиях принятого плана импортозамещения программного обеспечения [1] необходимо констатировать тот факт, что в перспективе видится серьезный сдвиг в область использования в государственных структурах в качестве системного (общего) программного обеспечения (ПО) операционных систем (ОС) на базе ядра Linux. Это подталкивает производителей к разработке прикладного и специализированного ПО для данной платформы и, как следствие, к увеличению их доли в общем объеме как на рынке коммерческих продуктов, так и в сегменте сертификации.

ОС семейства Linux предоставляют широкие возможности по выявлению дефектов и уязвимостей в исходных текстах программ посредством штатных механизмов операционных систем и свободно распространяемого ПО, реализующего функции анализа кода (как статического, так и динамического) [2]. Активное применение данного инструментария, в дополнение к имеющимся сертифицированным средствам анализа исходных текстов программ, позволяет повысить обоснованность и достоверность принятия решения экспертами при выполнении соответствующих проверок.

Проведенный анализ публикаций, посвященных проблемам выявления уязвимостей в программном обеспечении [3–7], показал их существенный акцент в сторону освещения вопросов, касающихся проведения статического анализа исходных текстов и динамического анализа исполняемого кода. Вопросы же динамического анализа исходных текстов программ, проведение которого регламентируется руководящим документом [8], на наш взгляд, не так широко раскрыты и требуют более детального рассмотрения.

Сущность задачи динамического анализа исходных текстов программ

Исходя из определения [8], целью динамического анализа является контроль соответствия реализованных и декларированных в документации функциональных возможностей исследуемого ПО. При этом контроль заключается в сопоставлении фактических маршрутов выполнения функциональных объектов с маршрутами, построенными в процессе проведения статического анализа.

Как видно из цели, основной задачей динамического анализа исходных текстов программ является выявление (идентификация) фактических маршрутов выполнения функциональных объектов,

1 Мельников Павел Вячеславович, кандидат технических наук, Академия ФСО России, г. Орёл, palmel@inbox.ru
2 Горюнов Максим Николаевич, кандидат технических наук, Академия ФСО России, г. Орёл, max.gor@mail.ru
3 Анисимов Дмитрий Владимирович, Академия ФСО России, г. Орёл, dimadikiy@mail.ru

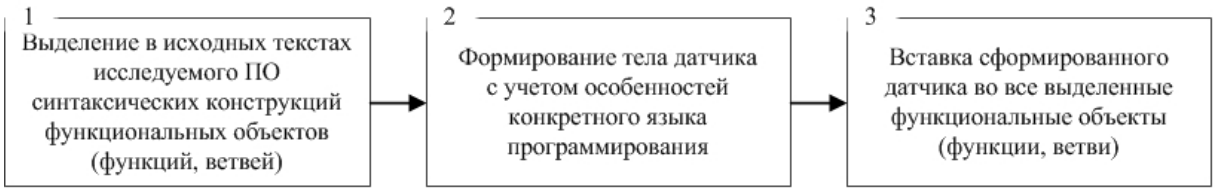

***Рис. 1**. Обобщенная последовательность действий вставки датчиков в исходные тексты программ*

что, как правило, сопровождается модификацией исходных текстов путем вставки в них так называемых датчиков, позволяющих восстановить последовательность их срабатывания в ходе выполнения исполняемых модулей исследуемого ПО.

Учитывая тот факт, что указанная модификация не предусматривается на стадии разработки исследуемого ПО, ее проведение, как правило, влияет на процесс последующей компиляции модифицированной версии исходных текстов. Особенно остро это проблема встает для сложных объемных проектов и связана она с возможными случаями некорректной вставки датчиков, что не позволяет успешно осуществить последующую сборку проекта. Это приводит к необходимости проведения значительного числа повторных итераций сборки с последовательным устранением ошибок компиляции. При отсутствии (для коммерческих продуктов) априорных знаний об используемом алгоритме вставки датчиков в исходные тексты, работа эксперта по проведению динамического анализа может быть сопряжена с большими временными и вычислительными затратами.

Необходимо отметить, что в настоящее время существуют апробированные средства, реализующие подобный функционал модификации исходных текстов, например АК-ВС, вторая версия которого существенно минимизирует ошибки вставки датчиков. Вместе с тем данное программное средство ориентировано на широко распространенные языки программирования и с этой точки зрения не является универсальным. Все это обусловливает актуальность вопросов, касающихся разработки методов проведения динамического анализа и программных средств, их реализующих, обеспечивающих возможность адаптации под новые языки программирования.

Динамический анализ исходных текстов программ с использованием регулярных выражений

Одним из вариантов решения указанной проблемы является применение подхода на основе регулярных выражений. К его достоинствам относятся простота реализации и гибкость модификации при последовательном устранении однотипных групп ошибок вставки датчиков.

Регулярные выражения представляют собой мощный инструмент обработки текстов, получивший глубокую поддержку в ОС семейства Linux и позволяющий решать широкий круг задач, в том числе и вставку датчиков в исходные тексты при проведении динамического анализа. Выполняемая при вставке датчиков обобщенная последовательность действий представлена на рисунке 1.

Рассмотрим применение механизма регулярных выражений при проведении динамического анализа на примере. В качестве тестовых данных выбрано программное средство sloccount с открытым исходным кодом на языке С, а в качестве среды анализа и сборки – операционная системой Ubuntu 15.04. При подготовке среды анализа и сборки были установлены пакеты llvm и clang (команда «sudo apt-get install clang»), а также gawk (команда «sudo apt-get install gawk»). При подготовке исходных данных были произведены загрузка и распаковка дерева исходных текстов ПО sloccount:

```
$ wget http://www.dwheeler.com/sloccount/sloccount-2.26.tar.gz
$ tar -xvf ./sloccount-2.26.tar.gz
```

ПО sloccount может применяться при статическом анализе исходных текстов [9]. Однако в данном случае предлагается использовать код указанного ПО в качестве исходных данных для примера проведения динамического анализа на основе регулярных выражений.

При переходе в директорию с исходным кодом ПО sloccount видно (команда «find . –name '*.c'»), что имеется девять файлов описания функциональных объектов (stripccomments.c, jsp_count.c, ml_count.c, temp.c, php_count.c, driver.c, lexcount1.c, c_count.c и pascal_count.c), используемых при сборке[4].

В последовательности проверок при проведении исследований исходных текстов программ динамический анализ выполняется после статиче-

4 С целью соблюдения объемов статьи примеры вставки датчиков в функциональные объекты других языков программирования не рассматриваются.

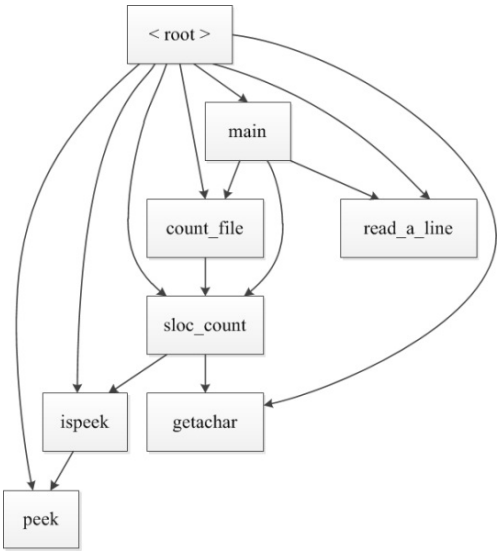

***Рис. 2**. Граф вызова функциональных объектов для файла «c_count.c»*

ского [8]. Вопросы проведения статического анализа выходят за рамки излагаемого материала, поэтому предполагается, что проверки, составляющие суть статического анализа исходных текстов [4], уже закончены, и у экспертов имеются графы вызовов функций (рис. 2), построенные, например, с использованием команды:

```
$ clang -Xclang -analyze -Xclang -analyzer-checker=debug.ViewCallGraph
                                                          c_count.c
```

Используя графы вызовов функциональных объектов, эксперт получает представление о маршрутах их выполнения (дополнительная декларативная описательная информация извлекается из документации на исследуемое ПО). Однако идентификация фактических маршрутов выполнения функциональных объектов возможна только при получении некоторой отладочной информации, генерируемой при их выполнении.

Для идентификации фактических маршрутов выполнения функциональных объектов предлагается модифицировать код исследуемого ПО, вставив в исходные тексты, реализующие функциональные объекты (т. е. в файлы «*.c»), идентификаторы (датчики), позволяющие отследить последовательность их вызовов. Для этого в директории с исходными текстами необходимо выполнить скрипт со следующим содержанием[5]:

```
#!/bin/bash
find . -type f  -name \*.c | while read i
do
 # вставка тела датчика
 sed -i ‹1 i#include <stdio.h>› $i
 sed -i ‹/)/,+1{s/{[ ]*$/{FILE \*pmout=fopen(«\/tmp\/log.txt», «a»);fprintf(pmout, «%s\\n», «_XYZ_»);fclose(pmout);/}› $i
 # удаление тела датчика из ветвей функциональных объектов
 sed -i ‹/^[ ]*while/,+1{s/{FILE \*pmout=fopen(«\/tmp\/log.txt», «a»);fprintf(pmout, «%s\\n», «_XYZ_»);fclose(pmout);/{/g}› $i
 sed -i ‹/^[ ]*if/,+1{s/{FILE \*pmout=fopen(«\/tmp\/log.txt», «a»);fprintf(pmout, «%s\\n», «_XYZ_»);fclose(pmout);/{/g}› $i
 sed -i ‹/^[ ]*for/,+1{s/{FILE \*pmout=fopen(«\/tmp\/log.txt», «a»);fprintf(pmout, «%s\\n», «_XYZ_»);fclose(pmout);/{/g}› $i
 sed -i ‹/^[ ]*try/,+1{s/{FILE \*pmout=fopen(«\/tmp\/log.txt», «a»);fprintf(pmout, «%s\\n», «_XYZ_»);fclose(pmout);/{/g}› $i
 sed -i ‹/^[ ]*case/,+1{s/{FILE \*pmout=fopen(«\/tmp\/log.txt», «a»);fprintf(pmout, «%s\\n», «_XYZ_»);fclose(pmout);/{/g}› $i
 sed -i ‹/^[ ]*else/,+1{s/{FILE \*pmout=fopen(«\/tmp\/log.txt», «a»);fprintf(pmout, «%s\\n», «_XYZ_»);fclose(pmout);/{/g}› $i
 # обработка ошибок вставки (проявляющихся при компиляции), характерных для синтаксиса языка C
 sed -i ‹/switch/,/case/{s/{FILE \*pmout=fopen(«\/tmp\/log.txt», «a»);fprintf(pmout, «%s\\n», «_XYZ_»);fclose(pmout);/{/g}› $i
 sed -i ‹/^[ ]*enum/,+1{s/{FILE \*pmout=fopen(«\/tmp\/log.txt», «a»);fprintf(pmout, «%s\\n», «_XYZ_»);fclose(pmout);/{/g}› $i
 sed -i ‹/^[ ]*extern/,+1{s/{FILE \*pmout=fopen(«\/tmp\/log.txt», «a»);fprintf(pmout, «%s\\n», «_XYZ_»);fcliose(pmout);/{/g}› $i
 sed -i ‹/^[ ]*do/,+1{s/{FILE \*pmout=fopen(«\/tmp\/log.txt», «a»);fprintf(pmout, «%s\\n», «_XYZ_»);fclose(pmout);/{/g}› $i
 sed -i ‹/^[ ]*catch/,+1{s/{FILE \*pmout=fopen(«\/tmp\/log.txt», «a»);fprintf(pmout, «%s\\n», «_XYZ_»);fclose(pmout);/{/g}› $i
 sed -i ‹/^[ ]*struct/,+1{s/{FILE \*pmout=fopen(«\/tmp\/log.txt», «a»);fprintf(pmout, «%s\\n», «_XYZ_»);fclose(pmout);/{/g}› $i
 sed -i ‹/^[ ]*union/,+1{s/{FILE \*pmout=fopen(«\/tmp\/log.txt», «a»);fprintf(pmout, «%s\\n», «_XYZ_»);fclose(pmout);/{/g}› $i
 sed -i ‹/^[ ]*out /,+1{s/{FILE \*pmout=fopen(«\/tmp\/log.txt», «a»);fprintf(pmout, «%s\\n», «_XYZ_»);fclose(pmout);/{/g}› $i
done
```

5 Содержание скрипта для файлов «*.cpp» отличается от представленного, однако подход к его разработке остается прежним.

В данном скрипте для текущей директории происходит последовательная обработка (с использованием команд «sed») каждого файла с расширением «*.c» . Вставка тела датчика осуществляется двумя командами: первая команда вставляет в начало файла директиву «#include <stdio.h>» с заголовочным файлом, необходимым для работы тела датчика; вторая каждый раз при нахождении конструкции «/)/,+1{s/{[ ]*$/»[6] осуществляет вставку тела датчика, которое представляет собой несколько команд, позволяющих передавать в файл «/tmp/log.txt» идентификационные данные доступа (строка «_XYZ_») к соответствующему участку кода (функциональному объекту). После выполнения вставки тела датчика осуществляется обработка файла исходных текстов двумя блоками команд, которые производят:

– удаление тела датчика из ветвей функции (если в этом есть необходимость);

– обработку ошибок вставки датчика, обусловленных особенностями синтаксиса определенного языка программирования.

Подстрока идентификационных данных «_XYZ_», используемая на первом этапе вставки тела датчика, не является уникальной, поэтому произведем ее преобразование, выполнив скрипт со следующим содержанием:

```
#!/bin/bash
find . -type f -name \*.c | while read i
do
 echo «$i ---------- OK»
 gawk ‹{gsub(«_XYZ_», «_XYZ_ «FNR» «FILENAME, $0); print $0}› $i >
                                                    tmpfile; mv tmpfile $i
 grep -o «_XYZ_.*\»» $i >> /tmp/all_datchik.txt
done
```

В результате выполнения данного скрипта все найденные подстроки «_XYZ_» в файлах «*.c» текущей директории будут дополнены номером строки и именем файла, в которых они были найдены. Таким образом, каждый вставленный в функциональный объект датчик получает уникальный идентификатор, например «_XYZ_ 48 ./sloccount-2.26/driver.c», а список всех вставленных датчиков помещается в файл «/tmp/all_datchik.txt».

Теперь для получения фактических маршрутов выполнения функциональных объектов ПО sloccount необходимо выполнить ряд действий:

- произвести сборку модифицированных исходных текстов со вставленными датчиками (команда «make»);

- осуществить установку исполняемых файлов, полученных в результате сборки (команда «sudo make install»);

- запустить полученные исполняемые файлы на выполнение, используя команду «sloccount --wide --filecount --multiproject --addlangall ./sloccount-2.26 > sloccount_result.txt» (при этом результат работы ПО sloccount помещается в файл sloccount_result.txt, а список всех сработавших датчиков – в файл «/tmp/log.txt»);

- отсортировать список всех сработавших датчиков (файл «/tmp/log.txt») по критерию уникальности с использованием команды «sort ./log.txt | uniq >> ./uniq.txt».

Результатом выполнения указанной последовательности действий является набор данных, необходимых эксперту для достижения цели динамического анализа исходных текстов программ, а именно:

- графы вызовов функциональных объектов, построенные в ходе проведения статического анализа (маршруты выполнения функциональных объектов);

- log.txt – трасса сработавших датчиков при выполнении исполняемых объектов исследуемого ПО (фактический маршрут выполнения функциональных объектов);

- all_datchik.txt – список всех вставленных датчиков;

- uniq.txt – список всех сработавших датчиков.

В ходе сопоставления графов вызовов, полученных на этапе статического анализа, с трассами вызовов, полученными на этапе динамического анализа, экспертами делается вывод о (не)соответствии реализованных и декларированных в документации функциональных возможностей ПО и тем самым достигается цель динамического анализа программ.

В ходе сопоставления списка всех вставленных датчиков со списком всех сработавших эксперт получает показатель в виде процента покрытия функциональных объектов. Требуемое значение данного показателя руководящими документами не определяется, однако очевидно, что одной из подзадач в процессе проведения динамического анализа является получение как можно большего значения данного показателя. При этом назрела насущная необходимость в задании значения данного показателя на определенном (достаточно обоснованном) уровне.

6 Несложно заметить, что при наличии двух и более пустых строк после сигнатуры «)» вставка датчика не будет выполнена. Такая ситуация решается предварительной обработкой файлов исходных текстов с удалением пустых строк.

Динамический анализ исходных текстов программ с использованием средств динамической трассировки

В настоящее время существует ряд средств динамического анализа программ [10], позволяющих осуществлять их трассировку. По функциональным возможностям особо выделяется технология SystemTap, работающая на уровне ядра ОС Linux и позволяющая контролировать как само ядро и его модули, так и приложения пользователей. Это позволяет получить полную информацию о том, что происходит с любым исполняемым модулем при его работе в системе.

Программное обеспечение SystemTap контролирует основные действия трассируемого объекта – выполнение процессорных команд, работу с памятью, с файловой системой, с сетью и при этом динамически модифицирует объект исследования таким образом, что выключенные датчики не оказывают влияния на производительность и включаются только при явном указании, что делает данную технологию привлекательной с точки зрения ее использования при проведении динамического анализа исходных текстов программ, представляющих собой объемные проекты. Важным и наиболее существенным преимуществом средства динамического анализа SystemTap является то, что его можно использовать для анализа ПО, не прерывая его работы.

Проведение динамического анализа программ с помощью технологии SystemTap предусматривает написание соответствующих скриптов на скриптовом языке. Данный язык предоставляет широкие возможности по трассировке исполняемых модулей и позволяет проводить исследования с разным уровнем детализации получаемых данных. Построение требуемого в соответствии с [8] графа потока управления может быть осуществлено с помощью простого скрипта calls.stp:

```
function header(arrow:string, indent:long)
{ printf(«%s %s %s «,
  thread_indent(indent), arrow, ppfunc());}
probe $1.call
{ header(«->», 1);println();}
probe $1.return
{ header(«<-», -1);println();}
```

В качестве входных параметров ему определяется объект контроля с указанием шаблона типа (ядро, модуль ядра, приложение) и контролируемых функций. Например, для контроля последовательности функций, вызываемых при работе приложения ls (входят в сборку одноименного исполняемого файла из каталога /src), достаточно вызвать скрипт со следующими параметрами:

```
stap calls.stp ‹process(«ls»).function(«*@src/ls.c»)› -c «ls»
```

Работой скрипта будет трасса вида:

```
  0 ls(18256): -> main
145 ls(18256):  -> clear_files
151 ls(18256):  <- clear_files
153 ls(18256):  -> queue_directory
159 ls(18256):  <- queue_directory
```

При этом использование специализированных переменных parms и return дополнительно позволяет просматривать параметры функций и возвращаемые ими значения, что является необходимым с точки зрения контроля информационных объектов, содержащих критически важную информацию, например, с модифицированными в рассматриваемом скрипте датчиками вида

```
probe $1.call
{ header(«->», 1); println(«params: «, $$parms);}
probe $1.return
{ header(«<-», -1);println(«returns «, $$return);}
```

трасса будет выглядеть следующим образом:

```
987 ls(18673):  -> process(«/usr/local/bin/ls»).function(«gobble_file@
        src/ls.c:2887»).call,params: name=»call.txt» type=0command_
        line_arg=0 dirname=».» inode=0
995 ls(18673):  <- process(«/usr/local/bin/ls»).function(«gobble_file@
                                src/ls.c:2887»).return,returns: return=0
```

В целом, технология SystemTap позволяет получить граф вызовов функций в приложении и проводить динамический анализ по третьему уровню контроля отсутствия недекларированных возможностей в полном объеме, а по второму уровню контроля – только в части контроля ветвей, содержащих функциональные объекты.

Выводы

В работе представлены возможности штатных механизмов операционных систем и свободно распространяемого ПО для проведения динамического анализа исходных текстов программ. Рассмотрены подходы на основе использования механизма регулярных выражений и технологии трассировки Systemtap, предоставляющие большой потенциал по контролю выполнения функциональных объектов. Решение на основе регулярных выражений характеризуется простотой реализации и гибкостью, заключающейся в возможности оперативной корректировки алгоритма вставки датчиков (при выявлении ошибок на новых тестовых данных) и адаптации под новые языки программирования.

Важным достоинством SystemTap является динамическая модификация объекта исследования с минимальным результирующим влиянием на

производительность. Это делает данную технологию привлекательной с точки зрения ее использования при проведении динамического анализа объемных проектов. Применение специализированных переменных SystemTap позволяет просматривать параметры функций и возвращаемые ими значения, что является необходимым с точки зрения контроля информационных объектов, содержащих критически важную информацию. Технология SystemTap позволяет в полном объеме проводить динамический анализ по третьему уровню контроля отсутствия недекларированных возможностей (по второму уровню контроля – только в части контроля ветвей, содержащих функциональные объекты).

Таким образом, при проведении динамического анализа исходных текстов программного обеспечения целесообразно использовать комплексный подход, предполагающий применение как специализированных средств вставки датчиков, так и штатных средств отладки, имеющихся в операционных системах. Это позволит повысить обоснованность и достоверность принятия решения о наличии дефектов и недекларированных возможностей, а также снизить временные и вычислительные затраты при проведении проверок.

***Рецензент**: Невров Алексей Александрович, кандидат технических наук, сотрудник Академии ФСО России, г.Орел, newrow@mail.ru*

Литература:

1. Об утверждении плана импортозамещения программного обеспечения. URL: http://minsvyaz.ru/ru/documents/4548/.
2. The LLVM Compiler Infrastructure. URL: http://llvm.org.
3. Анисимов Д.В., Мельников П.В. Проведение сертификационных исследований программного обеспечения с использованием технологии LLVM // Информационные системы и технологии. 2016. № 2 (94). С. 99-104.
4. Аветисян А., Белеванцев А., Бородин А., Несов В. Использование статического анализа для поиска уязвимостей и критических ошибок в исходном коде программ // Труды Института системного программирования РАН. 2011. Т. 21. С. 23-38.
5. Аветисян А.И., Белеванцев А.А., Чукляев И.И. Технологии статического и динамического анализа уязвимостей программного обеспечения // Вопросы кибербезопасности. 2014. № 3 (4). С. 20-28.
6. Марков А.С., Фадин А.А. Систематика уязвимостей и дефектов безопасности программных ресурсов // Защита информации. Инсайд. 2013. № 3 (51). С. 56-61.
7. Марков А.С., Матвеев В.А., Фадин А.А., Цирлов В.Л. Эвристический анализ безопасности программного кода // Вестник Московского государственного технического университета им. Н.Э. Баумана. Серия: Приборостроение. 2016. № 1 (106). С. 98-111. DOI: 10.18698/0236-3933-2016-1-98-111.
8. Руководящий документ Гостехкомиссии России «Защита от несанкционированного доступа к информации. Часть 1. Программное обеспечение средств защиты информации. Классификация по уровням контроля отсутствия недекларированных возможностей». Гостехкомиссия России, 1999. URL: http://fstec.ru/component/attachments/download/294.
9. SLOCCount. URL: http://www.dwheeler.com/sloccount.
10. Dynamic Tracing with DTrace & SystemTap. URL: http://myaut.github.io/dtrace-stap-book.

APPROACH TO IMPLEMENTATION OF DYNAMIC ANALYSIS PROGRAMS SOURCE TEXTS

Melnikov P.V.[7], Gorunov M.N.[8], Anisimov D.V.[9]

In article the overview of the approach which is carried out for analysis and comparison declared and implemented routes of function objects execution in programs source texts is provided. The problem of a correct insert of sensors and obtaining the necessary debug information when carrying out the analysis

7 Pavel Melnikov, Ph.D., Employee, Academy of Federal Agency of protection of the Russian Federation, Orel, palmel@inbox.ru
8 Maxim Gorunov, Ph.D., Employee, Academy of Federal Agency of protection of the Russian Federation, Orel, max.gor@mail.ru
9 Dmitry Anisimov, Employee, Academy of Federal Agency of protection of the Russian Federation, Orel, dimadikiy@mail.ru

of difficult and volume projects is considered. Within problem decision the mechanism of regular expressions and SystemTap dynamic trace is offered. Advantage of the provided approach of regular expressions is openness and simplicity of modification the source code an insert sensors, that allows the expert to adjust quickly it under specific conditions of carrying out the analysis. The SystemTap technology, except receiving the graph of a flow call, allows to view functions parameters and values returned by them that is necessary to control the information objects containing crucial information. The offered integrated approach is flexible, scalable and can be used when carrying out certified tests.

***Keywords**: certified tests, debug information, regular expressions, trace, SystemTap*

References:

1. Ob utverzhdenii plana importozameshcheniya programmnogo obespecheniya, URL: http://minsvyaz.ru/ru/documents/4548.
2. The LLVM Compiler Infrastructure, URL: http://llvm.org.
3. Anisimov D. V., Mel'nikov P. V. Provedenie sertifikatsionnykh issledovaniy programmnogo obespecheniya s ispol'zovaniem tekhnologii LLVM [Tekst], D. V. Anisimov, Informatsionnye sistemy i tekhnologii. 2016. No 2 (94), pp. 99-104.
4. Avetisyan A., Belevantsev A., Borodin A., Nesov V. Ispol'zovanie staticheskogo analiza dlya poiska uyazvimostey i kriticheskikh oshibok v iskhodnom kode program, Trudy Instituta sistemnogo programmirovaniya RAN. 2011. T. 21, pp. 23-38.
5. Avetisyan A.I., Belevantsev A.A., Chuklyaev I.I. Tekhnologii staticheskogo i dinamicheskogo analiza uyazvimostey programmnogo obespecheniya, Voprosy kiberbezopasnosti. 2014. No 3 (4), pp. 20-28.
6. Markov A.S., Fadin A.A. Sistematika uyazvimostey i defektov bezopasnosti programmnykh resursov, Zashchita informatsii. Insayd. 2013. No 3 (51), pp. 56-61.
7. Markov A.S., Matveev V.A., Fadin A.A., Tsirlov V.L. Evristicheskiy analiz bezopasnosti programmnogo koda, Vestnik Moskovskogo gosudarstvennogo tekhnicheskogo universiteta im. N.E. Baumana. Seriya: Priborostroenie. 2016. No 1 (106), pp. 98-111. DOI: 10.18698/0236-3933-2016-1-98-111.
8. Rukovodyashchiy dokument Gostekhkomissii Rossii «Zashchita ot nesanktsionirovannogo dostupa k informatsii. Chast' 1. Programmnoe obespechenie sredstv zashchity informatsii. Klassifikatsiya po urovnyam kontrolya otsutstviya nedeklarirovannykh vozmozhnostey», Gostekhkomissiya Rossii, – M., 1999. URL: http://fstec.ru/component/attachments/download/294.
9. SLOCCount / URL: http://www.dwheeler.com/sloccount (data obrashcheniya 24.04.2016).
10. Dynamic Tracing with DTrace & SystemTap, URL: http://myaut.github.io/dtrace-stap-book.