Российская Академия наук
Институт Системного Программирования

Труды Института Системного Программирования

Математические методы и алгоритмы

Том 11

Под редакцией
чл.-корр. РАН В.П.Иванникова

Москва 2006

УДК 51

Труды Института системного программирования: Том 11.
/Под ред. В.П.Иванникова/ — М.:ИСП РАН, 2006. — 128 с.

Одиннадцатый том Трудов Института системного программирования составлен из научных статей, посвященных некоторым актуальным вопросам компьютерной безопасности, которые обсуждались на школе-семинаре «Теоретические и экспериментальные аспекты маскировки программ»
(28–30 апреля 2005, ИСП РАН, Москва).

© Институт Системного Программирования РАН, 2006

Содержание

Предисловие 5

С.И. Алейников, А.О. Богатов
Защита программ от дизассемблирования 9

Н. Вяххи и А. Коваль
Архитектурный подход к защите программ 17

Н.П. Варновский, В.А. Захаров, Н.Н. Кузюрин, А.В. Чернов, А.В. Шокуров
Об особенностях применения методов обфускации программ для информационной защиты микроэлектронных схем 27

В.А. Захаров, К.С. Иванов
О проблеме логико-термальной эквивалентности последовательных программ с динамической памятью 61

A. Belevantsev and O. Malikov
Using data flow analysis for detecting security vulnerabilities 83

Y. Lifshits
Guaranteed Slowdown, Generalized Encryption Scheme, and Function Sharing 99

R. Podlovchenko, D. Rusakov and V. Zakharov
The equivalence problem for programs with mode switching is PSPACE-complete 109

Предисловие

В настоящем сборнике представлены статьи, посвященные разработке математических методов и алгоритмов для различных задач дискретной математики и теоретического программирования. Первые девять статей посвящены разработке и анализу алгоритмов в некоторых задачах дискретной математики, а последние четыре работы связаны с анализом и преобразованиями программ в целях обеспечения компьютерной безопасности.

В статье А.И.Поспелова рассмотрена задача упаковки прямоугольников в заданное число полос, тесно связанная с построением расписаний для группы кластеров. В статье предложен приближенный алгоритм и показано, что он гарантированно дает размещение, не более чем в 3 раза худшее по сравнению с оптимальным.

Статья С.Н.Жука также посвящена анализу некоторых эвристик в задаче упаковки прямоугольников в несколько полос. Предложен эффективный алгоритм, который размещает прямоугольники по полосам в онлайновом режиме и гарантирует константную мультипликативную точность. Это достигнуто за счёт правильной формализации понятии «допустимая полоса для прямоугольника». Показано, также, что полученная оценка точности достижима на некоторых исходных данных.

В статье С.А. Фомина описан новый приближенный алгоритм для задачи положительного линейного программирования, имеющий наилучшую из известных оценок быстродействия.

В статье Н.Н. Кузюрина и О.А. Прокопьева доказана алгоритмическая трудность распознавания возможности аппроксимации заданной булевой функции другой (более простой) булевой функцией.

В статье М.Н. Вялого рассматривается алгоритмическая сложность основных задач комбинаторики слов при задании слова сжатым описанием.

А именно, в качестве слов, в которых ищутся вхождения подслов, рассматриваются таблицы значений булевых полиномов. Построены эффективные алгоритмы проверки вхождения слова фиксированной длины в таблицу значений полинома фиксированной степени, последовательного порождения вхождений при тех же условиях.

В статье Т.В.Андреевой доказана унимодальность частично упорядоченного множества, диаграмма Хассе которого является декартовой степенью k–звезды, т.е. дерева с $k+1$ вершинами, одна из которых имеет степень k.

В статье Н.Н. Кузюрина «Обобщенные покрытия и их аппроксимации» получены верхние и нижние оценки размеров обобщенных покрытий матриц, состоящих из 0, 1 и -1. Полученные оценки свидетельствуют о точности по порядку метода вероятностного округления в задачах ЦЛП с $(0, \pm 1)$-матрицами ограничений.

В статье Н.Н. Кузюрина "Probabilistic analysis of the greedy algorithm" доказана асимптотическая точность жадного алгоритма в задаче о покрытии при анализе в среднем (на случайных исходных данных).

В статье Н.Н. Кузюрина, С.А. Мартишина и В.М. Храпченко рассматривается задача поиска часто встречающихся комбинаций, связанная с анализом данных (data mining). Рассмотрены некоторые теоретические аспекты, связанные с алгоритмической сложностью задачи и существованием эффективных приближенных алгоритмов. Предложен генетический алгоритм для решения этой задачи и проведено исследование его эффективности на случайных данных.

В статье Н.П.Варновского исследуется проблема определения стойкости обфускирующих преобразований программ. В статье обсуждаются требования к стойкости обфускации и устанавливается невозможность построения универсального обфускатора, удовлетворяющего требованиям стойкости, основанным на парадигме "серого ящика".

В статье К.С.Иванова и В.А.Захарова предложен новый подход к оценке качества обфускирующих преобразований, используемых на практике: об-

фускация считается относительно стойкой, если она способна противодействовать прикладным алгоритмам верификации и анализа программ. Построены примеры обфускирующих преобразований, соответствующих введенному критерию, и оценена их сложность.

В статье А.В.Шокурова изучается задача построения гомоморфных систем шифрования данных, позволяющих проводить вычисления над зашифрованными данными. Показано, что для одного класса алгебраических схем вычислений существует такое эффективное гомоморфное шифрование, при котором размер схемы изменяется незначительно, но извлечение исходных данных на основе наблюдаемых вычислений становится практически невозможным.

В статье И.М.Захарьящева и В.А.Захарова предложена новая модель программ, предназначенная для решения задач проверки эквивалентности программ и проведения эквивалентных преобразований. Отличительная особенность этой модели состоит в том, что построение трассы вычисления и формирование результатов вычисления проводятся на основе разных операционных семантик. Это позволяет описывать в рамках единой модели поведение программ различных типов, включая реагирующие программы. Предложен новый метод построения алгоритмов проверки эквивалентности программ в предложенной модели.

Чл.-корр. РАН В.П.Иванников

Защита программ от дизассемблирования

С. И. Алейников, А. О. Богатов

Аннотация. Задача дизассемблирования для архитектуры x86 весьма сложна. Для ещё большего её усложнения, что важно для защиты программ от взлома, неоднократно предпринимались попытки преобразовать бинарный код, чтобы увеличить трудоёмкость дизассемблирования. Задачей данной работы является, прежде всего, сравнение известных методов обфускации бинарного кода. Мы также оценим эффективность приёмов борьбы с ними и выявим наиболее перспективные техники. В заключение мы ответим на вопрос, насколько вообще действенна обфускация бинарного кода. Предлагаются два нестандартных метода. Перекрывающиеся инструкции, если использовать их повсеместно, фактически уничтожают граф потока управления. Двойное шифрование делает невозможными малейшие изменения кода при попытке взлома.

1. Введение

Большинство программ распространяются в виде исполняемого файла. Для взлома программы с целью пиратства необходимо прежде всего перевести машинный код в ассемблерное представление. Такой процесс носит название *дизассемблирования*. Если удастся сделать неразрешимой задачу взломщика уже на этом этапе исследования программы, шансы на сохранность интеллектуальной собственности станут стопроцентными.

Трудность задачи дизассемблирования для архитектуры x86 известна давно. Она обусловлена следующими причинами:

1. отсутствует чёткое разделение данных и кода, в результате чего код можно принять за данные и наоборот;

2. не фиксирована длина инструкций, вследствие чего иногда нельзя определить, где начинается та или иная инструкция;

3. имеется косвенная адресация, поэтому мы можем даже примерно не знать адрес, на который осуществится переход в результате исполнения некоторой команды передачи управления;

4. возможна модификация кода во время исполнения;

5. наконец (хоть это и не является отличительной особенностью рассматриваемой архитектуры), главную проблему создают многочисленные ветвления кода.

Возникает вопрос: как использовать данные особенности при создании кода, трудного для дизассемблирования, и каковы шансы успешно восстановить код, подвергнутый подобным преобразованиям?

Для ответа мы прежде всего рассмотрим два базовых метода дизассемблирования, затем обсудим преобразования графа потока управления и выявим его неустойчивость к многократной обработке и частотному анализу. В последнем разделе мы предложим нестандартные способы защиты кода, наиболее важными из которых кажутся двойное шифрование и перекрытие инструкций.

2. Разновидности и цели дизассемблирования

Прежде всего следует поставить границы рассматриваемой задачи. Анализировать программы можно двумя способами.

Во-первых, статически (дизассемблирование) — анализируя программный код, записанный в исполняемом файле, физически этот файл не исполняя.

Во-вторых, динамически (отладка) — анализируется программный код во время исполнения (в пошаговом режиме или при помощи статического анализа снятого «дампа» программы).

Приёмы защиты от статического анализа полностью уязвимы при анализе динамическом (как и наоборот). Поэтому для надёжной защиты программы требуется использовать приёмы против всех типов анализа. В настоящей работе рассматривается прежде всего защита от анализа первого рода.

Основными целями дизассемблирования (как и отладки) являются понимание и/или изменение кода программы — к примеру, для обхода защиты от копирования, извлечения/изменения функциональности. Защита на уровне бинарного кода придаёт программе, даже если её исходный код подвергся обфускации, большую устойчивость ко взлому. Поэтому исследование возможностей обфускации бинарного кода действительно актуально.

3. Простейшие методы дизассемблирования

Имеются простейшие приёмы дизассемблирования бинарного кода, не дающие хорошей гарантии результата, но являющиеся интуитивно очевидными (см., например, [3]). В случае анализа методом линейного прохода инструкции дизассемблируются последовательно.

Как только алгоритм видит следующий необработанный байт, он пытается рассмотреть этот байт как начало инструкции. Если не получилось восстановить инструкцию, алгоритм переходит к следующему байту. Основной недостаток этого способа — очень часто данные трактуются как инструкции.

При использовании рекурсивного обхода одновременно с дизассемблированием инструкций строится граф потока управления. После декодирования инструкции передачи управления (ИПУ) начинается декодирование по всем возможным адресам перехода. Главный недостаток такого подхода — в том, что возникают значительные затруднения в случае динамического вычисления адреса перехода (косвенной адресации).

Приведём простой пример (взятый из [2]) работы этих двух методов на одном и том же коде. Исходный (ассемблируемый) код таков:

```
push ebp                    ;55
mov esp, ebp                ;89 e5
call 1978808 <branch fnct> ;e8 00 00 74 11
    [junk]                  ;0a 05
cmp 0, eax                  ;3c 00
jne 8048014 <L1>            ;75 06
mov 0, eax                  ;b0 00
jmp 8048019 <L2>            ;eb 07
    [junk]                  ;0a 05
L1: mov (1740000), eax      ;a1 00 00 74 01
L2: mov ebp, esp            ;89 ec
pop ebp                     ;5d
ret                         ;c3
nop                         ;90
```

Линейный проход даёт такой результат:

```
push ebp                    ;55
mov esp, ebp                ;89 e5
call 1978808 <branch fnct> ;e8 00 00 74 11
or 675003c, al              ;0a 05 3c 00 75 06
mov 0, eax                  ;b0 00
jmp 8048019                 ;eb 07
or 740000a1, al             ;0a 05 a1 00 00 74
adc ecx, 90c35dec(ecx)      ;01 89 ec 5d c3 90
```

Следующий результат получен методом рекурсивного обхода:

```
push ebp                    ;55
mov esp, ebp                ;89 e5
call 1978808 <branch fnct>  ;e8 00 00 74 11
or 675003c, al              ;0a 05 3c 00 75 06
mov 0, eax                  ;b0 00
jmp 8048019                 ;eb 07
    [junk]                  ;0a 05 a1 00 00 74 01
mov ebp, esp                ;89 ec
pop ebp                     ;5d
ret                         ;c3
nop                         ;90
```

Как видно, метод линейного прохода допустил больше ошибок, второй нормализовался к концу, при этом оба метода приняли первую последовательность «мусорных» байт за код.

4. Основные методы затруднения дизассемблирования

Основные методы защиты от статических атак:

1. *Использование неразличимости данных и инструкций.* Цель — заставить дизассемблер принимать данные за инструкции или наоборот. В первом случае нарушается логика программы, появляются лишние блоки неопределённого кода, становится сложнее найти строковые константы (которые находились в секции данных) и отследить ссылки на них (к примеру, при поиске по ключевым словам ссылок на вызов диалога). Во втором случае часть кода остаётся вообще не обработанной дизассемблером. Методы, используемые для достижения этого эффекта:

    - *вставка «мусора»* — вставка «лишних» байт в недостижимые участки кода так, чтобы они казались достижимыми и были расшифрованы инструкции, следующие за ними;
    - вставка переходов на данные / (текстовых) ссылок на код.

2. *Вставка «непроницаемых предикатов»* для превращения безусловных переходов в условные. Этот приём подходит для создания никогда не исполняемых веток кода, изменения порядка дизассемблирования последующего кода и запутывания логики программы (требуется понять, что в этом условии управление передаётся всегда по одной ветке).

3. *Подмена адресов возврата функций* — используется для усложнения или полного исключения возможности вычислить статически все переходы. Обычно используется несколько техник:

   - *динамическое вычисление адреса перехода* — переход выполняется командой вида `jmp [edi]`;
   - *подмена адреса возврата в стеке* в самой функции: при переходе по команде `call <ptr>` в стек помешается адрес следующей команды, и, при возврате через `ret`, переход происходит именно на этот адрес. Командой вида `mov [esp-4], <new_ret_addr>` этот адрес можно изменить.

4. Наиболее эффективный метод против статического анализа — это *шифрование всего программного кода*. Совмещенный с антиотладочными приёмами для дешифратора, он даёт самые эффективные результаты. Основой для этого метода является возможность изменения программного кода во время исполнения. При использовании этого приёма для анализа остаётся доступным только небольшой участок незашифрованного кода — дешифратор, а весь остальной код можно получить либо сняв «дамп» с работающей программы, либо проанализировав дешифратор и написав эмулятор его выполнения.

У большей части способов защиты есть один недостаток — их невозможно реализовать при помощи компилятора с языка высокого уровня, требуется работа на уровне ассемблерных команд.

В работе [1] рассмотрен механизм запутывания бинарного кода с использованием следующих методов: вставка «мусора», «непроницаемых предикатов», подмена адреса возврата. Получены следующие результаты: неверно дизассемблировано около 75% инструкций методом линейного прохода и примерно 40% инструкций — методом рекурсивного обхода. В обоих случаях процент ошибок достаточно велик, чтобы утверждать, что дизассемблер со своей задачей не справился.

5. Обход основных методов защиты

В статье [2] предложена следующая техника дизассемблирования запутанного (без помощи шифрования) машинного кода.

Первоначально, чтобы локализовать свои действия, разбиваем весь код на функции: ищем в бинарном файле шаблоны, напоминающие заголовки функций. Затем декодируем инструкции, считая каждый байт возможным началом инструкции (т.е. получаем множество разнообразных вариантов).

Выделяя среди полученных инструкций ИПУ, строим первый вариант графа потока управления. Выделяем внутри каждой функции базовые блоки.

После этого из пары перекрывающихся блоков удаляем один. Конфликты разрешаются по следующим правилам: прежде всего, стартовая инструкция функции включается в окончательный вариант графа потока управления. Далее, если из некоторого блока достижимы два перекрывающихся блока, то блок-родитель считается неверным и удаляется. Затем из каждой пары перекрывающихся блоков оставляем тот, который имеет больше связей. Наконец, если конфликты ещё остались, удаляем один из конфликтующих блоков случайным образом.

Заполняем пустоты (там может содержаться код, достижимый при помощи ИПУ с динамически вычисляемым адресом перехода), при этом руководствуясь соображениями правдоподобия (используя эвристики).

При необходимости мы можем эмулировать исполнение некоторого блока кода незначительной длины.

Возможны и иные способы преодоления защиты.

Для разделения кода и данных дизассемблер может использовать несколько проходов по коду, использовать косвенные признаки. Правда, стопроцентной гарантии это не даёт.

В автоматическом режиме отделение «непроницаемых предикатов» невозможно, но возможно вычислить в коде ссылке на те данные, которые дизассемблируются как код, если дизассемблер делает несколько проходов по коду.

Переходы же по динамически вычисляемым адресам невозможно отследить иначе, как в отладчике во время исполнения.

Получение расшифрованного текста программы возможно двумя путями: во-первых, снятием дампа при помощи отладчика — в этом случае приходится бороться с антиотладочными приёмами; во-вторых, в известном коммерческом дизассемблере IDA Pro можно написать макрос, эмулирующий работу дешифратора, и получить код, пригодный для дизассемблирования, без использования отладчика.

6. Нестандартные приёмы защиты

Предложим не использовавшиеся в известных нам работах приёмы защиты.

Первый способ запутывания, вытекающий из рассмотренного выше, — это *использование недокументированных/нестандартных команд.* Есть шансы, что дизассемблер, руководствуясь соображениями правдоподобия, примет такие команды за что-либо другое и ошибётся.

Главный его недостаток — это одноразовый способ. После однократного обнаружения недокументированной команды она становится известной, заносится в базу команд дизассемблера и становится документированной.

Разновидность данного подхода, пригодная для борьбы с отладкой, — использование известных ошибок известных отладчиков и антивирусных анализаторов. Например, в отладчике TurboDebugger можно вызвать исключение путём следующих операций со стеком: `neg esp; neg esp`.

Следующий весьма эффективный метод — это *использование перекрывающихся инструкций*. В статье [1] приведены данные о том, что авторы предпринимали попытку включить этот метод в свой арсенал запутывающих преобразований. Результаты оказались такими: эффективность крайне высока, но велика трудоёмкость (сложно построить по произвольному коду эквивалентный ему и содержащий перекрывающиеся инструкции). Тем не менее, во многих случаях можно искусственно получить перекрывание инструкций. Вот типичный пример:

```
<..some code..>
add eax, <code_of_jump_instruction>
jmp -5
```

Последовательность байт, представляющая эти инструкции, выглядит так (вставлен переход по адресу +17):

```
b8 00 00 eb 17 eb f6
```

Выполняется первый переход — передача управления на 3-й переход, а из него мы попадаем в середину 2-й инструкции. Более того, `required_instruction` обязана быть инструкцией перехода — чтобы не произошло ошибки при попытке повторного разбора перехода номер 3 (иначе он будет выполняться тоже с середины инструкции).

Основной плюс метода — в том, что известные дизассемблеры не способны интерпретировать один и тот же участок кода другим способом, уже однажды поставив ему в соответствие последовательность команд. Задача полного рассмотрения всех вариантов в случае использования перекрывания инструкций практически невыполнима.

Таким образом, можно сформулировать следующую исследовательскую задачу: построить систему эквивалентных преобразований бинарного кода, направленных на получение как можно большего числа перекрывающихся инструкций.

Приём против отладки дешифратора — использовать в качестве кода для расшифровки *динамически вычисляемый CRC кода* самого *дешифратора*. Смысл таков:

1. Считаем CRC (Cyclic Redundancy Code) первого дешифратора (можно брать различные функции от кода дешифратора, в простейшем случае — `xor` всех байт). Цель приёма — в том, чтобы невозможно было установить точку останова на первый дешифратор (на какую-либо его инструкцию) и перехватить управление. Точка останова меняет байт в том месте, где она расположена, и CRC будет посчитан неверно. (Для невозможности установки аппаратной точки останова необходимо при этом использовать аппаратные регистры контроля точек останова.)
2. При помощи получившегося CRC первым дешифратором расшифровываем второй дешифратор.
3. Второй дешифратор расшифровывает основной код (желательно, чтобы до него труднее было добраться, расшифровывать код в динамически выделяемую область памяти).

7. Заключение

Итак, для взлома бинарного кода необходимо начать с его дизассемблирования. Существует множество способов затруднить дизассемблирование программы. Однако эффект от большинства из них, приложив некоторые усилия (использовав повторные проходы, статистический анализ, эмуляцию), можно свести к минимуму. По нашему мнению, способы, базирующиеся на усилении стандартных трудностей (неразличимость кода и данных, различная длина инструкций), дают лишь временный эффект, пока взломщики не научились их обходить. Наиболее надёжной защитой нам представляется шифрование кода, в особенности двойное (и вообще многоуровневое). Перспективным и заманчивым представляется также создание перекрывающихся инструкций. Хотя автоматическим методам получения ассемблерного кода обфускация противодействует весьма эффективно, но вообще дизассемблирование — во многом творческий процесс, и эксперт, варьируя подходы и методы, способен преодолеть большинство трудностей.

Литература

[1] Cullen Linn, Saumya Debray. Obfuscation of Executable Code to Improve Resistance to Static Disassembly.

[2] Christopher Kruegel, William Robertson, Fredrik Valeur and Giovanni Vigna. Static Disassembly of Obfuscated Binaries.

[3] Shengying Li. A Survey on Tools for Binary Code Analysis. August 2004.

[4] `http://www.wasm.ru/`

Архитектурный подход к защите программ

Николай Вяххи, Анатолий Коваль

Аннотация. Методы аппаратной («железной») защиты программ обычно более устойчивы к взлому, чем программная обфускация, потому что исследовать физическое устройство труднее и дороже, чем программу в уже известной архитектуре.

В данной статье рассмотрено совмещение двух перспективных направлений аппаратной защиты — eXecute Only Memory (XOM) и Dynamic Address Table (DAT). Они по отдельности хорошо сочетают в себе защищенность, прозрачность для программ и удобство использования. Но, главное, они дополняют друг друга, допуская совместное использование в одной архитектуре. Это позволяет защитить код программы от попыток его раскрытия какими либо средствами системы, что является шагом к достижению Black Box Security и избежанию многих атак, связанных с анализом исходного кода.

1. Введение

Каким образом можно защитить программу с помощью «железа» (то есть, не дать прямой доступ к исходному коду)? Можно попытаться «спрятать» некоторые данные или даже целые алгоритмы «внутри» аппаратуры. Но что значит спрятать? Разве программа, находясь в оперативной памяти (то есть «внутри» аппаратуры), не достаточно «спрятана»? На самом деле оперативная память, как и любой ресурс доступный приложениям, не является хорошим тайником, другие приложения могут читать код и данные из памяти нашего приложения (если наша архитектура предоставляет защиту приложений друг от друга, мы все равно не можем доверять операционной системе).

Далее в статье мы сначала рассмотрим общие положения архитектурного подхода и его особенности. Затем будут разобраны алгоритмы работы XOM и DAT. Во второй половине статьи будет предпринята попытка совместить XOM и DAT, исследовать полученную архитектуру на стойкость, оценить эффективность исполнения на ней программ, возможность дальнейших улучшений.

2. Архитектурный подход: общие положения

Неплохо было бы сделать новый компьютер для каждого нового приложения. Сделав его (с собственной уникальной архитектурой и системой команд) можно очень хорошо «спрятать» приложение, но такой подход противоречит принципу построения многоцелевых машин и стоит больших затрат.

Можно «сузить» решение следующим образом: разрабатывать не компьютер целиком, а только центральный процессор (который будет присоединяться к стандартным интерфейсам ввода/вывода). Идея продавать программно-аппаратные пакеты (software hardware package = SH-package) для защиты от незаконного тиражирования программ, была предложена в [Best79] и [Kent80], но, как можно заметить, не приобрела популярности за 20–25 лет существования. Само появление «железа» в программном проекте, существенно увеличивает стоимость и время разработки, стоимость конечного продукта (даже в том случае, когда аппаратура имеет очень маленькую функциональность). И понятно, что чем больше функциональности мы возложим на аппаратуру, тем больше будет стоить наш продукт.

Итак, существует два подхода архитектурной защиты программ:

- **реализовывать программу аппаратно (в железе)** — дорог в разработке и распространении, но более безопасен, ведь исследовать цельный процессор и понять, как он работает, очень трудоемко и ресурсоемко.
- **реализовывать ее программно (имея в железе универсальный механизм защиты любых возможных программ)** — дешевизна разработки нового ПО, дешевизна самого устройства (из-за большого количества одинаковых устройств для различных задач), но возможны проблемы с безопасностью при передаче данных и, так как устройство одинаково для всех программ, то его взлом более вероятен.

Мы рассмотрим два вида защиты, принадлежащие второй группе подходов, следовательно являющиеся более доступными и перспективными. При этом, они хорошо дополняют друг друга, позволяя повысить надежность системы.

Но сначала зададимся вопросом, зачем же аппаратная защита нужна, если есть обычная, программная?

Программная обфускация пока что обладает только такими эвристическими методами, как запутывания потока, переименования переменных и т.п. Криптография может использоваться только очень ограниченно. В то время как аппаратная защита способна использовать криптографию в полном объеме, потому что может размещать важные блоки в сложнодоступном для взлома месте — в железе. В целом, можно сделать такой вывод:

Преимущества:

- Можно оценить степень защиты (в отличие от полностью программных подходов, где на данный момент нет удовлетворительных результатов в области оценок защиты).
- Как правило цена взлома может быть напрямую выражена в затратах на покупку специальных дорогостоящих устройств или затратах при использовании лабораторий.
- Существующие решения, полностью исключают внутренние методы атаки, то есть со стороны других программ или операционной системы.

Недостатки:

- Сложность разработки высока (как следствие высока стоимость конечного устройства и стоимость разработки).
- Ограничение на распространение (некоторые программно-аппаратные структуры могут применяться только во встроенных системах, другие не поддерживают многозадачность).
- При наличии соответствующей аппаратуры можно взломать любую программно-аппаратную систему.

Необходимо отметить существование работ направленных на устранение последнего недостатка (возможность взлома сколь угодно сложной системы). Например, статья [Yuval03], рассматривает интегральную схему, как набор элементарных логических единиц (вентилей И, НЕ) и соединительных линий. Далее определяется t-adversary — противник, который на каждом такте может наблюдать не более t линий, соединяющих элементарные логические единицы. В статье приводится алгоритм запутывания, работающий против t-adversary (алгоритм зависит от t).

Качество алгоритма оценивается числом дополнительных вентилей, добавляемых в микросхему. Необходимо заметить, что число дополнительных логических единиц довольно велико (как минимум линейно относительно произведения числа вентилей в исходной микросхеме на t).

3. eXecute Only Memory

XOM (eXecute Only Memory) — модель (надстройка над процессором), где программа исполняется, но не может глубоко анализироваться даже из операционной системы и каким-либо программным путем. Обладает свойствами, которые мы опишем ниже. Поддерживает криптографию на аппаратном уровне, допускает многозадачные среды.

XOM не является целым вычислительным устройством, следовательно его разработка не настолько дорога, как разработка целого процессора. Также, возможна его интеграция с уже существующими процессами, хотя ни одной известной практической реализации XOM еще нет.

Достаточно полное описание XOM можно прочитать в [Lie00] и [Lie04]. Тут мы остановимся на основных принципах.

3.1. Основные принципы XOM

XOM являясь надстройкой над процессором, читает пошагово программу и передает ее на исполнение, обеспечивая нужный уровень защиты исходного кода и данных в памяти. Рассмотрим все его действия поподробнее.

Каждый XOM-процессор имеет свой **открытый и закрытый ключи**, которые присваиваются ему при изготовлении. Все **инструкции программы изначально (автором) зашифрованы** открытым ключом XOM'а (несущественные для защиты куски могут быть не зашифрованы вообще). Хотя, для эффективности открытым ключом можно шифровать и только начальный блок программы, содержащий симметричный ключ (compartment key), которым шифруется оставшаяся часть (симметричное шифрование намного быстрее). **Никто не может расшифровать программу не зная закрытого ключа.**

XOM расшифровывает нужную инструкцию и посылает ее на исполнение процессору. Далее расшифровывается следующая необходимая инструкция и так далее... Но, так как программа постоянно должна оперировать с данными, а не только исполнять свой код, то встает проблема **сохранения** этих **данных в незащищенную область памяти**. Это решается

также с помощью стандартных методов криптографии. Данные, записываясь в память, шифруются, подписываются вместе с адресом ячейки и идентификатором исполняемого процесса (определяемого по compartment key). При чтении данных из памяти программой, они расшифровываются, их идентификаторы сравниваются, адрес ячейки проверяется. Если что-то не совпадает, то программа-читатель блокируется. Никакой другой процесс не может расшифровать данные или скопировать их в другую ячейку, не зная уникального идентификатора.

Также нужно уметь **сохранять регистры**, чтобы поддерживать переключение контекста (context switch). Это делается по аналогии с хранением данных в памяти: **все регистры** подписываются вместе с идентификатором процесса, **шифруются** и записываются в память. А при переключении обратно к этому процессу, регистры расшифровываются и проводятся нужные проверки правильности. Против атаки повторениями (replay attack) **используются постоянно сменяющиеся ключи шифровки регистров** — последний используемый ключ запоминается для каждого процесса. Против этой же атаки над данными, записываемыми в память, надежной защиты нет, так как там генерировать каждый раз новый ключ не удастся. Ведь данные используются постоянно, а не один раз записали все сразу — один раз прочитали, как регистры.

XOM — криптографическая система с доказуемой секретностью, в отличие от многих программных методов обфускации. Ниже в статье, после рассмотрения DAT, будет дан более развернутый вывод про оба алгоритма сразу.

3.2. Критика

XOM предоставляет защиту, близкую к black box security, потому что полностью скрывает исходный код программы, не предоставляя возможностей для его статического анализа. То есть, без низкоуровневого вмешательства в XOM невозможно расшифровать код программы и ее данные (естественно, при стандартных криптографических предположениях).

Конечно, на практике достаточно будет сломать одну машину (аппаратно), узнать ее секретный ключ и расшифровывать программы под нее. После этого опубликовывать взломанные программы в открытом виде. То есть, тут необходим некий централизированный контроль за взломанными машинами.

Например, использование водяных знаков в программе — для какой именно машины она поставлялась. И если эта программа опубликовывается в открытом виде, то соответствующий XOM-процессор оказывается в черном списке для разработчиков ПО и для него больше не поставляются программы.

Поэтому, остается открытым вопрос, как раздавать ключи XOM-ам.

Недостатки:

- каждому свой — очень трудоемко и неудобно создателям ПО, сложности с защитой;
- всем один — никакой выгоды в защите;
- зонально (как DVD), партиями и т.д.

4. Динамическая таблица адресов

Опираясь на криптографию, мы получили систему, полностью защищенную от внутренних атак и некоторых внешних атак наблюдения (например, получение информации из памяти). Тем не менее, осталась еще одна уязвимость, которой можно воспользоваться, не прилагая особых денежных усилий. Не смотря на то, что вне процессора все данные шифруются, адреса памяти, по которым расположены эти данные, остаются теми же. Иными словами, взломщик может следить за потокам управления и потоком данных, отслеживая адреса появляющиеся на шине. Исследование последовательности адресов может дать достаточно подробную структуру о графе потока управления (выдать структуру программного обеспечения). Именно этой проблеме посвящена статья [Xiatong04]. Рассмотрим эту проблему поподробнее:

4.1. Утечки

Определение 1. *Layout leakage — получение данных о линейных участках кода благодаря принципу последовательного выполнения (две последовательные команды имеют два последовательных адреса).*

Определение 2. *Recurrence leakage — получение данных за счет принципа обращения к памяти (для двух обращений, в процессе выполнения, к одной и той же переменной на шине дважды появляется один и тот же адрес), с помощью этого типа «утечек» можно восстановить структуру циклов, уловных выражений.*

4.2. Статическая таблица адресов

Решения для первого типа утечек в следующем: достаточно сделать в процессоре статическую таблицу трансляции адресов (SAT static address table). SAT логически располагается между шиной и процессором, так, что любой адрес памяти который процессор выдает на шину проходит через преобразование SAT. Статическая таблица трансляции — абсолютно прозрачная структура, логически представляющая перестановку адресов памяти. SAT можно представить себе как виртуальную память (где любой линейный адрес проходит прозрачно через страничное преобразование и выдается на шину), с той разницей, что SAT статична (задается при производстве процессора и не может быть изменена в дальнейшем). Необходимо отметить, что при таком решении возникают трудности для разработчиков ПО. А именно, разработчик должен «знать», каким образом записать ПО в память («знать» в том смысле, что либо он должен знать функцию перестановки, которую реализует SAT, либо ему должно быть предоставлено устройство, которое запишет приложение по правильным адресам в физической памяти). Начальная запись кода и данных по правильным физическим адресам — единственная проблема, после того как приложение записано в память оно работает обычным образом (фактически можно пользоваться прежними средами разработки, правильно записанное приложение не «заметит» SAT).

Существуют некоторые промышленные чипы, реализующие этот подход (например, DS5000, DS5002FP производства Dallas Semiconductor описаны в [Dallas]). Авторы статьи [Xiatong04] отмечают слабость такой защиты (информация о возвратах к одному и тому же адресу по-прежнему легко доступна). Таким образом, основной интерес для нас, представляет борьба с recurrence leakage...

4.3. Динамическая таблица адресов

Идеи измененного алгоритма: дополнить каждое чтение из памяти записью в память (теперь взломщик будет почти всегда видеть пары циклов чтение/запись), заменить SAT на DAT (dynamic address table — в некотором смысле виртуальная память, так как теперь позволено менять таблицу трансляции), пытаться перемешивать блоки памяти на протяжении работы процессора (при этом корректируя DAT).

Итак, помимо таблицы адресов (изначально тождественной) в системе присутствует небольшая буферная память, которая дополняет основную (Shuffle Buffer). То есть блок может находится либо в нем, либо в оперативной памяти (но не одновременно в двух местах). Алгоритм делится на две стадии:

- **стадия разгона**
- **стадия работы**

На стадии разгона происходит линейное заполнение Shuffle Buffer прочитанными блоками (в DAT помечается, тот факт, что блок находится в Shuffle Buffer, а не в памяти). На стадии работы для любого прочитанного блока происходит случайный выбор блока из буфера, который меняется с ним местами (из буфера записывается в основную память и наоборот). Далее происходит корректировка DAT.

Авторы DAT (Xiatong Zhuang, Tao Zhang, Hsien-Hsin S. Lee, Santosh Pande) приводят вероятность $1/M^n$ для обнаружения атаки n-recurrence (обнаружение n повторений одной команды), где M — количество блоков, находящихся одновременно в Shuffle Buffer.

4.4. Критика

DAT-таблица и буфер по своим размерам не могут полностью содержаться внутри процессора, тогда часть таблицы будет храниться в памяти (авторы рассматривают такой вариант, но не рассматривают как это отразится на «утечках» информации, ведь часть таблицы DAT находящаяся в памяти не будет проходить через Shuffle Buffer). То есть необходимо усовершенствование таблицы и методов ее хранения. Хотя, если оценка $1/M^n$ правдоподобна, то тогда достаточно буфера из всего 512-1024 команд, чтобы даже обнаружение 2-recurrence уже было маловероятно.

5. XOM + DAT

Попробуем совместить два описанных подхода и оценить получаемое преимущество.

Архитектура XOM гарантирует стойкость против попыток узнать что именно исполняется на процессоре, в то время как DAT против исследования потока управления программы. То есть, они могут гармонично дополнять друг друга, если не будут конфликтовать.

Если расположить Shuffle Buffer между ядром процессора и XOM-надстройкой, тогда данные, содержащиеся в нем не будут шифроваться, что может быть плохо для стойкости. Следовательно, разместим буфер после всего процессора. Динамическая таблица адресов будет располагаться, как и в классической модели, прямо в процессоре. Тогда, жизненный цикл программы будет выглядеть так. Сначала с помощью закрытого ключа XOM-процессора расшифровывается compartment key программы, ей присваивается XOM ID. Затем, программа последовательно исполняется, причем при обращении в память адреса транслируются с помощью DAT, и во время исполнения некоторые ячейки в памяти меняются с ячейками в Shuffle Buffer в случайном порядке. То есть, эти две архитектуры работают прозрачно для друг друга, не замечая каких-то изменений.

Рассмотрим стойкость против взлома. XOM нам гарантирует секретность выполняемых процессором инструкций, DAT — секретность потока исполнения. То есть, если раньше (при использовании одного метода) можно было узнав одно (например поток исполнения), с помощью него узнать другое (например что исполняется — зная некоторые стандартные библиотечные последовательности инструкций для функций, можно их обнаружить расположенными подряд в памяти). То теперь это исключено, и программа будет полноценно защищена на архитектурном уровне. В результате сложность взлома сильно возрастает, а цена изготовления устройства является лишь суммой цены XOM и DAT.

Попробуем применить атаку пошаговым исполнением к полученной системе. Например, мы исполняем программу до какой-нибудь явно известной и обнаруживаемой инструкции (например вывести символ на экран). При этом мы узнаем, как выглядит зашифрованная инструкция вывода и где хранятся интересные нам данные в памяти. Дает ли это нам хоть какую-то информацию о программе, ее ключе, содержимом таблицы адресов или буфера? Очевидно, что если и дает, то намного меньше чем при программной обфускации. Из-за применения в архитектуре стандартных криптографических алгоритмов (как симметричных, так и несимметричных) полученная информация никак не поможет нам расшифровать или понять оставшуюся часть программы.

6. Заключение

По сравнению с программной обфускацией на нынешнем уровне, аппаратная защита более перспективна. Обфускация пока предоставляет только практические приемы, как, например, запутывание потока, переименование переменных и т.п., что не является гарантией защищенности программ. И, если необходимо узнать, где в программе выполняется определенный кусок, это в большинстве случаев можно сделать несложным пошаговым исполнением инструкций. Аппаратные же методы дают гарантированную защиту, а в сочетании XOM + DAT и поток, и данные оказываются надежно скрыты.

Литература

[Best79] *Best R* "Microprocessor for Executing Encrypted Programs"US Patent 4,168396 September 1979.

[Kent80] *Kent S.T* "Protecting Externally Supplied Software in Small Computers"Ph.D. Thesis MIT/LCS/TR-255 1980.

[Yuval03] *Yuval Ishai, Amit Sahai, David Wagner* "Private Circuits: Securing Hardware Against Probbing Attacks"2003.

[Dallas] "DS5002FP Secure microprocessor chip data sheet", Dallas Semiconductor.

[Lie00] *D.Lie, C.A.Thekkath, M.Mitchell, P.Lincolny, D.Boneh, J.Mitchell, M.Horowitz* "Architectural Support for Copy and Tamper Resistant Software", 2000.

[Lie04] *D.Lie, J.Mitchell, C.A.Thekkath, M.Horowitz* "Specifying and Verifying Hardware for Tamper-Resistant Software", 2004.

[Xiatong04] *Xiatong Zhuang, Tao Zhang, Hsien-Hsin S. Lee, Santosh Pande* "Hardware Assisted Control Flow Obf uscation for Embedded Processors", 2004.

Об особенностях применения методов обфускации программ для информационной защиты микроэлектронных схем

Н.П. Варновский, В.А. Захаров, Н.Н. Кузюрин, А.В. Чернов, А.В. Шокуров

Аннотация. Данная статья посвящена проблеме обеспечения информационной безопасности микроэлектронных схем. В первой части статьи проведен анализ технологической цепочки проектирования и изготовления микроэлектронных схем. На основе проведенного анализа выделены те этапы проектирования, на которых целесообразно применение специальных методов информационной защиты проектных решений. Для каждого из выделенных этапов проектирования сформулирована задача информационной защиты схемы и устанавлены требования, предъявляемые к методам решения этой задачи. Во второй части статьи дается краткое описание наиболее распространенных методов обфускации программ. Далее мы оцениваем, в какой степени эти методы могут быть пригодны для информационной защиты микроэлектронных схем на разных этапах проектирования.

Современная высокотехнологичная индустрия очень сильно зависит от уровня развития микроэлектроники. Микроэлектронные схемы широко используются во многих отраслях промышленности, в системах государственного управления, в современных видах вооружения, и поэтому качество используемых схем во многом определяет успешную работу в этих сферах деятельности. Не менее важной является проблема обеспечения безопасности (информационной, технологической и пр.). Это относится, в частности, и к технологии производства микроэлектронных схем: процесс проектирования микроэлектронных схем может быть отделен от процесса их производства, и в силу этого возникают серьезные проблемы, связанные с предотвращением утечки секретной информации и защитой проектных решений от несанкционированного вмешательства на этапе изготовления микросхем.

Главная специфика указанных проблем состоит в том, что традиционные организационно-технические процедуры обеспечения безопасности не могут в полной мере гарантировать их решение — некоторые звенья технологической цепочки производства микроэлектронных схем будут неизбежно оставаться открытыми для несанкционированного постороннего вмешательства.

В связи с этим возникает необходимость создания таких технологий и инструментальных средств обеспечения информационной защиты проектных решений, которые позволяли бы предотвратить извлечение секретной информации из открытого описания проектного решения микросхем на этапе их изготовления. В основу этих технологий могут быть положены математические методы защиты проектных решений, наподобие тех, которые успешно зарекомендовали себя в криптографии.

Анализ технологической цепочки проектирования и производства схем показывает, что все возникающие здесь проблемы защиты информации можно разделить на две категории. В первую входят задачи обеспечения конфиденциальности и целостности информации, передаваемой по каналам связи, а также информации, хранимой в памяти компьютеров (как оперативной, так и на внешних носителях). Под конфиденциальностью понимается недоступность информации для потенциального противника (злоумышленника). Целостность означает защиту информации от активного противника, который пытается внести в нее какие-либо искажения. Обеспечение конфиденциальности и целостности — основные и наиболее глубоко исследованные задачи в обширной междисциплинарной сфере, известной под названием «информационная безопасность». Для их решения разработаны и успешно применяются разнообразные криптографические средства. Исходя из современного состояния развития криптографии, математики и технологии, можно констатировать, что в целом эти задачи решены (см. [1, 2, 3, 4, 5]).

Ко второй категории относится специфическая проблема, которую необходимо решить для обеспечения информационной безопасности на этапе изготовления схем. Отличительная особенность задачи информационной защиты на этом этапе состоит в том, что описание изготавливаемой схемы (устройства) передается в открытом (незашифрованном) виде в неконтролируемую рабочую среду и вероятно станет доступным противнику. Анализ открытых публикаций, посвященных проблемам информационной безопасности, свидетельствует о том, что данная проблема, по-видимому, ранее не исследовалась.

Вместе с тем в последние годы интенсивно изучаются две смежные задачи обеспечения компьютерной безопасности, результаты решения которых, наряду с методами математической криптографии, могут быть использованы и для информационной защиты проектных решений при производстве микросхем.

Одна из этих задач — это задача обфускации программ (от англ. to obfuscate — затруднить понимание, скрыть смысл). Она заключается в разработке механизма, позволяющего сделать невозможным или чрезвычайно трудоемким извлечение из открытого текста программы той ключевой информации, которая позволила бы понять принцип ее устройства и затем при необходимости проводить с этой программой сложные преобразования, связанные с целенаправленным изменением ее структурных и функциональных характеристик. В целом ряде работ для обфускации [7, 8, 13, 14, 15, 16, 17, 18, 32, 33] предложены специальные преобразования программ, противодействующие алгоритмам статического анализа и декомпиляции. В отдельных случаях удалось провести строгое математическое обоснование стойкости предложенных преобразований [29, 25].

Другая задача, связанная с обеспечением безопасности мобильных агентов, заключается в разработке таких методов организации распределенных вычислений, которые позволяли бы защитить распределенную вычислительную систему от противника, получившего неограниченный доступ к одному из ее компонентов. Несмотря на то, что эта задача давно известна в теории вычислительных сетей, имеется сравнительно немного работ, в которых проводится ее систематическое исследование с использованием математических методов.

Для того, чтобы оценить, в какой мере известные методы решения задачи программного обеспечения могут быть применены для информационной защиты микроэлектронных схем, мы провели анализ технологической цепочки проектирования и изготовления микроэлектронных схем. В результате удалось выделить этапы проектирования, на которых целесообразно применение тех или иных методов информационной защиты проектных решений, а также установить требования, предъявляемые к этим методам. На основе этих требований мы проводим краткий обзор методов обфускации программ, используемых для решения ряда проблем защиты информации, возникающих при разработке и распространении компьютерных программ. Основное внимание уделяется оценке перспектив применения и развития указанных методов для решения задач информационной безопасности в процессе проектирования и производства микроэлектронных схем.

1. Типичный маршрут проектирования микроэлектронных схем

Вначале мы рассмотрим типичный сценарий проектирования микроэлектронных схем (СБИС). Во многих системах автоматизированного проектирования СБИС все инструментальные средства проектирования интегрированы в единый маршрут от системного уровня описания схемы до логического и топологического уровней. Для того, чтобы правильно оценить возможности применения методов информационной защиты схем, необходимо проанализировать все этапы цикла проектирования схемы и на основании этого анализа выбрать те этапы проектирования, на которых эффективность применения обфускации будет наибольшей. Рассмотрим этапы проектирования СБИС на примере САПР Synopsys.

Описание СБИС при проектировании представляется на разных уровнях абстракции, в соответствии с которыми выделяются уровни проектирования и входящие в них этапы маршрута проектирования. Маршрут проектирования Synopsis включает в себя три основных уровня:

1. Системный уровень, включающий этапы алгоритмического (или функционального) и архитектурного проектирования и состоящий из трех этапов:

    - Этап 1: создание алгоритмической модели и ее верификация;
    - Этап 2: создание модели уровня транзакций (модели макроархитектуры) и ее верификация;
    - Этап 3: создание т.н. «золотой модели» СБИС как технического задания для проектирования аппаратной и программной части схемы.

2. Логический, или вентильный, уровень, состоящий из пяти этапов:

    - Этап 1: создание синтезируемой RTL-модели системы на кристалле на одном из языков описания аппаратуры (Verilog, VHDL);
    - Этап 2: логическая верификация RTL-модели посредством моделирования;
    - Этап 3: физическая верификация посредством прототипирования;

- Этап 4: синтез в базисе вентильных схем и автоматическая генерация тестовых структур для контроля годности;
- Этап 5: логическая и формальная верификация списка цепей.

3. Топологический уровень.

Для каждого из уровней реализуется идея «горизонтальной» методологии проектирования, или «спиралевидного» маршрута, когда за каждым этапом проектирования следует этап верификации, и при этом интегрируемость средств проектирования обеспечивает верификацию с использованием описаний на разных уровнях абстракции, а также дает возможность при необходимости совершать переход от одного уровня к другому. Системный уровень проектирования включает два подуровня — алгоритмический подуровень и подуровень транзакций. Проектирование СБИС — это многостадийный процесс, в который вовлечено несколько соисполнителей. Это обстоятельство повышает вероятность ошибки, обнаружить которую можно уже только после изготовления микросхемы. Поэтому в процессе проектирования большое значение приобретают так называемые исполняемые спецификации проекта — виртуальные модели разрабатываемой системы на разных уровнях абстракции. Эти абстрактные модели детализируются при переходе с одного уровня на другой и принимают завершенное описание схемы на топологическом уровне проектирования. При этом на каждом уровне проектирования разработчики, имея дело с моделью системы соответствующего уровня абстракции, должны иметь возможность сравнивать то, что у них получается, с исходными требованиями к будущей СБИС. Для этой цели абстрактные модели представляются в виде исполняемых спецификаций, т.е таких описаний, которые позволяют проверять функциональные характеристики системы. Современные САПР позволяют изначально создавать исполняемую модель системы (на алгоритмических языках высокого уровня, без привязки к конкретной реализации), постепенно детализировать ее, заменять более абстрактные блоки сначала архитектурными моделями (на уровне транзакций), а затем и конкретными описаниями аппаратуры — на RTL-уровне, на вентильном уровне, и.т.д. Такие виртуальные модели фактически являются техническим заданием для разработчиков СБИС. В процессе разработки СБИС модель постепенно детализируется, но при этом всегда можно убедиться, что по мере разработки не изменяется заданная функциональность исходной модели. Существуют исполняемые спецификации различного уровня абстракции в соответствии с этапами проектирования системы на кристалле:

- модель уровня алгоритмов (описание алгоритма на языке программирования высокого уровня, например, C или C++);
- модель уровня транзакций, макроархитектуры, создаваемая на базе алгоритмической модели инкрементально (язык System-C);
- модель RTL-уровня, микроархитектуры (язык Verilog, VHDL);
- модель логического вентильного уровня.

На первом этапе проектирования создается алгоритмическая модель системы. Эта модель представляет собой алгоритм функционирования схемы, описанный на языке программирования высокого уровня (например, C). С помощью этой модели можно проверить принципиальную работоспособность выбранного алгоритма и провести его необходимую отладку. На втором этапе на основе описания алгоритма функционирования системы создается модель макроархитектуры, или модель уровня транзакций. При проектировании этой модели исследуются различные варианты ее архитектуры с учетом возможностей аппаратной и программной реализации различных функций. Для описания этой модели на уровне транзакций используется более специализированный, чем C, язык, такой, как System-C. Модель, описанная на языке System-C, позволяет выполнить такие исследования, как анализ потоков данных между блоками системы, оценка эффективности использования памяти, влияние времени считывания/записи и т.д. В основе языка System-C лежит C++, и поэтому System-C модель можно разрабатывать на базе существующей C-модели, дополняя ее конструкциями, отражающими архитектуру системы. После окончательного выбора архитектуры системы, доводки параметров системы на базе как библиотечных, так и сконструированных пользователем моделей, созданная System-C модель передается разработчику СБИС в качестве исполняемой спецификации макроархитектурного уровня. Она сопровождается наборами тестов и генераторами псевдослучайных тестовых потоков для регрессионного тестирования, т.е. по сути дела является техническим заданием для разработчиков СБИС.

Окончание верификации модели макроархитектурного уровня знаменуется созданием «золотой модели» СБИС. На ее основе оформляется документация, которая становится техническим заданием для дальнейшего проектирования аппаратуры и программного обеспечения. На первом этапе логического уровня проектирования на основе «золотой модели» создается RTL-модель СБИС, описанная на языках типа Verilog или VHDL. Вся

работа ведется в основном вручную с использованием библиотечных модулей. Если System-C модель была составлена из системных блоков, то получить RTL-описание не составляет труда, поскольку для библиотечных модулей имеются параметризованные RTL-описания. Если же System-C модель создавалась вручную, то трансляция макроархитектурного описания системы в RTL-модель проводится вручную. На втором этапе логического уровня проектирования проводится логическая верификация посредством моделирования RTL-описания, созданного на этапе 1. Верификация выполняется с использованием универсальных инструментальных средств, способных моделировать смешанные описания Verilog/System-C/C++/.

Далее проводится физическая верификация проекта посредством прототипирования. Прототип создается из того же описания, которое разрабатывается для СБИС, поскольку любые изменения исходных кодов связаны с риском появления ошибок. На четвертом этапе проводится синтез в базисе вентильных схем и автоматическая генерация тестовых структур. В процессе этого синтеза схема оптимизируется сразу по нескольким параметрам, включая сложность, энергопотребление и т.д. В результате синтеза строится список цепей (netlist), полностью определяющий логическую структуру схемы. На завершающем этапе логического синтеза проводится верификация проекта, анализ временных статических характеристик, оценка влияния вариаций технологического процесса и выявление скрытых ошибок. На этом же этапе вырабатываются фабричные тесты для контроля годности СБИС. На топологическом уровне синтеза системы проводится размещение элементов на кристалле и трассировка. Результатом деятельности на этом этапе проектирования является описание топологии системы на кристалле. Это описание содержит все необходимые данные для производства микросхемы. На топологическом уровне проводится анализ правильности электрической схемы (т.е. правильности соединений, наличия неподключенных входов и выходов и т.п.) и анализ временных параметров. Проводится также проверка топологии путем восстановления из топологического описания списка цепей и сравнения его с исходным списком. Таким образом, на протяжении проектирования СБИС последовательно строятся 5 описаний системы:

- Алгоритмическое описание системы (на языке C/C++);
- Макроархитектурное описание системы (на языке System-C);
- Микроархитектурное (RTL) описание системы (на языках VHDL или Verilog);

- Описание списка цепей (на языках VHDL или Verilog и в форматах IEEE PDEF, DEF, LEF);
- Описание топологии СБИС (например, в формате GDSII).

При этом:

- переход от алгоритмического описания к макроархитектурному осуществляется вручную с применением системных библиотек и вспомогательных инструментальных средств, облегчающих моделирование и верификацию;
- переход от макроархитектурного описания к RTL-описанию проводится в основном вручную с использованием стандартных библиотечных блоков;
- синтез списка цепей по RTL-описанию системы проводится автоматизированно;
- синтез описания топологии для заданного списка цепей проводится автоматизированно;
- восстановление списка цепей из описания топологии также проводится автоматизированно.

2. Задачи информационной защиты схем на маршруте проектирования

На основании описанного выше типичного маршрута проектирования систем на кристалле можно уточнить постановку задачи информационной защиты схем в процессе их проектирования. Приведем ряд следующих положений, уточняющих те понятия, в терминах которых определяется задача обфускации.

Противник. Предполагается, что описание топологии проектируемой схемы представлено в одном из стандартных форматов (например, GDSII), и противник имеет неограниченный доступ к этому описанию. При этом общедоступные инструментальные средства позволяют противнику однозначно восстановить из описания топологии проектируемой системы ее логическое описание в виде списка вентильных цепей (netlist). В свою очередь, по списку вентильных цепей можно восстановить одно из возможных RTL-описаний системы. Поскольку несколько различных RTL-описаний

могут привести к одному и тому же списку цепей, однозначно восстановить исходное RTL-описание проектируемой системы невозможно. Помимо доступа к топологическому описанию системы, противник может обладать следующими данными о проектируемой системе:

- знание фабричных тестов, используемых для проверки годности изготавливаемой СБИС;
- знание о возможном предназначении изготавливаемой СБИС;
- знание о методах информационной защиты, примененных в ходе проектирования системы.

Последнее предположение означает, что противнику известны все алгоритмы и приемы, используемые проектировщиками для обеспечения информационной безопасности проектного решения, но при этом противнику неизвестны переменные параметры этих алгоритмов (следуя криптологической терминологии, эти параметры мы будем называть ключами). Предполагается также, что противник не имеет доступа к описанию проектируемой системы ни на каком другом уровне, кроме уровня топологии.

Угрозы. Можно выделить несколько угроз, как общего, так и специального вида:

1. *Раскрытие предназначения изготавливаемой СБИС.* Можно предполагать, что противнику известно о том, что изготавливаемая СБИС предназначена для решения одной из задач $T_1, \ldots, T_N$, и алгоритмическое описание проектируемой системы позволяет легко понять, для какой из перечисленных задач предназначена соответствующая СБИС. Таким образом, секретной информацией здесь является та задача T_i, для решения которой предназначена СБИС и угроза проявляется в стремлении противника распознать по описанию топологии СБИС эту задачу.

2. *Раскрытие секретных данных, содержащихся в изготавливаемой СБИС.* Здесь предполагается, что противнику известно предназначение СБИС, и известно также, что в СБИС реализован алгоритм $Alg(p_1, \ldots, p_N)$, параметризованный относительно некоторого множества данных (констант, коэффициентов). Именно эти параметры $p_1, \ldots, p_N$ и составляют секретную информацию, и угроза противника проявляется в стремлении распознать эти параметры известного

алгоритма по описанию топологии СБИС, реализующей этот алгоритм.

3. *Раскрытие алгоритма (функциональности), реализуемого изготавливаемой СБИС.* Здесь также предполагается, что противнику известно предназначение СБИС, но неизвестен алгоритм и функция, которые ей реализуются. Именно они и составляют секретную информацию.

Методы. Предполагается, что противник может применить любое из известных в настоящее время средство анализа программ или описаний СБИС, или разработать новое средство анализа, опираясь на известных в настоящее время алгоритмах и методах. Единственным ограничением при проведении атак считается неспособность противника взламывать криптографические системы определенного вида и решать те математические задачи, которые положены в основу этих криптосистем.

Таким образом, задача обфускации схем состоит в разработке таких преобразований описаний схемы на некоторых этапах ее проектирования, в результате применения которых противник, располагающий определенными знаниями о проектируемой СБИС (описанием топологии и набором фабричных тестов), ее возможном предназначении и применяемых методах информационной защиты, не сможет разработать ни одного метода для осуществления той или иной угрозы.

3. Обфускация компьютерных программ

Задача обфускации программ заключается в разработке механизма, позволяющего совместить такие взаимоисключающие качества компьютерных программ как открытость и защищенность. Открытость программы подразумевает свободный и потенциально неограниченный доступ к тексту программы, дающий возможность анализировать ее устройство, проводить с программой произвольные эксперименты и преобразования. В свою очередь, свойство защищенности программы предполагает невозможность или чрезвычайно высокую трудоемкость извлечения из текста программы той ключевой информации, которая позволила бы «осознать» программу, понять принцип ее устройства и функциональность, и затем при необходимости проводить с этой программой сложные преобразования, связанные с целенаправленным изменением ее структурных и функциональных характеристик. Необходимость сочетать эти два требования — открытости и за-

щищенности — делает задачу обфускации программ весьма трудной. Впервые идея обфускации (целенаправленного запутывания) программ была высказана в основополагающей работе Диффи и Хеллмана [19], определившей основные идеи и направления развития современной криптографии. В этой работе отмечалась возможность построения систем шифрования с открытым ключом путем обфускации криптосистем с секретным ключом. Обфускирующие преобразования программ имеют широкую область потенциального применения в программировании, криптографии, телекоммуникации (см., например, [6, 14, 17, 23, 32]). Разработка обфускирующих преобразований программ базируется на следующих теоретических и практических предпосылках:

1. Задачи анализа компьютерных программ относятся к числу вычислительно трудных задач, для которых либо вообще не существует общих методов (алгоритмов) решения, либо эти методы требуют затраты чрезвычайно больших вычислительных ресурсов;

2. Для обфускации программ можно использовать развитые математические методы криптографии;

3. Практика программирования показывает, что задача понимания программ сама по себе является очень трудной, и поэтому в простейшем случае обфускация программ достигается за счет отказа от использования средств, облегчающих понимание программы.

Различают два основных типа обфускирующих преобразований программ — обфускацию с высоконадежным устройством (trusted hardware) и обфускацию с неограниченным доступом к программе.

Обфускация с высоконадежным устройством предполагает наличие надежного вычислительного устройства, которое недоступно противнику. Это устройство имеет небольшую вычислительную мощность, позволяющую проводить лишь простые секретные вычисления. При обфускации с неограниченным доступом текст программы и все проводимые ею вычисления открыты для противника. Обфускация с неограниченным доступом к программе обладает очевидными преимуществами по сравнению с обфускацией с использованием высоконадежного вычислительного устройства, однако, реализация этой идеи и обоснование стойкости обфускирующих преобразований являются гораздо более трудными задачами.

Прежде всего, следует отметить, что в работах [6, 20, 30] была доказана невозможность построения «универсальных совершенных» обфускаторов,

которые позволяли бы скрыть от противника все сведения о произвольной защищаемой программе P. «Универсальным совершенным» обфускатором считается программа (компилятор) O, которая преобразует всякую программу P в программу $O(P)$, обладающую свойством «виртуального черного ящика»: любой противник, анализирующий текст программы $O(P)$, добьется не лучшего результата, чем некоторый тестирующий алгоритм, проводящий эксперименты с программой $O(P)$ как с «черным ящиком» (т.е. тестируя только входы-выходы). Эти результаты распространяются на широкий класс программ P, включая те из них, которые соответствуют комбинационным схемам небольшой глубины.

Тем не менее, упомянутые результаты представляют скорее академический интерес, поскольку требования применимости ко всем программам и сокрытие всех свойств программы, а также требование неотличимости от «черного ящика», представляются чрезмерными и интуитивно неосуществимыми. На практике, же интерес представляет защита классов программ и конкретной информации в них содержащейся. На этом пути удается получить ряд положительных результатов. Так для отдельных классов программ удается доказать возможность стойкой обфускации некоторых свойств: в работе [29] (а затем и в работах [25, 31]) был предложен и обоснован стойкий метод обфускации программ, в которых используется простая схема проверки ключа для идентификации пользователей.

В большинстве работ, посвященных обфускации с неограниченным доступом к программе, авторы ограничиваются описанием тех или иных методов обфускации, но не приводят строгого обоснования их стойкости, — предполагается, что метод обфускации обладает достаточной стойкостью, если возникающая в результате его применения задача декомпиляции программ будет вычислительно трудной. С точки зрения современных требований к защите информации, принятых в криптографии, такой уровень стойкости обфускирующих преобразований считается слабым.

Тем не менее, эти методы вызывают определенный интерес, поскольку их применение действительно затрудняет понимание обфускированных программ и существенно понижает эффективность современных средств декомпиляции и анализа программ. В большинстве работ (см., например, [7, 8, 15, 16, 17, 28, 32]) для обфускации используются сложные структуры данных, порожденные переменными-указателями. Стойкость этих методов обусловлена тем фактом, что современные алгоритмы анализа диапазонов значений переменных указателей не обладают достаточной точностью. Более строгий метод оценки степени непроницаемости обфускированных

программ для автоматических систем анализа программ предложен в [21]. В работах [10, 33, 24] предложены методы обфускации, направленные на противодействие автоматическим системам декомпиляции (дизассемблирования) машинных кодов. В настоящее время создано или находятся в процессе разработки несколько систем обфускации программ, в которых применяются описанные выше методы, но ни одна из них еще не получила достаточно серьезного признания.

4. Применимость обфускации к описаниям схем на разных уровнях проектирования

Вообще говоря, обфускации может быть подвергнуто описание схемы на любом из перечисленных уровней проектирования. Однако для правильного выбора того этапа, на котором целесообразно проводить обфускирующие преобразования, необходимо учесть ряд следующих факторов:

- (*устойчивость*) Обфускирующие преобразования, проведенные на начальных уровнях проектирования не должны быть «разрушены» при уточнении описаний системы на последующих уровнях. Следует помнить о том, что противник может иметь доступ только к топологии схемы, и поэтому эффект обфускации должен быть непременно отражен на уровне описания топологии.
- (*безвредность*) Обфускирующие преобразования не должны служить источником дополнительных ошибок при проектировании СБИС.
- (*трудоемкость*) Обфускирующие преобразования не должны чрезмерно усложнять процесс проектирования СБИС.

Учитывая перечисленные факторы, особенности описаний системы на каждом этапе проектирования, а также методы детализации описаний при переходе от одного уровня проектирования к другому можно выработать следующие рекомендации по применению обфускирующих преобразований к описаниям проектируемой схемы.

1. *Проведение обфускации описания топологии схемы недопустимо.* Технология проектирования СБИС требует строгого соответствия описания списка цепей netlist-1, который используется в качестве исходных данных при проектировании топологии, и списка цепей netlist-2, который автоматически восстанавливается на основе построенного описания топологии. При нарушении этой технологиче-

ской нормы существенно повышается риск возникновения ошибок в изготовленной СБИС.

2. *Проведение глобальной обфускации алгоритмического описания схемы нецелесообразно.* Трансляция описаний схемы на начальных этапах проектирования выполняется вручную, и поэтому глобальная обфускация С-программы, служащей исходной спецификацией проектируемой системы, значительно затруднит работу архитекторов схемы при построении ее описаний на макроархитектурном и микроархитектурном уровнях. При этом также значительно усложнятся отладка и верификация этих описаний, и повысится риск внесения в ходе детализации описаний трудно отслеживаемых ошибок. Нужно иметь в виду и то, что некоторые обфускирующие преобразования, затрудняющие понимание С-программ, могут не оказывать никакого влияния на сложность анализа описания списка цепей схемы.

3. *Проведение как локальной, так и глобальной обфускации описания схемы на макроархитектурном уровне нецелесообразно.* В пользу этого вывода можно привести те же самые аргументы, что и в предыдущем случае. Кроме того, даже проведение локальной обфускации (на уровне отдельных блоков, отражающих архитектуру системы) приведет к тому, что в System-C модели сократится число библиотечных блоков. Поэтому построение RTL-описания потребует большого ручного труда, что увеличит трудоемкость проектирования и послужит источником новых ошибок. Эти ошибки будет трудно выявить, коль скоро обфускация скрывает от разработчика содержательные детали устройства и функционирования отдельных блоков.

Таким образом, можно выделить те этапы в маршруте проектирования СБИС, на которых целесообразно применение обфускирующих преобразований. Таковыми являются:

- Этап разработки алгоритмической модели. На этом этапе допустимо применение локальных обфускирующих преобразований, не изменяющих в целом структуры описания, но скрывающих некоторые особенности и детали реализуемого алгоритма.

  Обфускирующие преобразования в этом случае будут применяться к C/C++ программе, служащей исполняемой спецификацией разрабатываемой системы на верхнем уровне абстракции.

- Этап разработки логической схемы в базисе производителя микросхем на основе верифицированного RTL-описания системы. На этом этапе допустимо неограниченное применение обфускирующих преобразований, которые будут применяться к описаниям системы на языках Verilog и VHDL.

В данной статье мы ограничимся рассмотрением задачи обфускации схемы на этапе разработки ее алгоритмической модели. В последующей статье мы более подробно рассмотрим специальные обфускации, которые могут быть применены для информационной защиты схемы на этапе разработки ее логической схемы.

5. Методы обфускации схем на алгоритмическом уровне проектирования

Далее приводится описание и классификация базовых обфускирующих преобразований программ, описывается методология оценки их устойчивости и трудоемкости, и проводится анализ применимости обфускирующих преобразований программ для информационной защиты схем на алгоритмическом уровне проектирования.

Мы будем использовать следующую терминологию. *Обфускирующим преобразованием* называется любое функционально эквивалентное преобразование программ. Выполнение обфускирующего преобразования может потребовать применения некоторых алгоритмов анализа для выявления необходимых свойств программы. Набор обфускирующих преобразований и сопутствующих им алгоритмов анализа программы составляет *метод обфускации*. Под *трудоемкостью* метода обфускации мы будем понимать объем вычислительных ресурсов, необходимый для применения этого метода.

Помимо трудоемкости, мы будем оценивать также, насколько значительно обфускирующее преобразование влияет на структурные (топологические) и временные параметры схемы. Наиболее важной характеристикой структуры схемы является ее размер, представляющий собой суммарную длину всех цепей в списке netlist описания топологии схемы. Временной характеристикой служит быстродействие схемы (частота тактовых импульсов), которое зависит от максимальной длины цепей в списке netlist.

Таким образом, для каждого обфускирующего преобразования может быть определена его цена TC, зависящая от коэффициента увеличение размера схемы и уменьшения ее быстродействия в результате выполнения

этого преобразования. Метод обфускации называется *допустимым*, если его цена не превышает директивно установленный предел TC_{max}. Предел TC_{max} устанавливается, исходя из эксплуатационных требований к обфускированной схеме.

Наиболее важным (и в то же время наименее формализованным) показателем качества обфускирующих преобразований является *устойчивость* обфускации. Этот показатель призван оценивать, насколько применение обфускирующего преобразования усложняет «понимание» схемы, а также (учитывая специфику проектирования схем) в какой мере результат применения этого преобразования проявляется в описании топологии проектируемой схемы. Последнее требование обусловлено тем, что некоторые преобразования схемы на алгоритмическом уровне проектирования могут существенно изменить описание схемы на уровне алгоритмической модели, но при этом не повлекут заметного усложнения описания на логическом или топологическом уровне. Использованные здесь неформальные понятия «понимание схемы», «усложнение понимания» требует уточнения и, если возможно, формализации. Следует заметить, что термин «понимание программы» является ключевым в исследованиях по обратной инженерии программ. Хотя предложено много различных моделей процесса понимания [26], ни одна из них не предоставляет удовлетворительных формализмов, так как цель процесса обратной инженерии пока не удается формализовать. Это, по-видимому, является одной из причин неудовлетворительного состояния дел в области обратной инженерии. Нам представляется, что задача де обфускации программ является достаточно узким частным случаем общей задачи обратной инженерии, и для нее может быть предложена удовлетворительная формализация, рассматриваемая далее. Мы предполагаем, что полностью ручная деобфускация программ (схем) нецелесообразна из-за своей высокой стоимости. Обычно размер программы или описания схемы настолько велик, что полностью ручная деобфускация потребует слишком больших трудозатрат. Поэтому в дальнейшем мы будем рассматривать только автоматизированные методы деобфускации. *Деобфускирующим преобразованием* мы назовем любое функционально эквивалентное преобразование программы. Деобфускирующие преобразования могут требовать выполнения некоторых алгоритмов анализа программ. Набор из алгоритмов анализа программы и деобфускирующих преобразований, образует *метод деобфускации*.

Деобфускацию программы мы будем считать успешной, если полученная в результате ее применения программа (схема) оказывается достаточ-

но близка к исходной программе (схеме). Оценка степени близости требует введения расстояния между программами, а также порога устойчивости, задающего минимальное расстояние между исходной и деобфускированной программами. Мы будем считать, что для программ (схем), близких друг к другу в смысле определенной метрики, сложность их понимания и повторного использования сопоставима. Другими словами, деобфускация считается успешной, если после ее выполнения сложности понимания и повторного использования деобфускированной программы и исходной программы сопоставимы. Таким образом, метод обфускации будет считаться *устойчивым*, если для него не существует эффективной деобфускации, и при этом результат его применения отражается в описании схемы на топологическом уровне.

Для оценки устойчивости обфускации мы рассмотрим следующий набор деобфускирующих преобразований и сопутствующих им алгоритмов анализа:

1. Анализ графа потока управления (control-flow analysis) [27].
2. Выявление и устранение недостижимого кода (unreachable code elimination) [27].
3. Анализ достигающих определений (reaching definitions) и доступных выражений (available expressions) [27], совмещённый с консервативным анализом указателей.
4. Устранение мёртвого кода (dead-code elimination) [27].
5. Устранение общих подвыражений (common subexpression elimination) [27].
6. Устранение частичной избыточности (partial redundancy elimination) [27].
7. Анализ интервалов жизни переменных (liveness analysis) [9].
8. Статическая минимизация количества переменных (variable minimization) [27].
9. Продвижение констант (constant propagation) [27].
10. Продвижение копий (copy propagation) [27].
11. Анализ диапазона значений переменных (value range analysis) [22].
12. Построение и анализ покрытия дуг и базовых блоков [11].
13. Анализ инвариантных и полуинвариантных значений [12].

5.1. Лексическая обфускация.

Простейшим видом обфускирующих преобразований являются лексические преобразования [14]. Для выполнения лексических обфускирующих преобразований не требуется семантического анализа исходной программы. После применения лексических обфускирующих преобразований текст программы может существенно измениться и стать очень трудным для человеческого восприятия. Однако для компилятора исходная и обфускированная программа останутся эквивалентными, и для них будет сгенерирован один и тот же код. К числу лексических обфускирующих преобразований относятся такие преобразования, как

- *удаление комментариев*;
- *переформатирование программы*;
- *изменение идентификаторов.*

Названия этих преобразований исчерпывающим образом описывают их суть и назначение. Обфускирующие преобразования такого рода имеют малую сложность реализации, высокую скрытность и совершенно не усложняют программы. Именно поэтому большинство современных обфускаторов (Crema, VHDL Obfuscator и др.) ограничиваются применением только перечисленными выше обфускирующими преобразованиями. Главный недостаток обфускации лексики программы заключается в ее чрезвычайно низкой устойчивости, поскольку структура программного кода и данных практически не подвергается никаким изменениям.

Учитывая, что лексические изменения в описании схемы на языках высокого уровня, никак не проявляется в ни в описании схемы на микроархитектурном уровне (RTL-модели), ни в описании топологии схемы применение таких обфускирующих преобразований для информационной защиты проектных решений нецелесообразно. Помимо этого обфускация лексики небезопасны: они усложняет понимание описания схемы самими проектировщиками и служит, таким образом, дополнительным источником возможных ошибок при проектировании.

5.2. Преобразования управляющей структуры.

Обфускирующие преобразования управляющей структуры, рассматриваемые в данном разделе, модифицируют структуру потока управления программы. Часть этих преобразований используется в оптимизирующих компиляторах, другие преобразования разработаны специально для обфус-

кации программ. Вначале мы опишем преобразования, которые изменяют структуру программы как библиотеки процедур.

К этим преобразованиям относятся *открытая вставка процедур*, *выделение процедур*, *устранение библиотечных вызовов*, *переплетение процедур*, *клонирование процедур*. Остальные описываемые преобразования модифицируют процедуры независимо друг от друга.

Открытая вставка процедур. Данное преобразование заключается в подстановке тела процедуры в точку вызова этой процедуры. Преобразование открытой вставки является стандартным для современных оптимизирующих компиляторов. В них открытая вставка выполняется для того, чтобы, во-первых, избежать накладных расходов на вызов процедуры, а, во-вторых, расширить поле применения глобальных оптимизаций в процедуре. Обфускирующий потенциал данного преобразования состоит как в расширении поля применения преобразований, так и в увеличении размера процедуры, в уничтожении функциональных абстракций программы. Частным случаем этого преобразования является устранение библиотечных вызовов, которое заключается в том, что в программу добавляется собственная реализация библиотечных процедур. Как известно, стандартная библиотека всякого языка содержит реализации повсеместно используемых концепций, например, контейнеров. Поскольку имена и семантика стандартных процедур библиотеки известны, точки вызова библиотечных процедур легко могут быть прослежены, а это может дать ценную информацию для анализа программы. Для устранения этого слабого звена программа может содержать свой набор библиотечных процедур с семантикой стандартных процедур. Эти процедуры могут обфускироваться вместе со всей программой. Такое преобразование позволяет резко сократить количество вызовов внешних библиотек, сохранив только необходимый минимум (например, вызовы процедур ввода/вывода). Недостатком этого преобразования является увеличение размера программы.

Обфускирующее преобразование открытой вставки процедур имеет небольшую сложность реализации, поскольку его применение не предполагает проведения сложного анализа программ. Это преобразование можно применять к описанию схемы на алгоритмическом уровне проектирования. Преобразование открытой вставки процедур само по себе обладает невысокой стойкостью. Однако даже незначительные изменения (переименование локальных переменных, перестановка операторов, введение избыточных операторов и пр.), внесенные в подставленные процедуры, придают этому

преобразованию высокую скрытность и устойчивость. Преобразование открытой вставки процедур может привести к значительному увеличению размера обфускируемой схемы, а в случае устранения библиотечных вызовов и к усложнению ее проектирования. Поэтому интенсивное применение этого преобразования на алгоритмическом уровне нецелесообразно.

Переплетение процедур. Данное обфускирующее преобразование состоит в объединении двух или более процедур в одну новую процедуру. В результате объединения новая процедура будет сложнее, чем каждая из исходных. Кроме того, одна и та же процедура будет использоваться в программе для совершенно разных целей, что тоже затруднит анализ программы. Для того, чтобы несколько процедур могли быть объединены в одну процедуру, необходимо, чтобы они были достаточно похожи.

Преобразование переплетения процедур имеет небольшую сложность реализации. Накладные расходы по выбору соответствующего варианта процедуры можно отнести к накладным расходам на вызов процедуры. Для выполнения преобразования в обфускирующем компиляторе достаточно провести простейший семантический анализ программы. Преобразование переплетения процедур имеет высокую скрытность, поскольку обратное преобразование расщепления процедуры требует проведения глобального анализа программы. При проведении этого преобразования сложность и быстродействие схемы будет изменяться незначительно. Устойчивость преобразования переплетения процедур зависит от того, насколько похожи друг на друга переплетаемые процедуры — чем больше общих фрагментов имеют эти процедуры, тем выше устойчивость обфускации. Это обфускирующее преобразование может безопасно применяться на алгоритмическом уровне описания схемы, поскольку сложность и быстродействие схемы будет изменяться незначительно.

Клонирование процедур. Это обфускирующее преобразование заключается в том, что в программе создаётся несколько копий одной и той же процедуры. Все вызовы исходной процедуры заменяются на вызовы вновь созданных процедур. Обфускирующий потенциал этого преобразования состоит в том, что, во-первых, контекст использования процедуры будет сложнее поддаваться анализу, и, во-вторых, к разным копиям одной и той же процедуры могут быть применены разные обфускирующие преобразования. При обратной инженерии важную информацию о назначении процедуры можно извлечь из контекста использования

этой процедуры. Интерес представляют типичные значения параметров процедуры, выражение, в котором используется процедура, и т. д. Затруднить анализ контекста использования можно, размножив процедуру в нескольких экземплярах. Каждый статический вызов данной процедуры в исходной программе будет заменен вызовом одной из эквивалентных процедур.

Обфускирующее преобразование клонирования процедур можно применять на разных этапах проектирования схем. С точки зрения быстродействия обфускированной схемы преобразование оценивается как бесплатное, но, как и для преобразования открытой вставки, размер обфускированной схемы может значительно возрасти. Для выполнения преобразования достаточно простого семантического анализа программы, и поэтому сложность реализации этого преобразования незначительна. Это обфускирующее преобразование не относится к числу устойчивых, но его применение открывает новые возможности для применения других видов обфускации.

Внесение недостижимого кода. Это обфускирующее преобразование состоит во внесении в программу инструкций, которые на самом деле никогда не выполняются. Инструкция (команда, оператор) программы называется *недостижимой*, если она не выполняется ни при каком наборе входных данных. Недостижимый код может вноситься только совместно с вспомогательными обфускирующими преобразованиями введения непроницаемых (opaque) предикатов [14, 15], которые создают новые пути в графе потока управления. Хотя недостижимые инструкции никогда не выполняются, задача статического определения достижимости кода является алгоритмически неразрешимой, и в этом заключается обфускирующий потенциал этого преобразования. Таким образом, если непроницаемый предикат имеет достаточно сложное устройство, статический анализатор программы окажется не в состоянии определить достижимость кода и будет вынужден предполагать, что инструкции выполняются при некоторых входных данных. За счет введения недостижимого кода можно очень сильно изменить синтаксическое устройство программы, совершенно не изменяя ее поведения.

Обфускирующее преобразование внесения недостижимого кода неприменимо для информационной защиты схем: оно обладает очень низкой устойчивостью. Это объясняется тем, что для проверки правильности работы микроэлектронных схем требуется построить полное тестовое покрытие. Это покрытие предусматривает прохождение всех логических цепочек

схемы. Поэтому все вхождения в программу (схему) недостижимого кода будут легко выявлены на этапе тестирования.

Внесение «мертвого» кода. Инструкция программы называется «мертвой», если при всех значениях входных данных ее выполнение не оказывает влияния на результат работы программы. Такая инструкция может быть безболезненно удалена из текста программы. Обфускирующее преобразование состоит во внесении в существующие базовые блоки процедуры новых «мертвых» инструкций. Обфускирующий потенциал этого преобразования состоит в том, что, как и задача о достижимости инструкции, задача о выявлении «мертвых» инструкций алгоритмически неразрешима. Поэтому при достаточно сложной структуре программы статический анализ может оказаться не в состоянии выявить и устранить «мертвые» инструкции.

В отличие от внесения недостижимого кода, который никогда не выполняется, и, соответственно, не налагает никаких ограничений на составляющие его инструкции, «мертвый» код выполняется хотя бы при некоторых входных данных. Поэтому все свойства вносимого «мертвого» кода должны быть известны при обфускации программы. Например, выполнение инструкций «мертвого» кода никогда не должно приводить к возникновению исключительных ситуаций. При выполнении преобразования внесения «мертвого» кода вносимый код может как генерироваться маскирующим компилятором, так и браться из самой обфускируемой программы. В первом случае компилятор может вставлять инструкции, которые манипулируют со специальными переменными, введенными обфускирующим компилятором и не используемыми в исходной программе. Тогда отсутствие побочного эффекта, влияющего на основную программу, и исключительных ситуаций может быть гарантировано. В этом случае для выполнения преобразования достаточно семантического анализа обфускируемой программы. Если «мертвого» код берется из самой программы, необходимо выполнить анализ потоков данных, чтобы гарантировать отсутствие нежелательных свойств у добавляемого «мертвого» кода.

Обфускирующее преобразование внесения «мертвого» кода (при правильной реализации) может иметь высокую устойчивость и скрытность, и его можно безопасно выполнять на алгоритмическом уровне описания схемы. Сложность его реализации зависит от конкретного способа внесения «мертвого» кода. Если «мертвый» код генерируется из элементов самой программы (схемы), то применение этого преобразования может потребо-

вать достаточно трудоемкого анализа исходной схемы. В этом случае внесение «мертвого» кода целесообразно проводить на самых ранних стадиях обфускации перед применением других обфускирующих преобразований. Вместе с тем, «мертвый» код можно вносить, используя стандартные библиотечные конструкции. Устойчивость обфускации в этом случае будет не столь высока, но зато и сложность реализации будет незначительна.

Внесение дублирующего кода. Назовем *дублирующей* инструкцию программы, вычисляющую значение, которое уже было вычислено ранее. Дублирующая инструкция не является ни недостижимой, ни «мертвой». Дублирующая инструкция может быть заменена на инструкцию пересылки из переменной, в которой хранится уже вычисленное значение. Данное обфускирующее преобразование состоит во внесении в программу дублирующего кода. Маскирующий потенциал этого преобразования состоит в том, что задача статического удаления дублирующего кода (как и две предыдущих задачи) алгоритмически неразрешима, то есть в достаточно сложных случаях статический анализатор программы не сможет идентифицировать дублирующий код. В итоге обфускированная программа будет значительно отличаться от исходной программы. Для выполнения преобразования внесения дублирующего кода необходим межпроцедурный анализ потоков данных.

Обфускирующее преобразование внесения дублирующего кода можно применять на алгоритмическом уровне проектирования схем. С точки зрения быстродействия обфускированной схемы преобразование оценивается как бесплатное, но размер обфускированной схемы может значительно возрасти. Для выполнения преобразования достаточно проведение локального синтаксического анализа программы, и поэтому сложность реализации этого преобразования незначительна. Это обфускирующее преобразование не относится к числу устойчивых, но его применение увеличивает возможности применения других видов обфускации.

Внесение тождеств. Это обфускирующее преобразование заменяет выражения или константы другими, тождественно равными, но более сложными выражениями или константами. Например, арифметическое выражение $x - y$ может быть заменено выражением $\frac{x^2-y^2}{x+y}$, а булева константа 0 выражением $((\neg z \wedge x) \vee (z \wedge \neg x)) \wedge x \wedge z$. Внесение некоторых тождеств может потребовать модификации программы (например, введение новых циклов). Обфускирующий потенциал этого преобразования состоит в том, что выра-

жение или константа, которые могут представлять ценность, оказываются скрытыми. Кроме того, может измениться граф потока управления. Для выполнения такого обфускирующего преобразования компилятор должен иметь библиотеку допустимых тождеств. В ней хранятся ограничения на типы переменных, вид исходного подвыражения и вид замаскированного подвыражения. Однако, из-за специфических свойств машинной арифметики тождественные преобразования имеют ограниченную применимость. Выражения над числами с плавающей точкой, как правило, не могут быть преобразованы (как известно, для машинных арифметических операций с плавающей точкой не выполняется ни свойство дистрибутивности, ни свойство ассоциативности). Целочисленные выражения могут быть преобразованы, только если заведомо известно, что при вычислении никогда не возникает арифметическое переполнение. Для выполнения преобразования внесения тождеств всегда достаточно семантического анализа программы, хотя в простых случаях, когда преобразуется константное значение, достаточно синтаксического анализа.

Устойчивость данного обфускирующего преобразования зависит от используемых тождеств — чем более разнообразна библиотека тождеств, тем более трудоемким будет деобфускация программы или схемы. При проведении обфускации описаний схем на алгоритмическом уровне целесообразно применять арифметические и булевы тождества. Если используется библиотека тождеств, то сложность реализации этого преобразования невелика.

Развертка циклов. Преобразование развертки циклов заключается в том, что несколько или все итерации цикла выписываются в процедуре явно. Частичная или полная развертка циклов — это одно из оптимизирующих преобразований, применяемое, в частности, для автоматического распараллеливания программ. Обфускирующий потенциал преобразования заключается в усложнении структуры цикла, усложнении условия выхода, усложнении приращения управляющей переменной. Если число итераций цикла статически известно, цикл может быть развернут полностью, в противном случае могут быть развернуты две-три итерации цикла. Для выполнения этого преобразования в обфускирующем компиляторе достаточно простого семантического анализа программы.

Преобразование развертки циклов можно применять к алгоритмическим описаниям схем. С точки зрения быстродействия преобразование развертки циклов оценивается как бесплатное, так как количество инструк-

ций, выполняемых на каждой итерации цикла, не изменяется. Однако, размер обфускированной схемы может значительно увеличиться в зависимости от глубины развертки цикла. У этого преобразования невысокая устойчивость и скрытность.

Разложение циклов. Преобразование разложения циклов (loop fission) заключается в разбиении одного цикла со сложной структурой на несколько циклов с более простой структурой. Обфускирующий потенциал преобразования состоит в увеличении сложности графа потока управления процедуры и расширении поля для других обфускирующих преобразований. Для выполнения развертки циклов маскирующий компилятор должен провести анализ потоков данных, чтобы выделить независимые фрагменты тела цикла. Возможен консервативный подход к вызовам процедур и указателям, но за счет пропуска кандидатов на разложение.

Преобразование разложения не ухудшает быстродействия схемы. Однако, его применение может приводить к увеличению размера программы за счет копирования управляющей структуры цикла. Для выполнения развертки циклов компилятор должен провести анализ потоков данных, чтобы выделить независимые фрагменты тела цикла, и поэтому сложность реализации этого преобразования может быть велика.

Переплетение циклов. Переплетение циклов (loop fusion) — это преобразование программы, которая объединяет несколько циклов с одинаковым пространством итерации в один цикл. Переплетение циклов является обратной трансформацией по отношению к преобразованию разложения циклов. Несмотря на то, что граф потока управления процедуры в результате преобразования упрощается, структура объединённого цикла усложняется. Для выполнения преобразования в обфускирующем компиляторе необходим анализ потоков данных, так как тела объединяемых циклов не должны изменять управляющие переменные цикла.

Преобразование разложения циклов не ухудшает быстродействия схемы и ее размера. Оно может быть применено на алгоритмическом уровне описания схемы. Для выполнения преобразования в обфускирующем компиляторе необходим анализ потоков данных, так как тела объединяемых циклов не должны изменять управляющие переменные цикла, и поэтому сложность реализации этого преобразования может быть велика. Устойчивость этого преобразования также невелика.

Аффинные преобразования индексов цикла. Аффинные преобразования индексов цикла состоят в изменении пространства итерирования циклов, что может быть использовано для увеличения сложности программы. Например, если вложенные циклы имеют простую структуру и простое пространство итерирования, то несколько вложенных циклов могут быть преобразованы в один цикл и наоборот. Хотя преобразование вложенных циклов в простой, казалось бы, упрощает структуру программы, но в результате преобразования индексные выражения значительно усложняются, что затрудняет анализ и понимание программы. Для выполнения аффинных преобразований обфускирующий компилятор должен провести анализ потоков данных, так как преобразованию можно подвергать циклы с простой структурой, в которых управляющая переменная не изменяется в теле цикла. В случае консервативного анализа программы циклы, пригодные для преобразований, могут быть пропущены.

Аффинные преобразование индексов циклов не ухудшают быстродействия схемы и ее размера. Хотя индексные выражения в результате преобразования могут усложниться, структура циклов может упроститься. Аффинные преобразования могут незначительно как увеличить, так и уменьшить размер программы. Они может быть применено на алгоритмическом уровне описания схемы. Сложность реализации этих преобразований невелика, и они имеют малую устойчивость и скрытность.

Локализация переменных. Это преобразование ограничивает область «активности» локальных переменных одним базовым блоком. В результате возникает большое количество переменных, что затрудняет анализ процедуры. Переменная v считается *активной* в некоторой точке программы, если при некоторых входных данных присваивание этой переменной в данной точке другого значения изменит результат работы программы. Другими словами, переменная активна, если ее текущее значение может понадобиться в дальнейшем. Описываемое обфускирующее преобразование трансформирует процедуру таким образом, чтобы область жизни переменной ограничивалась одним базовым блоком. Для этого в каждом базовом блоке заводится локальная копия всех используемых переменных. При входе в базовый блок локальным копиям переменных данного базового блока присваиваются значения локальных копий переменных предыдущего базового блока. Поскольку после базового блока может следовать более чем один базовый блок, и, соответственно, управление в базовый блок может попадать из более чем одного базового блока, инструкции согласова-

ния значений переменных выносятся в так называемые связующие базовые блоки.

Преобразование локализации переменных применимо только к описаниям схем на алгоритмическом уровне. С точки зрения скорости работы трансформированной схемы преобразование локализации может быть оценено как дешевое, так как в каждый базовый блок добавляются инструкции копирования переменных. Размер обфускированной схемы существенно увеличивается. Для выполнения преобразования обфускирующий компилятор должен произвести точный анализ потоков данных. В противном случае невозможно гарантировать, что скопированная переменная не будет модифицирована в результате вызова процедуры. Поэтому затраты на реализацию преобразования могут быть значительными. Стойкость преобразования пропорциональна сложности его реализации.

Расширение области действия переменных. Это преобразование является обратным по отношению к предыдущему. Оно максимально расширяет область действия переменных, вынося блочные переменные на уровень локальных переменных процедуры или преобразуя локальные переменные в глобальные. Результатом такого преобразования является утрата свойства локальности для переменных, что, с одной стороны, затрудняет восприятие программы человеком, а с другой стороны, может затруднить автоматический анализ программы. При анализе обфускированных программ с локальными переменными, вынесенными на глобальный уровень, требуется межпроцедурный анализ программы, что существенно сложнее, чем глобальный анализ процедуры. Для выноса локальных переменных на глобальный уровень требуется точный анализ графа вызовов процедур, так как для процедур, которые могут вызываться рекурсивно, или для процедур, вызываемых асинхронно, необходима специальная поддержка. Статическое построение полного и точного графа вызовов программы является алгоритмически неразрешимой задачей в случае динамической типизации или косвенных вызовов процедур. Возможен консервативный подход к этому преобразованию, если сохранять на входе в процедуру значение глобальной переменной в некоторой локальной переменной, и затем восстанавливать старое значение на выходе из процедуры. Однако такой подход требует специальной поддержки нелокальных переходов.

Преобразование расширения области действия переменных применимо на алгоритмическом уровне описания схемы. Оно не увеличивает размера схемы и не ухудшает ее быстродействия. Сложность реализации этого

преобразования может быть значительной в связи с применением процедур статического анализа. Устойчивость преобразования расширения области действия переменных пропорциональна сложности его реализации.

Повторное использование переменных. Это преобразование также является обратным по отношению к преобразованию локализации переменных. Оно объединяет несколько переменных, области жизни которых не пересекаются, в одну переменную. В результате одна и та же переменная в программе будет использоваться для разных целей в разных фрагментах процедуры или всей программы, что осложняет анализ программы человеком.

Чтобы две переменные можно было совместить в одну, необходимо и достаточно, чтобы интервалы активности этих переменных жива не пересекались. Компиляторы выполняют такой анализ при распределении регистров. При этом применяется консервативный подход: если свойство активности для переменной не может быть установлено (например, из-за возможности обращения по указателю), переменная предполагается активной.

Преобразование повторного использования переменных применимо на алгоритмическом уровне описания схемы. Оно не увеличивает размера схемы и не ухудшает ее быстродействия. Сложность реализации этого преобразования может быть значительной в связи с применением процедур статического анализа. Устойчивость преобразования повторного использования переменных пропорциональна сложности его реализации.

Внесение непроницаемых предикатов и переменных. *Непроницаемой переменной* назовем переменную v, вводимую при обфускации программы, которая удовлетворяет следующим условиям.

1. Значение переменной на стадии обфускации программы может быть определено по самой программе и по дополнительной информации, которая известна на стадии обфускации программы, но отсутствует в тексте программы.

2. Задача определения значения непроницаемой переменной по тексту обфускированной программы является сложной.

Непроницаемый предикат — это предикат (выражение, вырабатывающее результат булевого типа), удовлетворяющий вышеуказанным двум признакам. Непроницаемые предикаты могут быть одного из трех типов: PF — предикат, всегда принимающий значение «ложь»; PT — предикат,

принимающий всегда значение «истина»; $P?$ — предикат, который может принимать оба значения. Внесение в программу непроницаемых переменных и предикатов представляется нам одним из важнейших методов обфускации программ. От устойчивости непроницаемых предикатов зависит устойчивость многих методов обфускации программы, например, внесения недостижимого кода. Основная сложность при разработке преобразований потока управления состоит в том, чтобы сделать их не только дешевыми, но и устойчивыми к автоматическими деобфускаторами. Для достижения этого многие преобразования полагаются на существование непроницаемых переменных и предикатов. Изучение многих программ на языках высокого уровня показывает, что большинство таких предикатов чрезвычайно просты. Типичные предикаты имеют вид $p == NULL$, $p == q$, $n <= Value$, где $Value$ — это некоторое числовое значение. Необходимо уметь генерировать непроницаемые предикаты, которые бы напоминали такие шаблоны. Некоторые методы получения непроницаемых предикатов перечислены ниже:

- Использование различных тождеств, аналогично преобразованию внесения тождеств;
- Построение сложных булевых выражений с помощью эквивалентных преобразований из формулы true (или false). В простейшем случае мы можем взять k произвольных булевых переменных $x_1, \ldots, x_k$ и построить из них формулу $(x_1 \vee \neg x_1) \wedge \cdots \wedge (x_k \vee \neg x_k)$. Далее с помощью эквивалентных алгебраических преобразований часть скобок (или все) раскрываются, и в результате получается искомый непроницаемый предикат.

Диспетчеризация потока управления. Данное обфускирующее преобразование приводит граф потока управления к однородному («плоскому») виду. Оно преобразование представляется нам одним из наиболее важных, поэтому подробно рассмотрено в соответствующем разделе. В работах [7, 15, 18] предлагается подход, основанный на глубоком преобразовании графа потока управления. Граф потока управления перестраивается таким образом, что управление от от одного базового блока попадает на другой базовый блок не непосредственно, а через специальный базовый блок «диспетчера». Обфускирующий потенциал этого преобразования состоит в том, что внешний вид программы существенно изменяется: циклы, условные операторы исходной программы приобретут

совершенно другой вид, и это значительно затруднит восприятие программы человеком. Форма графа потока управления процедуры, наличие в нем шаблонов типичных конструкций высокого уровня, таких как условный оператор, операторы цикла уже дает определенную информацию для анализа программы. Например, по повторению некоторого фрагмента несколько раз подряд можно выдвинуть гипотезу о том, что в этом месте программы была выполнена развертка цикла. Кроме того в результате преобразования графа потока управления после блока диспетчера может следовать любой базовый блок процедуры, после которого снова следует базовый блок диспетчера. Граф потока управления принимает «плоский» вид. Из-за потенциального следования любого базового блока после любого базового блока графа потока управления анализ потоков данных окажется неточным. Данное преобразование, по сути, переводит задачу анализа графа потока управления в задачу анализа потоков данных, которая в общем случае алгоритмически неразрешима. С помощью преобразования построения диспетчера можно добиться того, что задача статического восстановления исходного порядка следования базовых блоков обфускированной процедуры будет вычислительно сложной.

Использование булевых формул В работе работе [18] используется другой подход к «внедрению» вычислительно-сложной задачи в диспетчер графа потока управления. В работе отмечается, что в простейшем случае диспетчер может рассматриваться как детерминированный конечный автомат. Для увеличения сложности его анализа предлагается значительно увеличить пространство состояний диспетчера. Новый автомат получается прямым произведением исходного автомата переходов обфускируемой процедуры и другого достаточно большого конечного автомата, который должен быть сложным экземпляром задачи принятия с линейной памятью. Предполагается, что такой автомат получается запросом у соответствующего оракула. Задача достижимости некоторого состояния для такого диспетчера оказывается PSPACE-полной, так как к ней сводится задача принятия с линейной памятью. Задача устранения избыточных инструкций из обфускированной программы оказывается PSPACE-трудной.

6. Заключение

Подводя итоги проведенного анализа обфускирующих преобразований программ, мы можно сформулировать ряд общих принципов оптимальной стратегии применения обфускирующих преобразований для информационной защиты проектных решений.

Эти принципы определяют состав библиотеки обфускирующих преобразований и порядок их применения на алгоритмическом уровне проектирования схем.

1. Нецелесообразно применять преобразования, которые обладают незначительной устойчивостью. Таким образом, нет необходимости использовать лексические преобразования.

2. Нецелесообразно применять преобразования, обфускирующий эффект которых разрушается на дальнейших этапах проектирования схемы. Таким образом, при обфускации схем можно отказаться от применения преобразования внесения недостижимого кода

3. Некоторые из преобразований способны разрушить эффект, достигнутый за счет применения других преобразований. Поэтому в процессе проведения обфускации схемы необходимо вести протокол, в котором учитывается результат применения каждого преобразования, так чтобы взаимно обратные преобразования не применялись к одному и тому же фрагменту описания.

4. Обфускирующие преобразования, требующие проведения сложного глобального анализа описания схемы должны применяться в первую очередь.

5. Обфускирующие преобразования, усложняющие проектирование схемы (небезопасные преобразования), не должны применяться на алгоритмическом уровне проектирования схемы.

6. Обфускирующие преобразования, усложняющие структуру описания схемы и тем самым затрудняющие анализ описания, на каждом этапе должны применяться в последнюю очередь.

Таким образом оптимальная стратегия применения обфускирующих преобразований на алгоритмическом уровне проектирования схем такова.

1. Применяются преобразования открытой вставки небиблиотечных процедур, переплетения процедур и клонирования процедур.

2. Применяются преобразования развертки циклов и разложения циклов.

3. Применяются преобразования дублирования кода и внесения «мертвого» кода с использованием непрозрачных предикатов.

4. Применяются преобразования локализации переменных, расширения области действия переменных и повторного использования переменных.

5. Применяется преобразование диспетчиризации программы.

Литература

[1] Варновский Н.П., Гырдымов П.А., Девянин П.Н., и др., Введение в криптографию. Под общ. Ред. В.В.Ященко, М.: МЦНМО: «ЧеРо». 2000.

[2] Домашев А.В., Грунтовик Л.М., Попов В.О., и др. Программирование алгоритмов защиты информации. — М.: «Нолидж», 2002.

[3] Грушо А. А. , Применко Э. А. , Тимонина Е. Е., Анализ и синтез криптоалгоритмов, Курс лекций, Москва, 2000.

[4] Казарин О. В., Безопасность программного обеспечения компьютерных систем, М.: МГУЛ, 2003, 212 с.

[5] Романец Ю.В., Тимофеев П.А., Шаныгин В.Ф., Защита информации в компьютерных системах и сетях. М.: Радио и связь, 1997.

[6] Barak B., Goldreich O., Impagliazzo R., Rudich S., Sahai A., Vedhan S., Yang K., On the (Im)possibility of obfuscating programs. *CRYPTO'01 — Advances in Cryptology*, Lecture Notes in Computer Science, v. 2139, 2001, p. 1–18.

[7] Чернов А.В., Анализ запутывающих преобразований программ, В сб. «Труды Института системного программирования», том 3, 2002, с. 137-163.

[8] Чернов А.В. Интегрированная исследовательская среда Пуаро для изучения обфускирующих преобразований программ, Препринт ИСП РАН, 2003.

[9] Appel A. W. Modern Compiler Implementation in C. Cambridge University Press. 1998.

[10] Aushmith D. Tamper resistant software: an implementation, Lecture Notes in Computer Science, v. 1174, 1996, p.317–333.

[11] Ball T., Larus J. R. Optimally Profiling and Tracing Programs. In *Proceedings of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'92)*, 1992.

[12] Calder B., Feller P., Eustace A. Value Profiling. In *Proceedings of the 30th International Symposium on Microarchitecture (MICRO-30)*, 1997.

[13] Chernov A.V., A new program obfuscation method. In Proceedings of the International Workshop on Program Understanding, 2003, Novosibirsk, p.70–80.

[14] Collberg C., Thomborson C., Low D., A taxonomy of obfuscating transformations, Tech. Report, N 148, Dept. of Computer Science, Univ. of Auckland, 1997.

[15] Collberg C., Thomborson C., Low D., Manufacturing cheap, resilient and stealthy opaque constructs. In Proc. of the Symposium on Principles of Programming Languages, 1998, p.184-196.

[16] Collberg C., Clark D., Thomborson C., Software Watermarking: Models and Dynamic Embeddings. In Proc. of the Symposium on Principles of Programming Languages, 1999, p. 311–324.

[17] Collberg C, Thomborson C., Watermarking, Tamper-Proofing, and Obfuscation — Tools for Software Protection, IEEE Transactions on Software Engineering, 28, No. 6, June 2002.

[18] Chow S., Gu Y., Johnson H., Zakharov V., An approach to the obfuscation of control flow of sequential computer programs. Information Security Conference, Lecture Notes in Computer Science, v. 2200, 2001, p. 144–156.

[19] Diffie W., Hellman M.E. New directions in cryptography. *IEEE Transactions in Information Theory*, v. 22, 1976, p.644–654.

[20] Goldwasser S., Kalai Y.T. On the Impossibility of Obfuscation with Auxiliary Input. *FOCS*, 2005, p.553–562

[21] Ivanov K. S., Zakharov V. A., Program obfuscation as obstruction of program static analysis, *Tech. Reports of the Institute for system programming*, v. 6, 2004.

[22] Harrison W. H. Compiler Analysis of the Value Ranges for Variables. *IEEE Transactions on Software Engineering*, v. 3, N 3, 1977, p. 243-250.

[23] Hohl F. Time limited blackbox security: Protecting mobile agents from malicious hosts, in *Mobile Agent Security*, Lecture Notes in Computer Science, v. 1420, 1998, p. 91–113. 1998.

[24] Linn C., Debray S. Obfuscation of executable code to improve resistance to static disassembly. In *Proc. of the 10th. ACM Conference on Computer and Communications Security (CCS 2003)*, Oct. 2003.

[25] Lynn B., Prabhakaran M., Sahai A. Positive results and techniques for obfuscation, Lecture Notes in Computer Science, v. 3027, 2004, p. 20–39.

[26] von Mayrhauser A. , Vans A. M. Program Understanding: Models and Experiments. In *Advances in Computers*, v. 40, 1995. San Diego: Academic Press. pp. 1–38

[27] Muchnick S. Advanced Compiler Design and Implementation. Morgan Kaufmann Publishers, 1997.

[28] Ogisso T., Sakabe Y., Sochi M., Miyaji A. Software obfuscation on a theoretical basis and its implementation, *IEEE Transactions on Fundamentals*, E86-A(1), 2003.

[29] Varnovsky N.P., Zakharov V.A. On the possibility of provably secure obfuscating programs, In Proc. of the 5th Int. Conference Perspectives of System Informatics (PSI'03), 2003, p.71–78.

[30] Varnovsky N.P. A note on the concept of obfuscation. *Tech. Reports of the Institute for system programming*, v. 6, 2004.

[31] Wee H. On obfuscating point functions, *STOC*, ACM, 2005, p. 523–532.

[32] Wang C., Hill J., Knight J. Davidson J., Software tamper resistance: obstructing static analysis of programs, Tech. Rep., N 12, Dep. Of Comp. Sci., Univ. of Virginia, 2000.

[33] Wroblewski G., General method of program code obfuscation, In Proc of the Int. Conference on Software Engineering Research and Practice (SERP), 2002, p.153–159.

О проблеме логико-термальной эквивалентности последовательных программ с динамической памятью

В. А. Захаров, К. С. Иванов

Аннотация. Некоторые методы обфускации программ предусматривают использование аппарата указателей. Это обусловлено тем, что задача анализа программ, в которых используются указатели, относится к числу алгоритмически трудных задач. Для разработки методов анализа программ, использующих динамическую память, целесообразно ввести и исследовать строгую математическую модель, которая могла бы позволить проводить доказательство корректности полученных алгоритмов, оценивать их сложность и описывать с ее помощью обфускирующие и деобфускирующие преобразования.

В предложенной работе предпринята попытка создать такую модель на основе известной модели стандартных схем программ. В статье рассматривается расширенный вариант стандартных схем программ, обогащенных средствами работы с указателями и динамической памятью. Для предложенной модели рассмотрена задача проверки логико-термальной эквиавлентности, которая ранее была исследована для обычных стандартных схем программ. Показано, что проблема логико-термальной эквивалентности для схем программ с динамической памятью разрешима за полиномиальное время.

1. Введение

Развитие скрипт-языков (Perl, PHP, JavaScript, VBScript), внедрение программных окружений (Java, Microsoft.NET) поставили перед разработчиками новую задачу: защищать от несанкционированного использования и плагиата свои программы. Особенность этих задач для подобных платформ заключается в том, что распространяемый код зачастую является неоткомпилированным, или же откомпилированным в форму, из которой легко извлекается исходный код на высокоуровневом языке. Таким образом становится трудно защитить не столько саму программу, сколько лицензированные алгоритмы, используемые в ней. Поэтому необходимо самим защищать, или подписывать свой код . Иногда это позволяет крупным компаниям выделять направления утечки данных и прекращать сотрудничество с дилерами, пиратирующими их продукцию.

Одной из развивающихся областей в защите информации является обфускация — эквивалентные преобразования кода программ, как исходного, так и откомпилированного, в форму менее понятную для человека и

автоматического анализатора. В идеале после применения методов обфускации программа должна быть абсолютно непонятной, т.е. из нее нельзя извлечь полезной информации (обычно рассматриваются определенные классы свойств). При этом функциональное поведение программы должно сохраняться. Одно из самых крупных исследований таких преобразований провел Wroblewski [5]. В работе [5] выделены все атомарные преобразования над программами, показана эффективность обфускации, проведен ряд статистических исследований.

Доказано [12], что универсального обфускатора не существует. Однако, в частных случаях можно строить эффективные преобразования. Одними из первых разработчиков методов обфускации являются Collberg и Thomborson [3, 4]. Стоит заметить, что в их работах используются структуры, хранящиеся в динамической памяти. На основе подобных структур, сложность анализа которых высока, строятся предикаты типа: указывает ли одна переменная туда же, куда и другая. Использование аппарата указателей является одним из мощнейших инструментов запутывания кода программ.

Однако, существует ряд алгоритмов анализа (Steensgaard [9], Shapiro and Horwitz [10], Andersen [8], Hind [11]), направленных на исследование диапазонов значений переменных-указателей. Первые три алгоритма являются нечувствительными к потоку управления и контексту. Множества, получаемые в результате их работы, действительны для всей программы в целом. Такие алгоритмы не учитывают порядок операций в программе. За счет этого существенно повышается скорость анализа программы, но снижается точность анализа. Тем не менее, такой анализ позволяет получить представление о структурах в динамической памяти программы. Четвертый алгоритм (Hind) является чувствительным как к потоку, так и к контексту. Его результат берется в определенной точки программы для определенной переменной. Анализ учитывает практически все особенности программы, длится существенно дольше, чем у предыдущих алгоритмов, но и результат получается высокой степени точности.

Такие алгоритмы могут выделять множества, на которые ссылаются указатели. Благодаря этому возможно избавиться от не влияющих на код указателей и конструкций в динамической памяти. Хотя и доказано, что полный точный анализ указателей является NP-трудным [6], в программах часто встречаются простые и разрешимые случаи. А иногда не обязательно исследовать все указатели, достаточно лишь выделить некоторые необходимые множества.

Основным подходом в обфускации является сложностной подход, когда выяснить какое-либо свойство программы можно, решив сложную (NP-

трудную) задачу. Но специалист может выделить частный случай задачи, проанализировать ее, воспользоваться опытом и либо решить ее реализацию, либо обойти поставленную задачу и в итоге получить полезную информацию о свойстве программы. Поэтому был рассмотрен вопрос о целесообразности постановки именно сложных задач перед алгоритмами анализа. Оказалось, что можно поступить иначе. Был введен новый подход к обфускации [7]. Его основная идея состоит в том, чтобы сводить к минимуму полезную информацию, которая может быть извлечена из программы применением алгоритмов анализа. Оценка осуществляется относительно существующих алгоритмов анализа. В случае анализа диапазона указателей минимизировать полезность можно, если все указатели по результатам анализа будут ссылаться на одно, общее множество адресов. В работе [7] так же был представлен алгоритм для минимизации полезности применения алгоритмов из [9],[10],[8].

Во всех перечисленных работах каждый раз используется свое представление программ, что затрудняет доказательство применимости алгоритмов обфускации. Более того, чтобы ввести эквивалентные преобразования программ в рамках обфускации, необходимо доказать их эквивалентность, что требует наличия сформулированного понятия эквивалентности, а также распознающих ее алгоритмов.

При разработке методов обфускации (маскировки) программ требуется обоснование стойкости этих методов относительно анализа программ. Однако, ни в одной работе не приводится строгого обоснования стойкости этих методов. В значительной части это обусловлено отсутствием такой же корректной постановки работы алгоритмов анализа, а так же критериев стойкости методов обфускации.

Проблемы с корректностью описания связаны с отсутствием формальных моделей программ, для которых было бы удобно провести корректное обоснование. Поэтому первый этап в исследовании и обосновании методов и критериев обфускации — создание адекватной модели. В свою очередь алгоритмы распознавания эквивалентности необходимо строить на основе некоторой математической модели программ. Наличие в преобразованиях динамической памяти требует описание ее в модели. Были проведены поиски подходящей модели, но положительного результата они не дали. На сегодняшний день языковые модели ограничиваются лишь спецификациями, меняющимися от реализации к реализации. Что касается формальных моделей, абстрагированных от реализации, большинство из них строятся на базе Abstract State Machines. Одна из таких моделей описана Замулиным в работе [2]. Это направление исследований представляется перспективным для построения новых алгоритмов верификации моделей программ.

Тем не менее авторами настоящей статьи была предпринята попытка создать собственную модель с динамической памятью, которая:

- приближала бы в некотором роде современные языки программирования,
- имела строгое математическое описание,
- просто описывала операции над динамической памятью.

Такая модель описывается в разделе 2 в качестве расширения стандартных схем программ, полностью математически описанных в [1].

Функциональная эквивалентность стандартных схем программ неразрешима [1], поэтому следует искать такие отношения эквивалентности, которые служили бы более грубым приближением функциональной экивавлентности. Примером такой эквивалентности является логико-термальная эквивалентность программ. Для программ без ветвлений и циклов она полностью совпадает с функциональной. А для остальных программ из логико-термальной эквивалентности следует функциональная. Обратное, вообще говоря, не верно. Поэтому в этой работе и исследуется разрешимость логико-термальной эквивалентности для программ с динамической памятью. На этой базе уже будет возможно создавать эквивалентные преобразования.

Логико-термальная эквивалентность важна еще по следующей причине. Для обфускации и в задачах анализа программ необходимо оценивать схожесть поведения программ. Например, при анализе метаморфных вирусов. Каждый такой вирус имеет автоматный шаблон своего кода. При попадании на конкретную машину он автоматически подставляет вместо шаблонных переменных некоторые машинно-зависимые конструкции. Необходимо уметь распознавать такие шаблоны и сличать их с уже сделанной подстановкой.

В разделах 4 и 5 исследуется проблема разрешимости логико-термальной эквивалентности при наличии динамической памяти.

2. Модель программ с динамической памятью

Определение 1. Множеством термов *сигнатуры* Σ *(*$Term_\Sigma$*) называется наименьшее множество, содержащее все слова алфавита* Σ.

Для определения структуры и функционирования программ, рассматриваемых в дальнейшим, будем пользоваться схемами программ.

Определение 2. *Под* схемой модели программ S *в базисе операторов* $\mathcal{Z}$ *сигнатуры* Σ *будем понимать конечный размеченный ориентированный граф, вершины которого имеют 5 типов:*

1. Начальная вершина. *В схеме имеется ровно одна начальная вершина, помеченная начальным оператором из $\mathcal{Z}$. Она имеет одну исходящую дугу и ни одной входящей.*

2. Заключительная вершина. *Схема может иметь несколько заключительных вершин, помеченных заключительным оператором. Из вершин этого типа не исходит ни одной дуги.*

3. Вершина-преобразователь. *Вершина этого типа помечена преобразующим оператором и имеет одну исходящую дугу.*

4. Вершина-распознаватель. *Данная вершина помечена некоторым атомом сигнатуры Σ и имеет две исходящие дуги, помеченные как И и Л(в соответствие истинности и ложности рассматриваемого атома).*

5. Пустой цикл. *Специальный вид распознавателя, с обеими исходящими дугами, направленными на вход этому же оператору.*

Определим модель с динамической памятью (далее МДП). Сигнатура Σ состоит из следующих непересекающихся множеств:

- $Var = \{x_1, \ldots, x_n\}$ — множество предметных переменных.
- $Func = \left\{f_1^{(k_1)}, \ldots, f_m^{(k_m)}, \&^{(1)}, *^{(1)}\right\}$ — множество функциональных символов. Верхние индексы k_i задают арность(местность) символа.
- $Pred = \left\{p_1^{(l_1)}, \ldots, p_r^{(l_r)}\right\}$ — множество предикатных символов (отношений). Верхний индекс l_i задаёт местность отношения.
- $Const = \{c_1, \ldots, c_l, \bot\}$ — множество констант.
- $\mathcal{A} \bigcup nil$ — множество всех адресных имен ($\mathcal{A}$ — счетное).

Замечание 1. Дополнительно будем рассматривать следующие множества: $\mathcal{X} = Var$, $\mathcal{AF} = \{*, \&\}$, $\mathcal{F} = (Func \setminus \mathcal{AF}) \cup Const$, $\mathcal{AF}(\mathcal{X}) = \underbrace{f \ldots f}_{i}(x)$, $f \in \mathcal{AF}$, $x \in \mathcal{X}$, $i \geqslant 0$.

Теперь можно построить терм над сигнатурой Σ (по определению). Далее будут рассматриваться термы только над этим множеством.

Определим базис операторов $\mathcal{Z}$ для модели программы:

- *Начальный оператор* — $start(x_1, \ldots, x_k)$, где $k \geq 0$, а $x_1, \ldots, x_k$ — попарно различные переменные, называемые результатом этого оператора.

- *Заключительный оператор* — $stop(\tau_1, \ldots, \tau_n)$, где $n \geq 0$, а $\tau_1, \ldots, \tau_n$ — термы(аргументы *stop*).
- *Оператор присваивания* — $x := \tau$, где $x, \tau \in Term_\Sigma$.
- *Атом* — $p_i^{(l_i)}(\tau_1, \ldots, \tau_{k_i})$, где $p_i \in Pred, \tau_1, \ldots, \tau_{k_i} \in Term_\Sigma$.
- *Оператор выделения памяти* — $new(x_i)$, где $x_i \in Term_\Sigma$

Определение 3. Интерпретацией *сигнатуры* Σ *называется функция* I, *которая сопоставляет:*

- *любой константе* $I(c) = d,\ d \in D_I$, *где* D_I — *конечная статическая область интерпретации;*
- *любому адресу* $I(a) = d,\ d \in \mathcal{M}_I$, *где* $\mathcal{M}_I$ — *конечная динамическая область интерпретации;*
- *функциональным символам вида* $f_i^{(k_i)} \in Func$ — *отображение:*

$$I\left(f_i^{(k_i)}\right) = F_i^{(k_i)} : D_I^{k_i} \to D_I$$

- *предикатным символам* $p_j^{(l_j)}$ *всюду определенный предикат*

$$I\left(p_j^{(l_j)}\right) = P_j^{(l_j)} : D^{(r)} \cup \mathcal{M}_I^{(s)} \to \{И, Л\}$$
$$r \geqslant 0, s \geqslant 0, l_j \geqslant 0, r + s = l_j$$

- *функциональным символам* $*$ *и* $\&$ — *функциональные операторы* $*_\psi$ *и* $\&_\psi$, *зависящие от состояния памяти.*

Определение 4. *Пара* (S, I), *где* S — *схема в базисе операторов* $\mathcal{Z}$ *сигнатуры* Σ, *а* I — *интерпретация сигнатуры* Σ, *называется* интерпретированной схемой *или* программой.

Определение 5. Состояние памяти ψ — *это пара* $\psi = \langle M, (\sigma_1, \sigma_2)\rangle$, *где:*

1. Адресное пространство M — *некоторое конечное подмножество* $\mathcal{M}, (M \subset \mathcal{M}_I)$;
2. Оценка — *пара* (σ_1, σ_2), *где* $\sigma_1 : Var \to\ M \cup \perp$, $\sigma_2 : M \to\ M \cup D_I$.

Заметим, что оценка σ_1 — биекция. Множество всех состояний — Ψ. Следующие выражения для $x \in Var, a \in M$ можно интерпретировать как: $\sigma_2(a)$ — значение по адресу a, $\sigma_2(\sigma_1(x))$ — значение x.

Тогда каждый оператор интерпретируется как отображение одного состояния памяти в другое:

$$Op : \Psi \to \Psi, \Psi = \{\psi \mid \psi = \langle M, (\sigma_1, \sigma_2)\rangle\}$$

Далее определим значение Ω_ψ на состоянии памяти $\psi = \langle M, (\sigma_1, \sigma_2)\rangle$:

$$\Omega_\psi(\tau) = \begin{cases} I(\tau),\ \tau \in Const \vee \tau \in \mathcal{A}; \\ \sigma_2(\sigma_1(\tau)),\ \tau \in Var; \\ *\left(\Omega(\tau_1)\right),\ \tau = *\left(\tau_1\right); \\ \&\left(\Omega(\tau_1)\right),\ \tau = \&\left(\tau_1\right); \\ I(f)(\Omega(\tau_1), \ldots, \Omega(\tau_n)),\ \tau = f(\tau_1, \ldots, \tau_n), f \in Prev \cup Func; \end{cases}$$

Теперь можно определить интерпретацию функциональных символов $*$ и $\&$ на состояние памяти $\psi = \langle M, (\sigma_1, \sigma_2)\rangle$:

$$I(*) = *_\psi$$

$$*_\psi(\tau) = \begin{cases} \sigma_2(\Omega_\psi(\tau)),\ \text{если } \Omega_\psi(\tau) \in M \\ \bot,\ \text{иначе} \end{cases}$$

$$I(\&) = \&_\psi$$

$$\&_\psi(\tau) = \begin{cases} \bot,\ \tau \in Const \vee \tau \in \mathcal{A}; \\ \sigma_1(\tau),\ \tau \in Var; \\ \Omega_\psi(\tau_1),\ \tau = *\tau_1; \\ \bot,\ \text{иначе} \end{cases}$$

Определим функционирование программы. До начала программы — состояние памяти ψ_0 определяется следующим образом:

$$M^0 = \{\varnothing\} \qquad \sigma_1^0 : x \to \bot, \forall x \qquad \sigma_2^0 : \varnothing$$

При переходе через начальную вершину с начальным оператором $start(x_1, \ldots, x_k)$ осуществляется следующие преобразование:

$$\psi_0 = \left\langle M^0, (\sigma_1^0, \sigma_2^0)\right\rangle \longrightarrow \psi_1 = \left\langle M^1, (\sigma_1^1, \sigma_2^1)\right\rangle$$

$$M^1 = \{a_1, \ldots, a_k, nil\}$$
$$\sigma_1^1 : x_i \to a_i,\ 1 \leqslant i \leqslant k$$
$$\sigma_1^1 : x \to \bot,\ x \notin \{x_1, \ldots, x_k\}$$
$$\sigma_2^1 : a_i \to d_i,\ 1 \leqslant i \leqslant k,\ d_i \in D_I \cup M^1$$

Пусть далее мы приходим в вершину v схемы S с состоянием $\psi = \langle M, (\sigma_1, \sigma_2)\rangle$, тогда возможны следующие случаи.

Если v — вершина-преобразователь с оператором выделения памяти $new(\tau)$ для $\&_\psi(\tau) \in M$, тогда мы получим состояние памяти $\psi' = \langle M', (\sigma_1, \sigma_2')\rangle$ по следующему правилу:

$$M' = M \cup \{a_l\}, a_l \in \mathcal{M}_I \setminus M$$
$$\sigma_2'(a) = \begin{cases} nil, & a = a_l; \\ a_l, & a = \sigma_2(\&_\psi(\tau)); \\ \sigma_2(a), & \text{иначе.} \end{cases}$$

Если же $\&_\psi(\tau) \notin M$, то $\psi' = \psi$. Далее осуществляется переход по единственной исходящей из v дуге с состоянием памяти ψ'.

Если v — вершина-преобразователь с оператором присваивания вида $\tau_1 := \tau_2$, тогда:

1. новое состояние определяется как $\psi' = \langle M, (\sigma_1, \sigma_2')\rangle$, где

$$\sigma_2'(a) = \begin{cases} \sigma_2(a), & a \neq \&_\psi(\tau_1) \\ \Omega_\psi(\tau_2), & a = \&_\psi(\tau_1) \end{cases}$$

Осуществляется переход по единственной исходящей из v дуге с состоянием памяти ψ'.

Если v — вершина-распознаватель с атомом $p_j(\tau_1, \ldots, \tau_{l_j})$, то происходит переход по дуге, соответствующей значению этого атома на аргументах $\tau_1, \ldots, \tau_{l_j} \in D_I \cup M$. Состояние памяти при переходе сохраняется.

Если v — заключительная вершина с оператором $output(x_1, \ldots, x_m)$, тогда $\psi' = \langle A', (\sigma_1', \sigma_2')\rangle$ определяется следующим образом:

$$\widetilde{x} \rightleftharpoons \sigma_2(\sigma_1(x)) \text{ — значение переменной } x$$
$$\sigma_1' = \sigma_1|_{\{x_1,\ldots,x_m\}}$$
$$\sigma_2' = \sigma_2|_{M'}$$
$$M' = \left(\bigcup_{\widetilde{x} \in D_I} \{\sigma_1'(x)\} \cup \bigcup_{\widetilde{x} \in M} \bigcup_{i=1}^{\infty} \{*^i(x)\} \right) \setminus D_I$$

После достижения заключительной вершины программа считается завершенной в состоянии ψ'.

Определение 6. *Фрагментом МДП назовем некоторое обобщение схемы МДП. В отличие от графа схемы, граф фрагмента может содержать*

свободные дуги (входы и выходы), но они должны быть уникально пронумерованы. Кроме того свободным дугам приписаны конечные множества термов (результаты входа и аргументы выхода соответственно).

Подробно свойства фрагментов описаны в [1, гл. 6].

3. Свойства модели программ с динамической памятью

Введем несколько определений, которые понадобятся в данном разделе.

Определение 7. *Стандартные схемы S_1 и S_2 в базисе операторов $\mathcal{Z}$ сигнатуры Σ функционально эквивалентны ($S_1 \sim S_2$), если для любой интерпретации I сигнатуры Σ программы (S_1, I) и (S_2, I) либо обе зацикливаются либо останавливаются с одинаковым результатом, т.е. $\Omega(S_1, I) \simeq \Omega(S_2, I)$.*

Можно показать, что введённая в разделе 2 МДП является расширением модели, описанной в [1]. Для этого рассмотрим статические программы (без использования динамической памяти и адресов). Тогда в МДП получим не использование оператора *new* и функциональных символов &, $*$.

Теорема 1 (о консервативном расширении)**.** *МДП является консервативным расширением стандартной модели схем программ, т.е. в классе статических программ каждой вычислительной истории z_1 схемы программы S_1 из стандартной модели схем программ (далее СМСП) можно взаимнооднозначно (с точностью до перенумерации адресов переменных) сопоставить вычислительную историю z_2 схемы S_2 модели МДП.*

Следствие 1. *Функциональная эквивалентность статических программ из СМСП верна и для МДП.*

Следствие 2. *Проблема функциональной эквивалентность схем МДП неразрешима.*

Введем понятия значения терма на пути, а также логико-термальной истории. Обозначим за $vl(w, t)$ значение терма t на пути w.

$$\begin{aligned} &\forall w: \\ &vl(w, \&x) = a_x, \\ &vl\left(w, f(t_1, \ldots, t_n)\right) = \begin{cases} f\left(vl(w, t_1), \ldots, vl(w, t_n)\right), \text{ если } vl(w, t_1) \notin \mathcal{A} \cup \{\perp\} \\ \perp, \text{ иначе} \end{cases} \end{aligned}$$

$$vl(w, *t_1) = \begin{cases} vl(w, x), \text{ если } vl(w, t_1) = a_x \\ vl(w, *(a)), \text{ если } vl(w, t_1) = a \\ \bot, \text{ иначе} \end{cases}$$

$$\begin{aligned} addr(\varepsilon) &= a_{x_1}, \ldots, a_{x_n}, a_{x_i} \in \mathcal{A} \wedge x_i \in Var \\ addr(w.\langle x := t\rangle) &= addr(w) \\ addr(w.\langle new(x)\rangle) &= addr(w) \cup \{a\}, a : \begin{bmatrix} a \in \mathcal{A} \\ a \notin addr(w) \end{bmatrix} \\ vl(\varepsilon, x) &= x \\ vl(\varepsilon, *x) &= \bot \end{aligned}$$

$$\begin{aligned} vl(w.\langle x := t\rangle, x) &= vl(w, t) \\ vl(w.\langle x := t\rangle, y) &= vl(w, y) \\ vl(w.\langle *t_1 = t_2\rangle, x) &= \begin{cases} vl(w, t_2), \text{ если } vl(w, t_1) = a_x \\ vl(w, x), \text{ иначе} \end{cases} \end{aligned}$$

$$\begin{aligned} vl(w.\langle new(x)\rangle, x) &= a : \begin{bmatrix} a \in \mathcal{A} \\ a \notin addr(w) \end{bmatrix} \\ vl(w.\langle new(y)\rangle, x) &= vl(w, x). \end{aligned}$$

Для пути w в некоторой схеме S определим *логико-термальную историю* $lt(S, w)$ как слово, которое строится следующим образом:

1. Если путь w не содержит распознавателей и заключительной вершины, то $lt(S, w)$ — пустое слово.
2. Если $w = w've$, где v — распознаватель с атомом $p_j(\tau_1, \ldots, \tau_{k_j})$, а e — выходящая из него δ-дуга, $\delta \in \{И, Л\}$, то
$$lt(S, w) = lt(S, w')p_j^{\delta}\left(vl(w', \tau_1), \ldots, vl(w', \tau_{k_j})\right).$$
3. Если $w = w'v$, где v — заключительная вершина с оператором $stop(x_1, \ldots, x_k)$, то
$$lt(S, w) = lt(S, w')\left[vl(w', x_1), \ldots, vl(w', x_k)\right].$$

Определение 8. Детерминантом *(det(S))* — *стандартной схемы* S *назовём множество логико-термальных историй всех ее путей, завершающихся заключительным оператором.*

Определение 9. *Схемы* S_1 *и* S_2 *называются* логико-термально эквивалентными $\left(S_1 \overset{lt}{\sim} S_2\right)$, *если* $det(S_1) = det(S_2)$.

4. Сети инвариантов

С каждым путём w, оканчивающимся дугой e фрагмента G можно связать множество равенства термов:

$$gen(G, w) = \{x = \tau \mid vl(w, x) = vl(w, \tau),\ x \neq \perp\}$$

Будем говорить, что множество $gen(G, w)$ порождается путём w.

Определение 10. Инвариантом *дуги* e *из* G *назовём множество равенств:*

$$invar(G, e) = \bigcap_{w \in W_e} gen(G, w)$$

Где W_e — *множество всех путей, начинающихся входами фрагмента* G *и оканчивающихся дугой* e.

Определение 11. Функциональная сеть — *это четверка* $s = \langle C, V, \Psi, \Gamma \rangle$, *где:*

C — *конечное множество сигнатурных элементов.*

V — *конечное множество вершин.* $V = \{v\}$. *Вершина* $v = \{c\}$ — *конечное множество элементов* $c \in C$

Ψ — *функция означивания.* $\Psi : C \to \mathcal{X} \cup \mathcal{F} \cup \mathcal{AF} \cup \mathcal{A}$

Γ — *отображение зависимости.* $\Gamma : C \times \mathbb{N} \to V$. $\Gamma(c, i)$ — *обозначает зависимость элемента* $c : \Psi(c) \in \mathcal{F} \cup \mathcal{AF}$ *по аргументу* i. *Количество дуг из элемента определяется арностью функционального символа и обозначается* $d(\Psi(c))$.

Определение 12. Полуправильной сетью *назовём сеть, удовлетворяющую следующим условиям:*

1. *Каждая вершина содержит не более одного элемента* c, *которому приписан функциональный символ* $(\Psi(c) \in \mathcal{F})$;
2. *В сети нет элементов-дублей, т. е. не существует таких различных элементов* c_1 *и* c_2, *для которых:*

$$\Psi(c_1) = \Psi(c_2) \wedge \forall i \left[(1 \leqslant i \leqslant d(\Psi(c_1))) \wedge \Gamma(c_1, i) = \Gamma(c_2, i)\right]$$

Определим далее множества $know_s(c)$ и $know_s(v)$:

$$know_s(c) = \begin{cases} \{g\}, \text{если } \Psi(c) = g \wedge d(g) = 0 \\ \{f(\tau_1, \ldots, \tau_n) \mid \forall i \ (1 \leqslant i \leqslant n) \ \tau_i \in know_s(\Gamma(c, i))\}, \\ \qquad\qquad \text{если } \Psi(c) = f \wedge d(f) = n > 0 \end{cases}$$

$$know_s(v) = \bigcup_{c \in v} know_s(c)$$

Отметим, что множества в общем случае — счетно-бесконечные, так как в сети возможны циклы по элементам из $\mathcal{AF}$.

Определение 13. Множеством равенств *сети s назовём:*

$$assert(s) = \bigcup_{v \in V} \{\tau_1 = \tau_2 \mid (\tau_1, \tau_2 \in know_s(v)) \wedge (\tau_1 \in \mathcal{AF}(\mathcal{X}))\}$$

Определение 14. *Назовём сеть* приведенной, *если она не содержит:*

- *забытых вершин, т.е. вершин не содержащих ни одного элемента;*
- *недоступных вершин, т.е. вершин, входящих в компоненты связности графа сети, которые не содержат переменных;*
- *некорректных связей, т.е. цепей по элементам $c : \Psi(c) \in \mathcal{AF}$, не оканчивающихся переменными (адресными термами).*

Определение 15. *Сети s и s' называются равными, если существует биективное отображение $ID : V_s \to V_{s'}$, а также биективные отображения $id_v : v \to ID(v), \forall v \in V_s$ такие, что:*

$$\forall v \in V_s, \forall c \in v \ : \ \Psi_{s'}(id_v(c)) = \Psi_s(c) \wedge \forall i \ \Gamma_{s'}(id_v(c), i) = ID(\Gamma_s(c, i))$$

Обозначим через $\mathbb{O}$ сеть с $V_{\mathbb{O}} = \varnothing$

Пусть у нас есть две сети $s = \langle C, V, \Psi, \Gamma \rangle, s' = \langle C', V', \Psi', \Gamma' \rangle$, определим произведение сетей $s \times s' = \langle C'', V'', \Psi'', \Gamma'' \rangle$:

- $V'' = V \times V'$ — декартово произведение множеств вершин.

$$(v, v') = \{(c, c') \mid c \in v, c' \in v', \Psi(c) = \Psi(c')\}.$$

- $C'' = \{c | c \in v, v \in V''\}$
- $\Psi''((c, c')) = \Psi(c)$
- $\Gamma''((c, c'), i) = (\Gamma(c, i), \Gamma'(c', i)) \ \ \forall i \ (1 \leqslant i \leqslant d(\Psi(c)))$

Лемма 1 (О приведенной сети). *По всякой сети s можно построить приведенную сеть $red(s)$ такую, что $assert(s) = assert(red(s))$.*

Лемма 2 (О сужении знания вершин). *Для любых двух приведенных сетей $s = \langle C_s, V_s, \Psi_s, \Gamma_s \rangle$ и $s' = \langle C_{s'}, V_{s'}, \Psi_{s'}, \Gamma_{s'} \rangle$ верно, что $\forall v \in V_s, \forall v' \in V_{s'}$*

$$know_{s \times s'}((v, v')) = know_s(v) \cap know_{s'}(v').$$

Лемма 3 (Единственность приведенной сети). *Для приведенных сетей $s = s' \Leftrightarrow assert(s) = assert(s')$*

Определим операцию пересечения двух приведенных сетей:

$$s \sqcap s' = red(s \times s')$$

Лемма 4 (О пересечении приведенный сетей). *Для двух приведенных сетей s и s' справедливо следующее утверждение:*

$$assert(s \sqcap s') = assert(s) \cap assert(s')$$

Введем понятие *веса* сети.

Определение 16. *Весом сети s назовем пару $\omega(s) = \langle k_1, k_2 \rangle$, где:*

$$k_1 = 2e_X + l_F + e_{AF} - l_1 - l_2$$
$$k_2 = l_{F*}$$

e_X — количество элементов-переменных сети s, e_{AF} — количество элементов из $\mathcal{AF}$ сети s, l_F — количество вершин с элементами из $\mathcal{F}$ сети s, l_{F*} — количество вершин сети s, имеющих элементы — $*$, а также элементы из $\mathcal{F} \cup \mathcal{A}$, l_1 — количество вершин сети s, имеющих хотя бы один элемент-переменную, l_2 — количество вершин сети s, у которых все элементы — переменные. Порядок весов — обратный лексикографический.

Лемма 5 (О монотонном убывании веса). *Для приведенных сетей s и s' верно, что*

$$\omega(s \sqcap s') \leqslant min\,(\omega(s), \omega(s'))$$

Теорема 2. *Множество приведенных сетей вместе с операцией пересечения образует ограниченную полурешетку.*

Обозначим через $\mathcal{L}$ множество приведенных сетей. Дополним его искусственным элементом $\mathbb{I}$, для которого положим по определению $\forall s \in \mathcal{L}$: $s \sqcap \mathbb{I} = \mathbb{I} \sqcap s = s$. Получаем ограниченную полурешетку $(\mathcal{L}, \sqcap)$.

Определим операцию add, которая по сети s и терму τ строит сеть $add(s, \tau)$, в которой есть вершина, знающая τ. Пусть $s = \mathbb{I}$ или сеть s уже содержит вершину, знающую терм τ, положим тогда $add(s, \tau) = s$.

В противном случае пусть $\tau = f(\tau_1, \ldots, \tau_n), n \geqslant 0, f \in \mathcal{X} \cup \mathcal{F} \cup \mathcal{AF}$. Построим тогда сети $s_0 = s, s_i = add(s_{i-1}, \tau_i), i = 1, \ldots n$ и получим $add(s, \tau)$ из сети s_n добавлением новой вершины v с единственным элементом c, для которого мы полагаем $\Psi(c) = f, \forall i\ (1 \leqslant i \leqslant n)\ \Gamma(c, i) = v_i$, где v_i — вершина, знающая терм τ_i.

Замечание 2. Отдельно рассмотрим случай с термами вида $\&\tau_1$. Если $\tau_1 \in \mathcal{X}$, то в сеть добавляется вершина, знающая терм a_x. Это не противоречит определению оператора &. Если τ_1 имеет вид $*\tau_2$, то рекурсивно переходим к операции $add(s, \tau_2)$. В остальных случаях добавляется терм $\perp$.

Определим операцию *эффект присваивания (ЭП)*. По заданному оператору присваивания $\langle \tau_1 := \tau_2 \rangle$ и сети s строим сеть $\langle \tau_1 := \tau_2 \rangle s$. Положим $\langle \tau_1 := \tau_2 \rangle \mathbb{I} = \mathbb{I}$. Для $s \neq \mathbb{I}$ сеть преобразуется в зависимости от τ_1. Пусть $\tau_1 \in \mathcal{X} (\tau_1 = x)$, тогда:

1. Сеть s_1 получается из сети $add(s, \tau_2)$ добавлением в вершину, знающую τ_2 элемента $c, \Psi_{s_1}(c) = z$, где z — переменная, не встречающаяся в $add(s, \tau_2)$.
2. Сеть s_2 строится из сети s_1 удалением элемента c, для которого $\Psi_{s_1}(c) = x$.
3. Сеть s_3 выводится из сети s_2 заменой функции означивания для элемента c с $\Psi_{s_2}(c) = z$ на $\Psi_{s_3}(c) = x$.

Пусть теперь $\tau_1 = t(\tau_3), t \in \mathcal{AF}$, тогда:

1. Сеть s_1 получается из сети $add(s, \tau_2)$ добавлением в вершину, знающую τ_2 элемента $c, \Psi_{s_1}(c) = z$, где z — переменная, не встречающаяся в $add(s, \tau_2)$.
2. Сеть s_2 строится из сети $s' = add(s_1, \tau_1)$ удалением элемента c из вершины, знающей τ_1, для которого $\Psi_{s'}(c) = t$ и $\Gamma(c, 1) = v$, где v — вершина, знающая терм τ_3.
3. Сеть s_3 выводится из сети s_2 заменой функции означивания для элемента c с $\Psi_{s_2}(c) = z$ на $\Psi_{s_3}(c) = t$, а также добавлением зависимости для элемента $c : \Gamma(c, 1) = v$, где v — вершина, знающая терм τ_3.

В обоих случаях сеть $\langle \tau_1 := \tau_2 \rangle s = red(s_3)$.

Определим операцию *эффект выделения памяти (ЭВП)*. По заданному оператору $\langle new(\tau) \rangle$ и сети s строим сеть $\langle new(\tau) \rangle s$. Положим $\langle new(\tau) \rangle \mathbb{I} = \mathbb{I}$. Для $s \neq \mathbb{I}$ сеть преобразуется в зависимости от τ.

1. Сеть s_1 получается из сети s добавлением новой вершины v_{new}, с элементом $c, \Psi(c) = a, a$ — адресная константа, соответствующая функционированию оператора new.

2. Сеть s_2 строится из сети $add(s_1, \bot)$ добавлением в вершину, знающую $\bot$ элемента $c, \Psi(c) = *, \Gamma(c, 1) = v_{new}$.

3. Далее возможны два случая:

   (a) $\tau \in \mathcal{X}$, тогда сеть s_3 выводится из сети s_2 переносом элемента c, для которого $\Psi(c) = x$ из его вершины в v_{new}.

   (b) $\tau = *(\tau')$, тогда сеть s_3 получается из $s' = add(s_2, \tau)$ переносом элемента c из вершины, знающей τ, для которого $\Psi(c) = *$ в вершину v_{new}.

4. Сеть $\langle new(\tau)\rangle s = red(s_3)$

Лемма 6 (О дистрибутивности операций)**.**

$$\forall s_1, s_2 \in \mathcal{L}, \forall \tau, \tau_1 \in \mathcal{AF}(\mathcal{X}), \forall \tau_2 \in Term_\Sigma :$$
$$\langle new(\tau)\rangle (s_1 \sqcap s_2) = (\langle new(\tau)\rangle s_1) \sqcap (\langle new(\tau)\rangle s_2), \quad (1)$$
$$\langle \tau_1 := \tau_2\rangle (s_1 \sqcap s_2) = (\langle \tau_1 := \tau_2\rangle s_1) \sqcap (\langle \tau_1 := \tau_2\rangle s_2). \quad (2)$$

Лемма 7. *Пусть $s \in \mathcal{L}$ и $gen(G, w') = assert(s)$ для пути w' во фрагменте G. Тогда $gen(G, w) = assert(\langle \tau_1 := \tau_2\rangle, s)$ для пути $w = w'Ae$, где A — оператор присваивания $\tau_1 := \tau_2$.*

Лемма 8. *Пусть $s \in \mathcal{L}$ и $gen(G, w') = assert(s)$ для пути w' во фрагменте G. Тогда $gen(G, w) = assert(\langle new(\tau)\rangle, s)$ для пути $w = w'Ae$, где A — оператор присваивания $new(\tau)$.*

Рассмотрим задачу глобального анализа. Подробно задача и методы её решения описаны в [1, гл. 2]. Выделим следующее справедливое утверждение.

Теорема 3. *Если в окружении для анализа все преобразования свойств — дистрибутивны, то стационарная разметка μ_c дает точное, единственное решение задачи глобального анализа, т.е.:*

$$\forall e \in E : \mu_c(e) = H(e)$$

Теперь можно сформулировать задачу глобального анализа фрагмента G. В качестве окружения для анализа возьмём $(\mathcal{L}, \sqcap, F)$, где F содержит все операции ЭП и ЭВП для соответствующих операторов фрагмента G, а так же функции $i(s) = s$ и $\mathbb{I}(s) = \mathbb{I}$.

Семантику $f_e^{e'}$ (функцию перехода вершину между дугами e' и e) определим следующим образом:

$$f_e^{e'}(s) = \begin{cases} \langle \tau_1 := \tau_2 \rangle s, \ e' \text{ ведёт к оператору } \tau_1 := \tau_2, \text{ а } e \text{ выходит из него,} \\ \langle new(\tau) \rangle s, \ e' \text{ ведёт к оператору } new(\tau), \text{ а } e \text{ выходит из него,} \\ i(s), \ e' \text{ ведёт к распознавателю, а } e \text{ выходит из него,} \\ \mathbb{I}(s), \ \text{в остальных случаях.} \end{cases}$$

Для входа e с результатами $\{x_1, \ldots, x_n\}$ строим s_e — сеть, содержащую n вершин, каждая из которых содержит один элемент $c_i \in v_i$, $\Psi_{s_e}(c_i) = x_i$, $(1 \leqslant i \leqslant n)$. В качестве начальной разметки $\mu_0(e) = s_e$ для входов e фрагмента G и $\mu_0(e) = \mathbb{I}$ для остальных дуг. Тогда из теоремы 2 и лемм 6, 7, 8 следует

Теорема 4. *Для стационарной разметки μ, сформулированной задачи глобального анализа, и для всех дуг e фрагмента G*

$$invar(G, e) = assert(\mu(e)).$$

5. Алгоритм распознавания логико-термальной эквивалентности

Введем систему преобразований фрагментов Δ_{lt}.

T1 *Удаление недостижимых.* Фрагмент, не имеющий входов, равносилен пустому фрагменту.

T2 *Стягивание тупиков.* Фрагмент, без выходов и без заключительных операторов, с номерами входов $a_1, \ldots, a_n$ равносилен петле со входными дугами — входами $a_1, \ldots, a_n$.

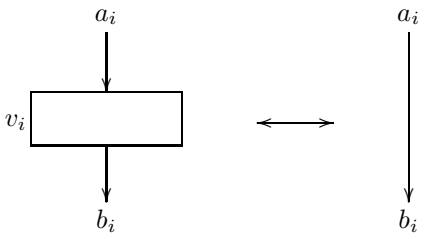
T3 *Склеивание копий, копирование.* Возьмем разбиение множества вершин фрагмента G на непустые, непересекающиеся классы $K_1, \ldots, K_n$ удовлетворяющие следующим свойствам:

1. каждый класс содержит графически совпадающие операторы;
2. если некоторая выходная дуга вершины класса K_i ведет к вершине класса K_j, то соответствующие выходные дуги всех других вершин класса K_i так же ведут к вершинам класса K_j;
3. если некоторая выходная дуга вершины класса K_i является выходом фрагмента G с номером b, то соответствующие выходные дуги всех других вершин класса K_i — также выходы фрагмента G с номером b.

Пусть фрагмент G' может быть получен из G следующим образом: из каждого класса $K_i (i = 1, \ldots, n)$ берется ровно по одной вершине v_i, и дуга от нее либо проводится к вершине v_j (условие 2), либо объявляется выходом G' с номером b (условие 3). Если в G имеется вход с номером a, ведущий к вершине класса K_r, то к G' добавляется вход с номером a, который направляется к вершине v_r. При перечисленных условиях $G \leftrightarrow G'$

T4 *Замена переменных.* Пусть переменные x и y фрагмента G не зацеплены, т.е. не существует дуги фрагмента G, чтобы обе эти переменные входили в сеть инвариантов, приписанных этой дуге. А также ни одно из вхождений переменной x не является результатом входа или аргументом выхода. Тогда $G \leftrightarrow G'$, где G' — фрагмент, получающийся из G заменой всех вхождений x в компоненту Γ на переменную y.

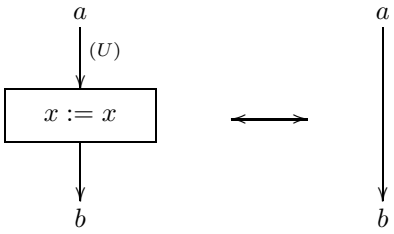
T5 *Удаление неиспользуемых преобразователей.* Пусть $\{v_i\}_{i=1}^n$ — такое подмножество преобразователей фрагмента G, что к вершине v_i ведет ровно одна дуга и вершина v_i не влияет ни на один из выходов фрагмента G, а так же ни на одну из вершин, отличных от $v_1, \ldots, v_n$. Фрагмент G' получается из G заменой каждого преобразователя $v_i (i = 1, \ldots, n)$ на дугу:



Тогда $G \leftrightarrow G'$.

T6 *Удаление неиспользуемого результата входа.* Если результат $x \in \mathcal{AF}(\mathcal{X})$ входа e фрагмента G не влияет на его вершины и выходы, то $G \leftrightarrow G'$, где фрагмент G' получается исключением x из множества результатов входа e.

T7 *Удаление вырожденной пересылки.* Если $x \in \mathcal{AF}(\mathcal{X}) \cap U$, то



T8 *Замена термов.* Пусть инварианты всех дуг, ведущих к вершине v во фрагменте G, содержат равенство $\tau_1 = \tau_2$. Тогда $G \leftrightarrow G'$, где G' получается из G заменой терма τ_1 на τ_2.

Теорема 5. $G_1 \overset{\Delta_{lt}}{\longleftrightarrow} G_2 \Rightarrow G_1 \overset{lt}{\sim} G_2$

Определение 17. *Фрагмент G называется* приведенным, *если для него выполнены следующие условия:*

R1 *Каждая вершина фрагмента G лежит хотя бы на одном пути, начинающимся входом фрагмента G.*

R2 *От каждой вершины фрагмента G (кроме операторов петли) имеется хотя бы один путь к выходу из G или к заключительному оператору.*

R3 *К каждому преобразователю, заключительному оператору или оператору петли ведет ровно одна дуга фрагмента G.*

R4 *В качестве функциональных подтермов в атомах и заключительных операторах фрагмента G используются только переменные.*

Лемма 9. *Применением правил **T1** — **T3**, **T5**, **T8** всякий фрагмент можно преобразовать в приведенный.*

Определение 18. Линейным участком (лучом) *называется всякий приведенный фрагмент без распознавателей и ровно с одним входом.*

Определение 19. Выходным лучом *приведенного фрагмента G называется вхождение в G всякого максимального луча, кончающегося выходом фрагмента G.*

Со всяким приведенным фрагментом G свяжем его *логический граф (л-граф)* $LG(G)$, который получается из G следующим образом:

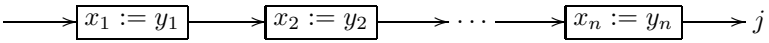
каждый преобразователь $a \longrightarrow \boxed{v} \longrightarrow b$ заменяется фрагментом-дугой $a \longrightarrow b$, после чего удаляются аргументы вершин и выходов, а также результаты входов.

С каждой конечной цепочкой w в л-графе (путем от входа к выходу или к заключительной вершине) свяжем слово, которое получается последовательным выписыванием номеров свободных дуг, а так же символов, приписанных вершинам цепочки w. При этом предикатный символ берется со знаком $\delta(\delta \in \{0,1\})$, если путь продолжается по δ-дуге некоторого распознавателя. Множество слов, построенных таким образом во по всем конечным цепочкам л-графа, называется *языком* л-графа. Два л-графа называются *автоматно-эквивалентными*, если их языки совпадают. Два приведенных фрагмента называются *подобными*, если их л-графы автоматно-эквивалентны. Из приведенных определений следует, что всякие lt-эквивалентные фрагменты являются подобными. Отношение подобия в точности соответствует эквивалентности одноленточных автоматов. Для распознавания эквивалентности двух л-графов достаточно применить алгоритм Мура распознавания эквивалентности одноленточных автоматов.

Пара приведенных фрагментов G_1, G_2 называется согласованной, если для нее выполнены следующие условия:

C1 Множества результатов входов с одинаковыми номерами совпадают

C2 Всякий выходной луч в G_1 и G_2 завершается так называемым выходным вектором пересылок, т.е. лучом



где $\{x_1, \ldots, x_n\}$ — множество всех аргументов выхода с номером j, а также $\forall i (1 \leqslant i \leqslant n)\ y_i \notin \{x_1, \ldots, x_n\}$.

C3 Если некоторая переменная x встречается в каждом из фрагментов пары, то преобразователи с результатом x имеются только внутри выходных векторов пересылок этих фрагментов.

C4 Логические графы фрагментов G_1 и G_2 совпадают.

Теорема 6. *Применением правил из Δ_{lt} всякую пару lt-эквивалентных фрагментов можно преобразовать в согласованную пару.*

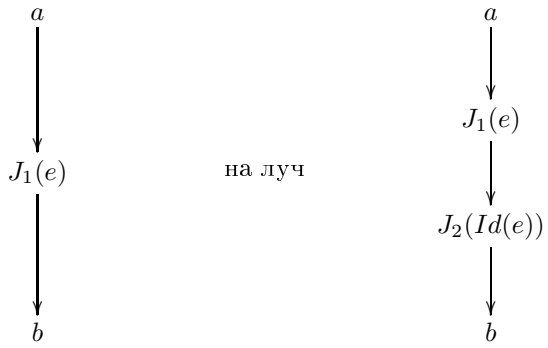
Теорема 7. *Система Δ_{lt} — полная система lt-эквивалентных преобразований фрагментов, т.е.*

$$G_1 \overset{lt}{\sim} G_2 \Leftrightarrow G_1 \overset{\Delta_{lt}}{\longleftrightarrow} G_2$$

Доказательство. С учетом теорем 5 и 6 достаточно показать $G_1 \overset{\Delta_{lt}}{\longleftrightarrow} G_2$ для фрагментов $G_1, G_2, G_1 \overset{lt}{\sim} G_2$, составляющих согласованную пару.

Обозначим через $J_i(e)$ луч, начинающийся δ-дугой e распознавателя или входом e во фрагменте $G_i (i = 1, 2)$ и заканчивающийся дугой, ведущей к распознавателю, оператору петли, заключительному оператору или к первой из пересылок выходного вектора пересылок этого фрагмента; $Id : zakh_i vanov.tex, v1.42006/12/1315 : 59 : 10 StasFominExp$ — взаимно однозначное соответствие входов, заключительных операторов, операторов петли, распознавателей и их выходных дуг, которое существует благодаря совпадению л-графов G_1 и G_2.

Преобразуем фрагмент G_1 во фрагмент $\widehat{G}_1$, заменяя в нем всякое вхождение луча



Это преобразование можно выполнить однократным применением **T5**, поскольку в силу условия согласованности **C3** результаты добавляемых преобразований не встречаются в исходном фрагменте G_1, а потому и не могут влиять на распознаватели, заключительные операторы и выходы в $\widehat{G}_1$.

Пусть теперь v — произвольный распознаватель, заключительный оператор или одна из пересылок выходного вектора пересылок в $\widehat{G}_1$, x — ее i-й аргумент, а y — i-й аргумент вершины $Id(v)$ во фрагменте G_2. Покажем, что тогда инварианты всех дуг e, ведущих к v в $\widehat{G}_1$, содержат равенство $x = y$. Действительно, предположение $\{x = y\} \notin invar(\widehat{G}_1, e)$ означает существование такого пути w, начинающегося некоторым входом и кончающегося дугой e в $\widehat{G}_1$, что $vl(w, x) \neq vl(w, y)$. Последнее, в свою очередь, означает, что фрагменты G_1 и G_2 не являются логико-термально эквивалентными, т.е. мы получили противоречия с исходным предположением.

Итак, применением **T8** i-й аргумент каждой из вершин-распознавателей, заключительных операторов и выходных пересылок v в $\widehat{G}_1$ можно заменить на соответствующий аргумент $Id(v)$ из G_2. В результате такого преобразования на каждом из лучей

$$a \to J_1(e) \to J_2(Id(e)) \to b$$

в $\widehat{G}_1$ преобразователи из $J_1(e)$ становятся неиспользуемыми, и их можно удалить применением схемы правил **T5**. После этого преобразования фрагмент $\widehat{G}_1$ окажется совпадающим с фрагментом G_2. □

Следствие 3. *Существует алгоритм распознавания lt-эквивалентности для схем МДП (фрагментов).*

Доказательство.

Шаг 1: Преобразуем исходные фрагменты G_1 и G_2 в приведенные фрагменты.

Шаг 2: Распознаем подобие полученных фрагментов. Если они не подобны, то $G_1 \overset{lt}{\nsim} G_2$ и алгоритм останавливается с отрицательным ответом. Если же фрагменты подобны, переходим к следующему шагу.

Шаг 3: Строим согласованную пару фрагментов $G_1' = mult(G_1, G_2)$ и $G_2' = mult(G_2, G_1)$ (теорема 6).

Шаг 4: По G_1' и G_2' строим фрагмент $\widehat{G}_1$ (см. доказательство теоремы 7), а затем его стационарную разметку.

Шаг 5: Если не выполнены условия некоторых из упомянутых в доказательстве теоремы 7 применений схемы правил **T8**, то $G_1 \overset{lt}{\nsim} G_2$ и алгоритм останавливается с отрицательным ответом, в противном случае $G_1 \overset{lt}{\sim} G_2$ и алгоритм останавливается с положительным ответом.

□

6. Заключение

Описанный в разделе 5 алгоритм распознавания логико-термальной эквивалентности для модели программ с динамической памятью является полиномиальным. Грубо оценить сверху время работы алгоритма можно величиной $O(n^8)$, где n — размер схемы программы.

Достигнутые результаты позволят в дальнейшем решить ряд важных задач: расширить модель операциями над адресами, строить и выяснять эквивалентность систем преобразований программ, обосновывать корректность работы алгоритмов анализа программ и алгоритмов обфускации, обосновывать стойкость алгоритмов обфускации, вводить метрики на программах, позволяющие выражать стойкость алгоритмов обфускации и информативность алгоритмов анализа.

Литература

[1] Котов В.Е., Сабельфельд В.К. Теория схем программ. Москва, «Наука», 2001.

[2] Zamulin A. A State-Based Semantics of a Pascal-like Language. Perspectives of System Informatics, «Эмари», Новосибирск, 2003, p. 47–63.

[3] Collberg C., Thomborson C., Low D., Manufacturing cheap, resilient and stealthy opaque constructs. In Proc. of the Symposium on Principles of Programming Languages, 1998, p.184-196.

[4] Collberg C., Thomborson C., Low D., A taxonomy of obfuscating transformations, Tech. Report, N 148, Dept. of Computer Science, Univ. of Auckland, 1997.

[5] Wroblewski G., General Method of Program Code Obfuscation, Wroclaw, 2002.

[6] Horwitz S. Precise Flow-Insensitive May-Alias Analysis is NP-Hard, *ACM Transactions on Programming Languages and Systems*, V. 19, N 1, 1997, p. 1–6.

[7] Ivanov K. S., Zakharov V. A. Program obfuscation as obstruction of program static analysis, Труды ИСП РАН, Том 6, 2004, с. 137-149.

[8] Andersen L.O. Program Analysis and Specialization for the C Programming Language, DIKU, University of Copenhagen, May 1994.

[9] Steensgaard B. Points-to analysis in almost linear time. In *Conference Record of the 23th Annual ACM Symposium on Principles of Programming Languages (POPL'96)*, p. 32–41. ACM Press, January 1996.

[10] Shapiro M., Horwitz S. Fast and accurate flow-insensitive point-to analysis. In *24th Annual ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages*, p. 1–14, January 1997.

[11] Hind M., Burke M., Carini P. Jong-Deok Choi, Interprocedural pointer alias analysis. ACM Trans. Programming Languages and Systems, V. 21, N 4, 1999, p.848–894.

[12] Barak B., Goldreich O., Impagliazzo R., Rudich S., Sahai A., Vedhan S., Yang K., On the (Im)possibility of obfuscating programs. CRYPTO'01 — Advances in Cryptology, Lecture Notes in Computer Science, 2139, p. 1-18, 2001.

Using data flow analysis for detecting security vulnerabilities

Andrey Belevantsev and Oleg Malikov
Institute for System Programming
Russian Academy of Sciences
Russia, 109004, Moscow, B. Kommunisticheskaya, 25
Phone: (+7-095) 912-07-54, Fax: (+7-095) 912-15-24
{abel,malikov}@ispras.ru

Abstract. This paper addresses the issues of using data flow analysis approach for detecting certain types of programming errors in the source code. The primary motivation of this work is that programming errors like a missing array or buffer bounds checking may lead to security vulnerabilities in programs. The article is devoted to the detection of *buffer overflow*, *unchecked input usage*, and *memory leak* errors.

A new approach for static detection of these kinds of errors is described. The approach combines forward data-flow intraprocedural alias and integer range analyses as well as flow-sensitive and partly context-sensitive interprocedural analysis. The algorithm is based on the notion of an *abstract memory location* (AML). The rules for computing attributes of AMLs as well as number of heuristics used to improve the algorithm are discussed.

The suggested approach has been implemented for checking C programs. The paper reports on evaluating the tool with a number of open source programs.

1. Introduction

Security vulnerabilities is a very important class of programming errors. According to the reports of SecurityFocus.com more than 3000 vulnerabilities have been discovered during 1.5 year starting from 01/2000. The most common are buffer overflow (23% to 50% [10, 13]), format string and tainted input errors.

A great number of already known security errors originates from the C language unsafe features and its standard library functions, e.g. unsafe pointer and array dereferences, pointer arithmetic and string handling routines. As it was noted in [10], even "safe" string primitives, such as `strncpy()`, encourage possible off-by-one bugs.

There are several approaches to classify the software security errors. A profound analysis of the fundamental taxonomies is given in [1] and [5]. We accept a widely used definition of the security vulnerability as a program error that causes incorrect program operation if a user feeds specially crafted data into the program. We distinguish between errors and corresponding vulnerabilities focusing on programming errors detection in particular. It often is a difficult problem to qualify an error as the security vulnerability. Moreover, an error, which is not vulnerable, still may turn to be a severe bug in the code.

It is known that software vendors allocate much resources to audit their code base for various kinds of possible security vulnerabilities. Manual code inspection and fixing is an expensive and time consuming solution to the problem. Existing publicly available tools for the automatic detection of programming errors are either not designed especially for finding security flaws, thus missing to report important error classes, or yield too many false positives (more than 90%). Several companies claim to possess such tools [14, 11], but use them internally in their auditing services. Many researchers are making a considerable effort towards automated static analysis tool for detecting security flaws with a relatively low number of false warnings [6, 10].

There is a number of tools and libraries, which are intended to detect attempts to exploit vulnerabilities at run time, as well as a number of hardware features aimed at protecting the system from hacking attempts. However, these features can either be bypassed or incur unacceptable running cost. The preferable way of anti-hacking protection is to fix all possible security vulnerability errors in the source code.

The source code programming errors can be detected either during static analysis or at run time during debugging, testing, etc. To detect programming errors during debugging or testing one has to provide input data that spots an error, which is a challenging task mainly because the exploitation of security vulnerabilities often needs non-trivial input data. For these reasons static analysis approach should be used.

Static analysis approach has its own limitations. Namely, it is impossible to detect all the errors in a program without false warnings. The false positives to the total number of detected errors ratio is a measure of quality of the static analysis algorithm. In our opinion, in order to achieve the best quality of analysis the algorithm should model operation of the program as close as possible. This includes tracking use of program data memory, memory allocations, buffer

accesses as well as tracking integer calculations in the program, for example, string lengths or array indices. The semantics of the standard library functions is also to be modeled.

1.1. Overview

The goal of our research is to develop new methods for automatic detection of various kinds of programming errors in existing (or legacy) programs. Another goal is to develop a tool for automatic detection of programming errors using this new approach. The tool should be applicable to real-world programs (hundreds and thousands of KLOC) and have reasonable false positives to the total number of detected errors ratio.

In this paper we concentrate on the buffer overflow, format string and memory leak programming errors. **Buffer overflow** errors occur as a result of missing or insufficient array bounds checking. **Format string** errors are results of using the unchecked input data as an actual parameter passed to the formatting input/output functions of the C standard library (`printf`, `fprintf`, etc). **Memory leaks** occur when a heap object is not deallocated and all the pointers to that object are lost in the running program.

The main contribution of the paper is a new method for finding certain kinds of programming errors, which combines interprocedural range and alias analysis. We show that the method can be practically used for error detection by implementing it in a prototype tool. In our opinion this is the first attempt to apply such combination of static analysis methods to the problem of automatic detection of programming errors.

Our approach uses the notion of *abstract memory location* (AML). An AML represents any object in memory, allocated either statically (variables, string constants) or dynamically (heap memory). Each AML possesses a number of attributes that reflects its properties useful for error detection. These attributes include, e.g *points-to* set (i.e., the set of AMLs to which the AML may point to); *size* (amount of bytes that occupied by the AML); or *length* (the length of the string stored in the AML).

The error detection process consists of two stages. In the first stage a classical iterative forward data-flow analysis algorithm [7] is used. The algorithm calculates the attributes of AMLs in every program point. It consists of several components as follows:

- Flow-sensitive intraprocedural analysis. This component makes both alias and integer range analysis, depending on the attributes of an AML it is calculating.
- Partially context-sensitive interprocedural analysis. We use a number of experimentally established tradeoffs to regulate context sensitivity of the analysis.
- Support for the major functions of the C standard library. The purpose of this part of the algorithm is to provide the information about the impact of the operational environment on the analyzed program.

The second stage performs verification of the attributes and issues warnings when certain conditions are not satisfied. The following warnings can be reported: buffer overflow, format string error, memory leak, use of undefined pointer value, dereferencing `NULL` pointer.

We have implemented the approach in the prototype tool for checking C programs. The preliminary evaluation of the tool on a number of open source programs shows promising results. Namely, in 4 packages we've analyzed the true positive rate ranges from 22.6% to 70.6%.

The rest of the paper is organized as follows. Section 2 introduces basic concepts of our analysis. Section 3 follows with a detailed description of the algorithm for attribute calculation. Section 4 describes the process of warnings generation. Section 5 continues with evaluation results of the implemented tool, Section 6 discusses related works. Section 7 gives the conclusion.

2. Basic concepts

In this section we describe the fundamental concepts and data structures used in our approach. First we define the lattices applied for tracking all integral computations performed in the program. We do not support floating-point calculations, because they rarely result in buffer overflow errors. Then we continue with a detail description of abstract memory locations and data structures concerned.

2.1. Data lattices

To model the integral types of the C language we use the integer lattice. The value set of the lattice consists of unlimited negative and positive integer numbers, $-\infty$, ∞, **undef**, **any**, and integer ranges $[l, h]$, $(-\infty, h]$, $[l, \infty)$, where l, h are integers. The integer lattice type is further denoted as $M_Integer$.

The approach of using integer ranges for tracking integral values in the program is also used in [3, 10]. Considering [10], we add "undefined" value **undef** and "overdefined" value **any** to the value set of the lattice. The **undef** value is assigned to a local variable at the location of its definition, if the definition has no initialization. The **any** value means that the value of the variable is completely indeterminable by static analysis algorithm. By definition $(-\infty, \infty) \equiv$ **any**.

We define the standard set of operations with integers and extend them to handle **undef** and **any** values. We also define the join ($\sqcup$) binary operation used in program locations where the control flow merges from several alternatives. We follow the [3] definition of join operation of two integral ranges as *bounding box* of these ranges.

2.2. Abstract memory locations

Each object, which may potentially exist during program execution, has the corresponding abstract memory location (AML) object during program analysis. AMLs for objects are created "lazily" when the corresponding variable declaration point of the program is analyzed. AMLs are created for static and automatic variables, string constants, dynamic data structures (allocated by `malloc()` and friends)and temporary variables of the intermediate representation of the program.

An AML is created for an aggregate type as the whole, as well as AMLs for individual fields of the aggregate type. These AMLs are linked together using the *overlap* AML attribute. For an array object two AMLs are created, the first represents the pointer to the array, whereas the second is the array itself. This allows us to handle array and pointer dereferences uniformly. Array elements are not distinguished and no separate AMLs are created for array elements. This is a trade-off decision to decrease resource and time consumption of the analysis. However, our experiments show little impact on the quality of analysis.

We use *(AML, offset)* to refer to an AML with the offset *offset*. Offsets have the *M_integer* type. By $AMLSet = \{(AML, offset)\}$ a set of such pairs is denoted.

An AML contains both static and dynamic attributes. The values of static attributes remain unchanged during the lifetime of the AML, whereas dynamic attributes are associated with some point of the program and make sense only in conjunction with it. There is a special AML **null**, which is referenced by any `NULL` pointer.

The static attributes of an AML are listed below:

- *size* is the amount of bytes occupied by the AML. For dynamically created AMLs the size is calculated using the corresponding parameter(s) of the memory allocation function, when it is possible. In this case the size may change during iterations of the analysis, as well as after the call to `realloc()`. In other words, the size of the AML is the same for all the points of the program, but may be adjusted dynamically.
- *overlap* is an *AMLSet* that contains the AMLs that overlap the current one. The attribute is used for specifying the layout of subobjects in the object of the aggregate type. Creation of subobject AMLs is done implicitly during creation of an aggregate AML, but this set can also be defined more precisely among the iterations. It happens due to impossibility of the correct determination of subobjects of dynamically created aggregate AML.
- *label* denotes the name of the AML.
- *type* denotes the type of the AML.
- *var* denotes the original variable that initiated the creation of the AML.

The dynamic attributes of an AML (denoted by *AMLAttributes*) are as follows:

- *aset* is an *AMLSet* of the objects which the AML may point to (i.e, points-to set).
- *len* is *M_integer* denoting the length of the string contained in the AML.
- *value* is *M_integer* denoting possible values of AMLs of integral types.
- *input* is a boolean attribute that shows whether the AML is "tainted", i.e. its value depends on user input.
- *alive* is a boolean attribute denoting that the memory allocated for this AML was not freed.

Note that the dynamic attributes of an AML are always bound to a specific program point.

3. Calculation of attributes

This section contains detailed description of the rules for an AML attribute calculation. Both intra- and interprocedural stages of the algorithm are discussed.

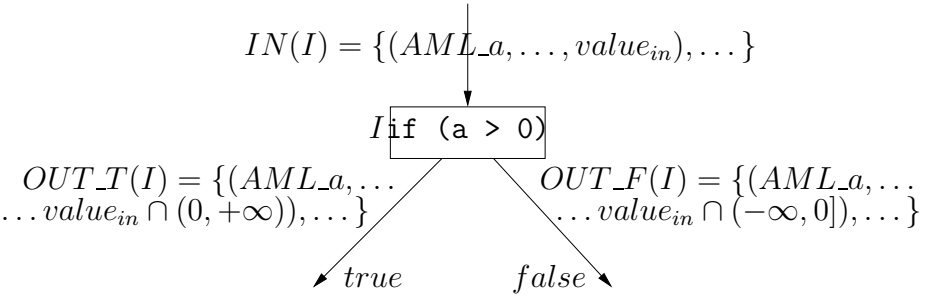


Figure 1: An example of the flow function calculation

There is also a number of heuristics used to improve speed and convergence of the analysis presented.

3.1. Intraprocedural analysis

The algorithm operates on the medium level of the intermediate representation of the program, generated by the C front-end. We use medium-level representation in form of quadruples $\langle opcode, res, op_1, op_2 \rangle$. Structured statements are converted to conditional and unconditional gotos. So-called *pseudoregisters* are used to hold temporary values.

The algorithm annotates each instruction I of the program with additional attributes. We denote the type of these attributes as *AML2Attr*. Each attribute of the type *AML2Attr* is a set of pairs $\{(AML, AMLAttributes)\}$. The instruction attribute $IN(I)$ is a set of attributes that are available at the point before the instruction I. If I is a single-exit instruction, $OUT(I)$ is the set of attributes available at the point after the instruction I.

When I is a conditional jump instruction, two sets *OUT_ T(I)* and *OUT_ F(I)* are defined and they contain the attributes effective when the condition holds and when the condition fails respectively. Similarly, if the instruction I has several exits, all of them have different attribute sets. In the Fig. 1 an example is given. AML_a is an AML corresponding to the variable a. At the entry of the instruction I this AML has the attribute $value = value_{in}$. At the exits of I the attributes will be as follows: $value_{out_t} = value_{in} \cap (0, +\infty)$, $value_{out_f} = value_{in} \cap (-\infty, 0]$.

$IN(I)$ and $OUT(I)$ ($OUT_T(I)$, $OUT_F(I)$) are connected by the flow equation:

$$
\begin{aligned}
&OUT(I) = Flow(I, IN(I)), \text{ or} \\
&(OUT_T(I),\ OUT_F(I)) = Flow(I, IN(I)) \\
&IN(I) = \bigsqcup_{p \in Pred(I)} OUT(P)
\end{aligned}
$$

Joining of $AML2Attr$ sets is performed as follows. For each $x = (AML, Attrib1)$ taken from $OUT(P)$, if $IN(I)$ does not contain $y = (AML, Attrib2)$ with the same AML, x is added to $IN(I)$. Otherwise, $Attrib1$ and $Attrib2$ are joined.

The flow function is defined for each instruction of the intermediate representation and is the core of the attribute calculation algorithm. For example, access via pointer instructions modifies *aset* and/or *len* attributes, arithmetic instructions modify *value* attributes, etc. Conditional jump instructions also modify attributes.

The function call instructions are handled in a special way. If the function being called is a standard library function (for example, `strcpy`), a special routine is called, which simulates the behavior of this library function and generates the output attribute set $OUT(I)$ from the current input attribute set $IN(I)$, i. e. handling of the standard library functions is always context sensitive. If the function is not a predefined function, yet its body is available, the analysis of the function is not performed immediately, and the current output attribute set $OUT(I)$ is used. It is the duty of the interprocedural analysis framework to analyze the function and to update its output attributes. If the callee is not a predefined function and its body is not available, the analysis algorithm has to use conservative assumptions concerning the function semantics.

Special support for loops is also implemented in order to ensure algorithm convergence and improve its precision. The loop counter variable is detected and its value range is set according to loop condition. All the further changes of the attributes of the loop variable are prohibited. This heuristics work for loops of simple structure, and still complex loops may not converge. A number of other tricks is used in this case.

3.2. Interprocedural analysis

A context-insensitive program call graph is constructed at the beginning of the analysis. Initially the call graph does not contain edges corresponding to call-by-pointer instructions, such edges are added when the corresponding points-to sets are computed.

The current implementation features only partially context-sensitive interprocedural analysis. On each iteration each function is analyzed only once with joint context (join of all the input attribute sets at all the call sites), thus the analysis is mostly context-insensitive. However, calculation of the flow function for a call instruction does some additional passes to make analysis more sensitive. For example, all the dynamic AML attributes, which cannot exist at the current path in the call graph, are killed.

The interprocedural analysis uses the worklist approach. Initially the worklist of functions to be analyzed contains all the dangling nodes of the call graph, i.e. all the functions which are not called from any other function. On each iteration each function f in the worklist is analyzed once as follows:

1. All the input sets $IN(I)$ of all the call sites of f are joined. Let IN_1 be the result of join operation.

2. All the pairs $(AML, Attributes)$ in IN_1 are replaced with (AML_1, $Attributes$). Here AML corresponds to an actual argument of the function at some call site, AML_1 corresponds to the respective formal parameter of the function. If several AMLs map onto the same AML_1, their attributes are joined and the result is assigned to the attributes of AML_1. The attributes of the actual parameters without an AML (for example, constants) are also joined to the attributes of the corresponding formal parameters. Let IN_2 be the resulting set of attributes.

3. The pairs $(AML, Attributes)$ are removed from IN_2 for all the AMLs, which are not directly or indirectly accessible from the current scope. Such attributes are useless during the analysis of f. Let IN_3 be the result of the operation.

4. Forward iterative intraprocedural data-flow analysis is performed with a set of input attributes IN_3 for the f function.

5. The pairs $(AML, Attributes)$ are removed from the output set of attributes of f for all the AML, which are not directly of indirectly accessible outside the f.

6. If the intraprocedural analysis performed at the step 4 has changed attributes of any instruction within f, then all the callers of f are marked to be reanalyzed on the next iteration. A callee of f is marked to be reanalyzed, if input attributes of the corresponding call instruction have changed.

3.3. Additional analysis heuristics

We have implemented a number of additional analysis heuristics to increase the analysis speed, reduce memory usage, and improve the precision of the analysis. These features are listed below.

- Loop widening and narrowing [3] is performed to ensure convergence of the analysis in typical situations.

- Context clipping is performed at the points of conditional jump instructions. The context is updated on both exits according to the condition tested.

- Attributes of the source and the destination AMLs of *transfer* instructions are linked instead of copying. A transfer instruction assigns attributes of a source AML to attributes of a destination AML without changes. This feature allows effective cutting of AML attributes later.

- The exits of the function are distinguished. The contexts at the function exit points may differ from each other providing different attribute sets to the callers. When the result of the function call is tested, the condition is propagated back to the function and the appropriate return statement is chosen, thus selecting specific attribute set. This is a trade-off, which allows increasing context-sensitivity without impacting on the performance.

  The code snippet below is an example of this heuristic. On the "then" branch of `if` (line 14) the value of `p` is undefined, but on the "else" branch (line 15) `p` points to the object of size 10.

```
1  int f(char ** p) {
2      char * pp = malloc(10);
3      if (pp == NULL)
4          return 0;
5      else {
6          *p = pp;
7          return 1;
8      }
9  }
10
11 int main() {
12     char *p;
13     if (f(&p) == 0)
14         return 0;
15     p[1] = 0;
16     ...
17 }
```

4. Error detection

In two previous sections we described the attribute computation algorithm. The programming errors are detected using the computed AML attributes. For each instruction of the program the input context is checked and the attribute values, which can indicate programming errors, are determined.

Buffer overflow warning messages are generated as a result of comparing *size* attribute of the destination AML with *offset* attribute of pointers to this AML. If the offset is negative or greater than the size of the AML, buffer overflow warning is issued.

Detection of memory leaks is implemented as follows. At the end of each function all the pairs (*AML*, *AMLAttributes*), which have *alive* attribute set to *true*, are checked. *AML* must correspond to some dynamically allocated data structure. If such pair is to be removed from the output sets according to step 5 of interprocedural analysis, then all the points of memory allocation for this AML are marked as potential sources of memory leaks, and the corresponding warnings are issued.

Potential format string errors are detected at the call sites of the corresponding library functions. The format argument of the function is checked for taintedness. The argument is tainted, if the *input* attribute of the corresponding AML is set to *true*. In such cases warning messages are also issued.

Dereferencing of undefined or possibly `NULL` pointers is also checked and appropriate warning messages are generated.

5. Experimental results

We have implemented our approach as a prototype tool for checking C programs. The tool uses the C compiler and is implemented in the framework of the Integrated Research Environment (IRE) developed at Moscow State University. The compiler translates C code to the medium-level intermediate representation. The IRE contains a number of standard and original analysis and transformation algorithms. All of them operate over the intermediate representation. The IRE provides graphical user interface and project support.

For understanding features of our prototype implementation, consider this example.

```
1  #define BSIZE 32
2
3  void copy_string(char *dst, const char *src,
4    int *cur, unsigned cnt) {
5    int i = 0;
6    for (; i < cnt; i++)
7          dst[*cur + i] = src[i];
8    *cur += i;
9  }
10
11 void fix_middle(char *dst, int *cur) {
12   dst[(*cur)++] = ' ';
13   dst[(*cur)++] = '=';
14   dst[(*cur)++] = ' ';
15 }
16
17 void do_cat(char *dst, const char *src1,
18   const char *src2, int *cur, unsigned cnt) {
19   copy_string(dst, src1, cur, cnt);
20   fix_middle(dst, cur);
21   copy_string(dst, src2, cur, cnt);
22 }
23
24 int main(void) {
25   char str1[BSIZE];
26   char str2[BSIZE];
27   char *str3;
28   int i = 0;
29   j = sizeof(str1) + sizeof(str2) + 3;
30   str3 = (char*) malloc(j);
31   if (str3) {
32     do_cat(str3, str1, str2, &i, sizeof str1);
33     printf("%d %d\n", i, j);
34     str3[i] = 0;
35   }
36   return 0;
37 }
```

Table 1: Evaluation results for our tool

Application	Source LOCs	True positives	Total warnings	% of true positives
bftpd-1.0.24	3389	19	60	31.7%
lhttpd-0.1	1141	7	19	36.8%
pcre-3.9	9102	7	31	22.6%
surfboard-1.1.8	823	12	17	70.6%

It contains buffer overflow in line 32. Index variable `i` is changed through calling `do_cat()` function when passing its address as a parameter. Both calls of `copy_string()` increase the variable by `BSIZE`, and `fix_middle()` increases it by 3. Then the value of this variable equals the allocated size for `str3` buffer. Thus, an off-by-one error occurs. Note that neither FlexeLint [15], nor ITS4 [9] and RATS [12] detect this error.

We delivered preliminary evaluation of the tool on a number of open source programs. Several examples are taken from [6]. Evaluation results are summarized in the Table 1. For each package we show the size of its source code as reported by the `wc` tool, total number of warnings generated, a number of true positives and the percentage of true positives.

There is relatively high percentage of false positives due to complex evaluations of array indices, which our analysis is not yet able to cover. Other major causes of false warnings include insufficient handling of calls by pointer, insufficient standard library support and imprecision of the analysis in certain situations.

6. Related work

There are many methods and tools applied for automatic detection of programming errors. We have only mentioned approaches based on program verification (see [8] for an overview) and run-time checking such as [4]. Below we focus on the existing static analyses and tools.

There are tools that use lexical source code analysis to detect security errors (for example, [9, 12]). The main advantage of such tools is the speed of the analysis. However, lexical analysis tools lack accuracy and gives a very large number of false warnings.

Static analysis tools of the `lint` family such as Splint and FlexeLint [16, 15] use semantic analysis and simple forms of data flow analysis to detect many common C coding errors. The tools can detect buffer overflow and format string errors in simple situations, but more complex situations are either missed or yield false positives.

Bush et al. [2] use static analysis for finding common dynamic programming errors, such as dereferencing `NULL` pointers, memory leaks, etc. The analyzer simulates execution paths by modeling a memory state of a program and identifying inconsistencies. Models for system library functions are provided with the analyzer, whereas models for user functions are generated from the source. The tool does not aim at finding security errors such as buffer overflow and format string errors, whereas our approach is designed to detect these kinds of errors too. Both approaches detect memory leak and `NULL` pointer dereferencing errors.

Wagner et al. [10] develop static range analysis technique for finding buffer overflows. The approach consists in generating constraints for modeling string and buffer operations and then solving the constraints with range analysis. The tool implemented finds one real error in 10 warnings generated. The main limitations of the approach are insufficient handling of aliases (both pointers and unions) and flow- and context-insensitivity. Our approach handles pointer operations with flow sensitivity. That allows us to detect buffer overflows resulting from primitive pointer operations, which is not performed in [10]. However, we use some ideas from [10] and [3] for our implementation of integer range analysis.

Livshits and Lam [6] proposed an alias analysis to be used expressly for error detection. The approach introduces a new representation, called IPSSA, for capturing pointer dereferences and procedure calls in SSA form. The algorithm presented is a hybrid approach, using context- and path-sensitive analysis in the key situations and fast imprecise analysis for all other references. However, error detector used in [6] is limited to discover buffer overflows resulting from putting user supplied string to a static buffer. Our approach performs tracking of integral values of array indices and string lengths. That allows us to detect buffer overflows resulting from handling strings as character arrays, which was not the aim of [6]. Nevertheless we think that the assumptions we have made to speed up the analysis and suppress false errors are to some extent similar to those of [6].

7. Conclusions

This paper describes an approach for using data flow analysis for detecting programming errors, especially those error kinds, which may lead to security vulnerabilities (buffer overflow and unchecked input usage errors). Our algorithm is based on flow- and partially context-sensitive interprocedural alias and integer range analyses.

We use our approach in a tool for automatic detection of security vulnerabilities, which is currently at the stage of prototype implementation. Preliminary evaluation of the tool yields the true positive rate from 22.6% to 70.6% on 4 tested packages, showing feasibility of our approach.

We may also state that the assumptions we have made to improve the precision of the tool match the semantics of many functions analyzed.

Future challenges for us include scaling up of the analysis to larger programs, as well as eliminating known drawbacks and further improving the precision of the tool. We also plan to implement the support for annotation language and perform thorough evaluation of the features introduced by annotations.

References

[1] T. Aslam. A Taxonomy of Security Faults in the Unix Operating System. M.S. thesis, Purdue University, 1995.

[2] W. R. Bush, J. D. Pincus, and D. J. Sielaff. A static analyzer for finding dynamic programming errors. In *Proceedings of Software Practice and Experience*, pages 775-802, 2000.

[3] P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *Proceedings of the Second International Symposium on Programming*, pages 106–130. Dunod, Paris, France, 1976.

[4] E. Haugh and M. Bishop. Testing C Programs for Buffer Overflow Vulnerabilities. In *Proceedings of the Network and Distributed System Security Symposium*, San Diego, CA, February 2003

[5] I. Krsul. Computer vulnerability analysis. Ph.D. thesis, Purdue University, West Lafayette, IN, May 1998.

[6] V. Benjamin Livshits and Monica S. Lam. Tracking Pointers with Path and Context Sensitivity for Bug Detection in C Programs. In *Proceedings of the 11th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE-11)*, September 2003.

[7] S. Muchnick. Advanced Compiler Design and Implementation. Morgan Kaufmann Publishers, 1997, 3rd ed.

[8] K. Rustan, M. Leino. Extended Static Checking: a Ten-Year Perspective. In *Proceedings of the Schloss Dagstuhl Tenth-Anniversary Conference*, Springer-Verlag, 2001

[9] J. Viega, J.T. Bloch, T. Kohno, and G. McGraw. ITS4: A Static Vulnerability Scanner for C and C++ Code. In *Proceedings of the 16th Annual Computer Security Applications Conference*, December 2000.

[10] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities. In *Proceedings of 7th Network and Distributed System Security Symposium*, Feb. 2000.

[11] @Stake, `http://www.atstake.com`

[12] RATS checker, `http://www.securesoftware.com`

[13] Bugtraq mailing list and vulnerability database, `http://www.securityfocus.com`.

[14] Secure Software, `http://www.securesoftware.com`

[15] Flexelint static checker, `http://www.gimpel.com`

[16] Splint static checker, `http://www.splint.org`

Guaranteed Slowdown, Generalized Encryption Scheme, and Function Sharing

Yury Lifshits[1]

Abstract. The goal of the paper is to construct mathematical abstractions of different aspects of real life software protection. We introduce three following notions: program slowdown, generalized encryption scheme and function sharing. These schemes allowed to discover new applications of such known ideas as trap-door functions and self-correcting programs.

1. Introduction

Software protection is very active research area now. It involves both practical approaches and theoretical investigation. In contrast to cryptography there is a lack of strict models for almost all aspects of program protection. In this paper we consider three notions motivated various intuitive protection ideas. We give a series of definitions examples and open questions around them.

There are a lot of practical tools, but the level of protection provided by them is unclear. In many cases we require an evidence that program is protected. So the main problem is to construct a *proof* of security. As soon as you start think about proof you immediately ask for formal model and definition of security.

In 2001 the general formal model for program protection (namely, *black-box security*) [3] was presented. However there are only few positive results [15, 11] in this framework. An informal conclusion of this study is that threats are different and in each case you need specific protection. Hence we ask for new formal models describing program protection and attacks on programs.

Our new approaches are the following.

First idea is *guaranteed slowdown* scheme. Guaranteed slowdown is a slow version of some algorithm which is difficult to speed up to the original level. One application is slowdown of encryption algorithms in public key cryptosystems. It helps against brute force attacks. In this work we introduce two examples of slowdown transformation. We show that idea of guaranteed slowdown is closely

[1]Support by grant RFBR 06-01-00584-a and contract with Federal Agency of Science and Innovation N 02.442.11.7290 are gratefully acknowledged

related to trap-door functions. The most interesting open question is how to get a formal proof that slowdown is difficult to reverse.

Second protection method we study is *generalized encryption scheme* (GES). This notion was already presented in work of Abadi et al. [1]. The problem is the following: how to use untrusted computational resource with information-theoretic security of your own data? Here we present one new example of such a scheme and discuss utilization of this notion in software protection. The main result of Abadi et al. [1] is that GES is impossible for NP-complete problems. Still it is very perspective to find such schemes for polynomial tasks.

Third notion is *function sharing*. This is a natural generalization of secret sharing schemes. The question is how to divide a computational task among several parties in the way such that subgroups can't say anything about the original task. This scheme are interesting for protecting mobile agents from malicious hosts [7, 10] development and as a basic block for new obfuscating transformations [6]. We present here a scheme for function sharing in the most simple model. For several additional restrictions existence of such schemes remains an open question.

Next three sections devoted to these models. For each of them we introduce some motivation, several examples, then formalization and conclude with related work and open questions.

2. Guaranteed slowdown

Informally, guaranteed slowdown is a pair of programs P_1 and P_2 with three following properties: P_1 and P_2 are functionally equivalent, P_1 is faster than P_2, and there is some "evidence" that given only source code of P_2 to construct a functionally equivalent program comparable (in speed) with P_1 is difficult.

The task of slowdown seems quite unusual so we need to explain our motivation. There are four points here:

A. Author of some program can distribute it in the slow version in order to protect his copyright. To prove the authorship of his code he (and only he) can show speedup version of the program.

B. We now explain how slowdown of encryption algorithm might be useful against brute force attacks on public key cryptosystem. Let Bob is sending message to Alice. Then to obtain the maximum level of security they choose the largest length of the key such that decryption and encryption algorithms are still feasible. Consider the case when decryption (using this long key) requires full resources from Alice while Bob has some re-

serve of computational power during encryption. In such circumstances instead of public key we can distribute slowdown version of encryption algorithm. Thus Bob still can encrypt what he want but any brute force attack that uses a lot of encryptions becomes harder.

C. It is popular to bound some functionalities in the trial versions of software. We can apply guaranteed slowdown scheme to obtain provable crack resistance for trial version. If we use slow version as a trial one, then by our informal definition recovering of original speed for the program is difficult.

D. We can divide all attacks on a program in the two classes: understanding (gaining knowledge) and modification. When we distribute a slow version program a potential attack is a speedup transformation. Hence guaranteed slowdown scheme is a partial case of modification protection. Therefore there is a hope to extend some ideas of slowdown scheme to protection against other modification threats.

We start with two examples of slowdown.

Example 1. Consider a function $f(x) = x^a \mod N$, where N is a production of two prime numbers as in RSA cryptosystem. We can slow down computation of f in the following way. We keep a in secret but announce $b = a + k\varphi(N)$ for some integer k instead. By Euler's theorem $x^a \equiv x^b \pmod N$. To proof that $g(x) = x^b \mod N$ is guaranteed slowdown for f we should check to things: a) g is slower than f and b) g is difficult to speed up. We have only informal evidence for this statements. Firstly, $b > a$ and, moreover, we are free to choose k as large as we want; it seems very natural that raising to the larger power requires more computation. Secondly, up to now there is no known polynomial algorithms to compute $\varphi(N)$ so we believe that given b to compute lesser degree c such that for any x the equality $x^c \equiv x^b \pmod N$ holds is also hard. However, we still need rigorous proofs here.

Example 2. Take any trap-door function f_k (see [9]). Knowing the secret key k there is a polynomial algorithm A to compute f_k^{-1}. As a slowdown version of A we can take the following algorithm A':

```
for y=1 to 2^n do
    if f_k(y)=x then return y;
```

Let us discuss the idea of guaranteed slowdown. The first question to answer is how to measure the speed of algorithm. On any particular input we can achieve

even a constant time for computation. Hence for the guaranteed slowdown definition speed should be an integral characteristic. We use average speed but other approaches might be reliable as well.

Another point of discussion is whether to study slowdown for functions defined on finite or infinite domains? Below we choose first answer (hence we can speak about circuit slowdown). But slowdown for functions on infinite domains also might be investigated. So the average speed of finite function is just a number. If we will speak in terms of circuits than we would use circuit size as a measure for speed.

We now give a slightly more formal description of our model. A pair of algorithms A_f (fast), A_s (slow) is called a *guaranteed slowdown* for function f if both algorithms compute f, A_f works on time t and given only A_s it is "difficult" to construct an algorithm computing f faster than t_1.

How can we prove difficulty of speed up? To have a complexity bound we should specify a computational problem. That is a family of questions and answers. The above description is a *single slowdown scheme* but security proof might exist only for *family* of such schemes. The way out is the following. We specify a family of functions F. Security proof should be a statement like: given A_s for some $f \in F$ it is difficult to construct a (fast) algorithm computing f.

The notion of guaranteed slowdown is very close to the idea of trap-door function. The common point is that in both cases we have pair of algorithms and knowing only the public one it is difficult to get the secret algorithm. The difference is that we do not require dramatic difference between efficiency of algorithms in the pair. Another point is that in trap-door function scheme there are three algorithms (straightforward computation, obvious slow inversion and secret fast inversion) while in guaranteed slowdown there are only two.

What are further questions about guaranteed slowdown? It might be interesting to know:

- Whether it is possible to make any level slowdown? That is whether for any function s there is a function f that could be slowdown from t to $s(t)$?
- Whether it is possible to organize multilevel slowdown? We mean here a series of algorithms $A_1, \ldots, A_k$ computing f such that A_i is slower than A_{i+1} and knowing any A_i it is difficult to get an algorithm comparable with A_{i+1}.
- How to prove that speed up is difficult?
- Function that is nonzero only on one input value is called *point function*. It seems quite natural that such functions have slowdown schemes. The

question is to make a full study of guaranteed slowdown schemes for point functions.

3. Generalized encryption schemes

In this section we investigate the following model [1]. There are two participants, Alice and Bob. Alice has some computational task, that is function $f \in F$ and input x, Bob has nothing. Alice want to get a result $f(x)$ faster than just doing all computation by herself. Alice can communicate with Bob and ask him to do some computation for her. We study only *semihonest* model here that is Bob send back correct results. The main restriction is information-theoretic security of x and f (Bob knows only that $f \in F$). In contrast to [1] below we also consider generalized encryption scheme with cryptographic security.

We present generalized encryption scheme for two functions. In these examples f is not a secret, but x is.

Example 3. [1] Discrete logarithm. Let prime number p be fixed and g be generator for Z_p^*. For every integer u such that $(u, p) = 1$ the value of discrete logarithm function $f(u)$ is the unique integer $e \in [1, p-1]$ for which $g^e \equiv u \mod p$. Generalized encryption scheme for discrete log is as follows. Alice send to Bob $u' = ug^r$ (for random r), Bob send back discrete log e' for u', and Alice get the answer $e = e' - r$. Value of ug^r is uniformly distributed on the set $[1, p-1]$, therefore Bob get no information about u. Discrete log in usual computation is believed to be a hard problem.

Example 4. Inverse operation in a group. Here we present a schema that could be applied for many functions. Let G be any group such that inversion operation as at least twice harder than multiplication. Then there is a fully encrypted computation scheme for inversion. Alice asks Bob to compute inversion of rx, receives $x^{-1}r^{-1}$ and multiply result by r. So Alice uses two multiplications instead of one inversion. Bob get no knowledge about x. Multiplication group of matrices over the finite field is a natural candidate for "inversion is twice harder than multiplication" property.

Use of auxiliary untrusted computational source arise in several practical areas. First is *market-based parallel computing*. That is we "encrypt" our hard computational tasks and (for money) ask outside computers to perform it for us. Second relevant framework is mobile agents technology. We send programs on others computers to make some task for us (usually depending on that host's inputs). Another property of mobile agents is their ability to change the host

from time to time. Third application field is development of smart cards. Informally smart card sends some computational task to an auxiliary computer and tries not to reveal any essential information about computation. In all these cases we have two security aspects: to hide internal information from outside host and to check whether it's results are correct. In our study we deal only with first one.

We now recall the definition of generalized encryption scheme. It is a two party (Alice and Bob) protocol. Alice has a computational task that is $x \in X$ and $f \in F$. There are finite number of communication rounds between Alice and Bob. In each round i Alice compute her massage a_i based on her input, all previous Bob answers and random coins and send a_i to Bob. He compute an answer based on all massages received from Alice to the moment. After the last round Alice compute an $f(x)$ based on all information she has. We have two requirements: efficiency and security. Efficiency means that total amount of computation performed by Alice should be less than that in the case of straightforward computation (without Bob). There are two different approaches two security. In the work [1] information-theoretic security is used. This means that distribution of Alice messages should be the same for all input values.

We now introduce cryptographic security for GES and discuss its relation *black-box security* [3]. If distribution of Alice massages for different pairs (x, f) are *computational indistinguishable* (see [9]), then we call GES to be cryptographically secure. Consider any family of functions F defined on a set of strings X. Let us fix some *pseudorandom function* G (again, see [9]). Then for any string r and any seed k we call

$$\tilde{f} = f(x \text{ XOR } r) + G_k(x)$$

to be a *prepared* form of f.

Proposition 1. Suppose for every prepared form of every function from F there exists an obfuscation with black-box security. Then there is a GES for F with cryptographic security.

Proof. Our construction is straightforward. We choose randomly r and k and send to Bob string $(x \text{ XOR } r)$ and obfuscated $\tilde{f}$. Bob send back $\tilde{f}(x \text{ XOR } r)$ and Alice subtract $G_k(x \text{ XOR } r)$ from it to obtain $f(x)$.

To get a security property we should combine two facts. First is that G_k is computationally indistinguishable with truly random function. Hence $\tilde{f}$ is also indistinguishable with random function. Second point is that it is computationally hard to get any knowledge besides input-output behavior about

function obfuscated with black-box security. Thus our GES is cryptographically secure. □

We know [3] that black-box secure obfuscation not possible for every function. Still obfuscation even with smaller level of security leads to a potentially applicable generalized encryption schemes.

We now recall models that are the most relevant to generalized encryption scheme. *Secure function evaluation* is a common computational task for several parties each of which knows only part of data. The security requirement is not to reveal more information about input data than just output value. *Encrypted computation* [14] is a two parties (Alice and Bob) task. Alice has a function, Bob has an input value, Alice sends her function in encrypted form to Bob, he performs computation on his value and returns an (encrypted) result back. Third problem is acceleration of raising to a power in RSA cryptosystem [4, 12], which is, in fact, a particular case of generalized encryption scheme with cryptographic security. Besides this models there is another relevant notion. As we see in examples, basic idea is to ask Bob to compute the same function but with different input. The same idea was used in *self-correctors* [5]. The difference in our works is that we ask that new input reveal no information about original one.

We conclude this section with two open problems:

- For which functions there exists a generalized encryption scheme with cryptographic security?
- For which computational tasks there is essential polynomial speed up with theoretic-informational security?

4. Function Sharing

Function sharing is a protocol for distributed computation satisfying some security requirements. There are three parties, say Alice, Bob, and Carl. Let family of functions F be fixed and known to everybody. We study the following process of computing a function $f \in F$. Alice, Bob and Carl have predefined secret functions A_f, B_f, and C_f, respectively. Input x goes to Alice and Bob, they compute their functions on it and send results $A_f(x)$ and $B_f(x)$ to Carl, Carl compute his function on these results and outputs the value of f.

Thus we have the decomposition formula

$$f(x) \equiv C_f(A_f(x), B_f(x)).$$

The security requirement is that knowing only two of functions A_f, B_f, and C_f it is impossible to get any knowledge about f except that f belongs to F.

Example 5. Let F_N be the class of all functions f that there exists an algorithm Alg_f computing it and that length of description of Alg_f is less than N. Then there is a simple function sharing scheme for F_N. We already have a good developed theory of data secret sharing. Hence our idea is to treat a function as a sort of data. So we write down a description string S of an algorithm computing f. Than we generate a random string R of the same length. We send R to left party, $S\mathrm{XOR}R$ to the right one. They wouldn't compute anything and just resend x and their part of f coding to Third party. Third party recover S end evaluate a universal circuit (universal Turing machine) on S and x.

In program obfuscation [6] the idea of usual secret sharing is used in the following obfuscating transformation: boolean variable v is splitted into two x, y such that on any point in time $v = x\mathrm{XOR}y$. In the same way any function sharing scheme will lead to a procedure splitting transformation. Notion of function sharing is applicable wherever program is decomposed into several parts. Now two such approaches are obfuscation on interpretation level [13] and dividing mobile agent into set of agentlets [7]. Notice that in all these applications function sharing will be useful mostly against static attacks.

It is a natural idea to decompose a secret into several parts such that knowing only some of them does not help to recover this secret. When the secret is a string of bits a lot of beautiful secret sharing schemes were constructed since the seminal paper [2] was published. Function sharing is an extension of that idea. There is another framework, namely minimal model for secure computation [8], close to function sharing. The common is three parties, result should be computed by third (Carl), inputs goes only to first two (Alice and Bob). But these models have a significant difference. Namely, in our case input circuit is a secret and data is not while in multiparty communication complexity circuit is public, but data is a secret.

Now we present two modifications of our main task. One additional requirement is to ask C_f be as simple as possible. Second is to restrict the size of output of A_f and B_f. More precisely an interesting questions are:

- For which function families it is possible to construct a function sharing scheme with C_f equal to XOR?
- For which function families it is possible to construct a function sharing scheme with output size of A_f and B_f at most constantly larger then output size of f?

5. Conclusion and future work

Theoretical research in the area of software protection goes in two ways: investigation of existed models and introduction of new ones. We present three frameworks: guaranteed slowdown, generalized encryption scheme and function sharing. These models allow us to connect problems of program protection with such good investigated notions as trap-door functions (for guaranteed slowdown) and random self-reducibility for (generalized encryption scheme).

A series of new theoretical problems naturally arise in our study. Probably the most interesting are to find a large class of functions having generalized encryption schemes and to construct a formal proof of irreversibility of slowdown transformation.

To define an adequate theoretical framework for software protection is not easy. The main difficulty is pure understanding of what are protected programs and what are threats. One of the unanswered question in modelling is: how to define what is known to adversary and what is a secret in a program. In cryptography there is a Kerckhoff law saying that an algorithm is public and a key is a secret. In practice of software protection real secrets have different nature.

The target of modelling is to get some theoretical result and bring it back to practice. Here we introduce several ideas with potential for applications, e.g. RSA encryption slowdown. Still for further theoretic investigation our formalizations should be more rigorous.

6. Acknowledgements

I thank Peter Holgerson for idea of program slowdown.

References

[1] Martín Abadi, Joan Feigenbaum, and Joe Kilian. On hiding information from an oracle. *J. Comput. Syst. Sci.*, 39(1):21–50, 1989.

[2] Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, 1979.

[3] Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil P. Vadhan, and Ke Yang. On the (im)possibility of obfuscating programs. In *CRYPTO '01: Proceedings of the 21st Annual International Cryptology Conference on Advances in Cryptology*, pages 1–18, London, UK, 2001. Springer-Verlag.

[4] Philippe Begiun and Jean-Jacques Quisquater. Fast server-aided rsa signatures secure against active attacks. In *CRYPTO '95: Proceedings of the*

15th Annual International Cryptology Conference on Advances in Cryptology, pages 57–69, London, UK, 1995. Springer-Verlag.

[5] Manuel Blum, Michael Luby, and Ronitt Rubinfeld. Self-testing/correcting with applications to numerical problems. *J. Comput. Syst. Sci.*, 47(3):549–595, 1993.

[6] Christian Collberg, Clark Thomborson, and Douglas Low. A taxonomy of obfuscating transformations. Technical Report 148, Dept. Computer Science, University of Auckland, July 09 1997.

[7] L. D'Anna, B. Matt, A. Reisse, T. Van Vleck, S. Schwab, and P. LeBlanc. Self-protecting mobile agents obfuscation report. Technical Report 03-015, Network Associates Labs, June 2003.

[8] Uriel Feige, Joe Kilian, and Moni Naor. A minimal model for secure computation (extended abstract). In *STOC*, pages 554–563, New York, NY, USA, 1994. ACM Press.

[9] Oded Goldreich. *Foundations of Cryptography: Basic Tools*. Cambridge University Press, New York, NY, USA, 2000.

[10] Fritz Hohl. Time limited blackbox security: Protecting mobile agents from malicious hosts. *Lecture Notes in Computer Science*, 1419:92–113, 1998.

[11] Ben Lynn, Manoj Prabhakaran, and Amit Sahai. Positive results and techniques for obfuscation. In *EUROCRYPT*, pages 20–39, 2004.

[12] Tsutomu Matsumoto, Koki Kato, and Hideki Imai. Speeding up secret computations with insecure auxiliary devices. In *CRYPTO '88: Proceedings of the 8th Annual International Cryptology Conference on Advances in Cryptology*, pages 497–506, London, UK, 1990. Springer-Verlag.

[13] Akito Monden, Antoine Monsifrot, and Clark Thomborson. A framework for obfuscated interpretation. In *CRPIT '04: Proceedings of the second workshop on Australasian information security, Data Mining and Web Intelligence, and Software Internationalisation*, pages 7–16, Darlinghurst, Australia, 2004. Australian Computer Society, Inc.

[14] Tomas Sander and Christian F. Tschudin. On software protection via function hiding. In *Information Hiding*, pages 111–123, 1998.

[15] Nikolay P. Varnovsky and Vladimir A. Zakharov. On the possibility of provably secure obfuscating programs. In *Ershov Memorial Conference*, volume 2890 of *Lecture Notes in Computer Science*, pages 91–102. Springer, 2003.

The equivalence problem for programs with mode switching is PSPACE-complete

Rimma Podlovchenko, Dmitry Rusakov and Vladimir Zakharov

Abstract. We study a formal model of imperative sequential programs. In this model programs are viewed as deterministic finite automata whose semantics is defined on Kripke structures. We focus on the equivalence problem for some specific class of programs — programs with mode switching — whose runs can be divided into two stages. In the first stage a program selects an appropriate mode of computation. Several modes may be tried (switched) in turn before making the ultimate choice. Every time when the next mode is put to a test, the program brings data to some predefined state. In the second stage of the run, once a definitive mode is fixed, the final result of computation is produced. The effect of mode switching may be used for automatic generation of opaque predicates, i.e. boolean expressions whose behavior is known a priori. Such predicates provide a very simple and effective means for virus obfuscation; therefore the development of efficient algorithms for the analysis of programs with mode switching is an urgent task in view of designing virus detection tools. We develop a new technique for simulating the behavior of such programs by means of finite automata and demonstrate that the equivalence problem for programs with mode switching is decidable within a polynomial space. By revealing a close relationship between the equivalence problem for this class of programs and the intersection emptiness problem for deterministic finite state automata we show that the former is *PSPACE*-complete.

1. Introduction

The intimate linkage among automata and formal models of programs used for the purpose of translation, verification, optimization, etc. is widely recognized. Since the early 60-th it was found out [4, 5, 9, 21, 29] that common finite state, multi-tape, multi-head, push-down automata give a suitable framework for developing decision procedures/proving undecidability for many program analysis problems originated from software engineering (see [34] for a survey). In this paper we study one of such problems, namely, the equivalence problem, for some specific class of imperative programs with mode switching. The runs of such programs are divided into two stages. In the beginning of a run an

```
program π1
read (y1,y2);
x=y1; z=y2;
u=x+z;
stop.
```

```
program π2
read (y1,y2);
x=y1; z=y2;
if P(x,z) then {x=z; goto L1}
          else {z=z+x; goto L2};
L1:  z=y1; x=y2;
if P(z,x) then {u=x+z; z=x;
x=u-z}
          else {u=x-z; z=u+x};
stop;
L2:  z=y2; x=y1;
if P(x,z) then u=2*x-z**2;
          else u=z+z;
stop.
```

Figure 1: Programs with mode switching

appropriate mode of computation is selected. A program may try (switch) a number of modes in turn before making the final choice. In the second stage, once some mode of computation is chosen, a program starts an ordinary run and yields the final result.

Usually mode switching is achieved by means of constant assignment statements. Two programs π_1 and π_2 in Fig 1. illustrate the concept of mode switching. It is easy to see that both programs are equivalent, i.e. they compute the same function $u = y_1 + y_2$. Boxed statements in these programs play the role of mode switches; when executing these statements the programs bring data to the predefined states that are specific for each mode. Our interest to the equivalence problem for programs with mode switching has both theoretical and practical motivations. In [23, 24] a theory of algebraic models of programs was 1 program introduced for the purpose of designing effective equivalence-checking techniques and complete systems of equivalent transformations of programs. In this theory programs are viewed as deterministic finite automata operating on semigroups or Kripke structures. A series of obtained results [26, 32] show that for many algebraic models of programs it is possible to design efficient (polynomial-time) equivalence-checking procedures. It was found out (see [14, 15, 16, 18]) that decidability and complexity of the equivalence problem for algebraic models of programs depend greatly on some group-theoretic properties of program semantics. That is why it is very important to know how much this or that property of a semigroups or Kripke structures used for the semantics in algebraic models of programs influences the decidability and complexity of the equivalence problem.

In the framework of the theory of algebraic models of programs the semantics of programs with mode switching can be specified in terms of semigroups with right zeroes. In [13, 14] the equivalence problem for finite automata operating on free semigroups with right zeros is considered. A decidability result was obtained by establishing the regularity preserving properties of the set-theoretic and closure operations on topological spaces of functions computed by such automata. In [20, 19] a "hard set" method was successfully applied to the equivalence problem for linear recursive programs with constants. It must be emphasized that in the model of recursive programs constants play the same role as mode switching statements in the propositional model of sequential programs we deal with in this paper. Both equivalence-checking techniques developed in [13, 14, 19] are very much sophisticated; their main deficiency is that they give no means to estimate the complexity of the problem. One of the aims of our paper to is to estimate precisely (as much as possible) the complexity of the equivalence problem for programs with mode switching.

Another topic which involves programs with mode switching is malicious pattern (viruses) detection in software. The classic virus-detection techniques look for the presence of a virus specific sequence of instructions (*virus signature*) inside the programs: if the signature is found, it is highly probably that the program is infected. A new generation of *metamorphic viruses* attempts to evade simple pattern-matching detection by using complex obfuscations: when replicating these viruses change their signatures by applying semantic-preserving transformations (see [1]). The only way to disclose such viruses is to develop efficient equivalence-checking techniques that could cope with the common obfuscating transformations. Thus, in [2] an architecture for detecting malicious patterns in executables is presented that is resilient to some obfuscating transformations (dead-code insertion and code transposition). But the properly resistant obfuscations [3] rely on the existence of opaque predicates whose behavior is known a priori to the obfuscator, but which is difficult for the deobfuscator to deduce. The most simple way to get the opaque predicates is to make the same predicate computable on the same data but in different program points. This may be achieved by bringing data into some fixed state before computing such predicate several times along a run. But this is just the effect of mode switching. When considering the program π_2 depicted in the Fig. 1 one could find that both underlined conditions in the branching statements are opaque predicates obtained with the help of mode switching: one of them is always evaluated to true whereas the other to false. Thus, to detect a metamorphic virus (the program π_2) by its signature (the program π_1) one need an effective equivalence-checking procedure which could cope with mode switching.

In our paper we reveal close relationships between the equivalence problem for programs with mode switching and the Intersection Emptiness Problem (IEP) for deterministic finite state automata (DFAs). The Intersection Emptiness Problem for DFA is that of checking, given a collection of k DFAs $F_1, \ldots, F_k$ of size $|F_i| = n$, if their intersection is empty: $\bigcap_{i=1}^{k} L(A_i) \stackrel{?}{\neq} \emptyset$, where $L(F)$ denotes the language accepted by the automaton F. If the parameter k is a constant then the problem has a polynomial time algorithm, but the general problem, where this parameter can depend on the input size (say, $k = n$), is much harder, known to be *PSPACE*-complete [12].

Some recent papers testify that the IEP would have a substantial impact on many aspects of complexity theory and formal verification of complex systems. In [10] it was proved that integer factoring of an n-bit number is solvable in time $n^{O(1)} \cdot 2^{\varepsilon n}$ for any $\varepsilon > 0$, provided that one can decide the IEP for a family $F_1, \ldots, F_k$ of DFAs F_i of size n in time $t(n,k) = n^{\frac{k}{f(k)}+d}$, where $f(\cdot)$ is an unbound function, and $d > 0$ is a constant. Moreover, by assuming that there is a non-uniform circuit that will solve the IEP with size $t(n,k)$, one could deduce $NLOG \neq NP$. In [28] the IEP was used to demonstrate that a great many verification problems of supervisory controllers for discrete-event systems are *PSPACE*-complete. We think that the results of our paper also give a new insight into the IEP.

The paper is organized as follows. In Section 2 we define formally the syntax and the semantics of propositional sequential programs. In Section 3 the model $\mathcal{M}_0$ which captures the semantics of programs with mode switching is introduced. We also reduce the equivalence problem $\pi' \sim_{\mathcal{M}_0} \pi''$ for programs with mode switching to that of checking three characteristic properties of the runs of π', π''. In Section 4 we show that these properties can be verified by constructing a finite number of DFAs and checking their emptiness. The DFAs we use for this purpose are similar to Vectorized Finite State Automata introduced in [11] for natural language processing. A vectorized DFA operates on a tape divided into N tracks and its internal states are vectors $s = \langle v_1, \ldots, v_N \rangle$. Some tracks may be synchronized; it is required that the input letters on the synchronized tracks should be unified (in our case this means that the letters on these tracks should be the same). The synchronization of tracks varies along a run of DFA. Vectorized DFAs have the same computation power as common DFAs and the emptiness problem for them is *NLOG*-complete. Since the state space of DFAs we use is exponential of the size of programs to be analyzed, we arrive at the conclusion that the equivalence problem $\pi' \sim_{\mathcal{M}_0} \pi''$ is in *PSPACE*. In Section 5 we reduce the IEP to the equivalence problem for programs mode switching and establish thus the *PSPACE*-completeness of the latter. We conclude with discussing some new research problems caused by the results obtained.

2. Preliminaries

In this section we define the syntax and the semantics of propositional sequential programs.

Fix two finite alphabets $\mathcal{A} = \{a_1, \ldots, a_r\}$, $\mathcal{P} = \{p_1, \ldots, p_k\}$. The elements of $\mathcal{A}$ are called *basic statements*; they stand for assignment statements in imperative programs. The elements of $\mathcal{P}$ are called *basic predicates*; they stand for elementary built-in relations on program data. Each basic predicate may be evaluated by 0 (false) or 1 (true). A tuple $\langle \delta_1, \ldots, \delta_k \rangle$ of truth-values of basic predicates is called a *condition*. The set of all conditions is denoted by $\mathcal{C}$; we write $\Delta_1, \Delta_2, \ldots$ for generic elements from $\mathcal{C}$.

Definition 1. *A deterministic* propositional sequential program *(*PSP *for short) is a finite transition system* $\pi = \langle V, \mathbf{entry}, \mathbf{exit}, T, B \rangle$, *where*

- V *is a non-empty set of* program points*;*
- **entry** *is the* initial point *of the program;*
- **exit** *is the* terminal point *of the program;*
- $T\colon (V - \{\mathbf{exit}\}) \times \mathcal{C} \to V$ *is a (total)* transition function*;*
- $B\colon (V - \{\mathbf{exit}\}) \to \mathcal{A}$ *is a (total)* binding function.

A transition function represents the control flow of a program, whereas a binding function associates with each point some basic statement. By the *size* $|\pi|$ of a program π we mean the cardinality of the set V. Any finite sequence of points $v_1, v_2, \ldots, v_n$ such that for every i, $1 \leq i < n$, $v_{i+1} = T(v_i, \Delta_i)$ holds for some condition Δ_i, is called a *control path* (or a *trace*) in the PSP π from v_1 to v_n. We say that v_n is *reachable* from v_1 if there exists a trace from v_1 to v_n.

The semantics of PSPs is defined with the help of Kripke structures used in the framework of dynamic logics.

Definition 2. *A* Kripke structure *is a quadruple* $M = \langle S, s_0, R, \xi \rangle$, *where*

- S *is a non-empty set of* data states*;*
- $s_0 \in S$ *is a distinguished* initial state*;*
- $R : \mathcal{A} \times S \to S$ *is a (total)* updating function*;*
- $\xi : S \to \mathcal{C}$ *is a (total)* evaluation function.

An updating function R gives the interpretation of basic statements: a data state $R(a, s)$ is the result of application of a basic statement a to a data state s. An evaluation function ξ is used for the interpretation of basic predicates: $\xi(s)$ gives a tuple of truth-values for all basic predicates on a data state s. By a *data path* in M we mean any sequence of states $s_1, s_2, \ldots, s_k$ such that $s_{i+1} = R(a_i, s_i)$ for some basic statement a_i, $1 \leq i < k$.

Let $\pi = \langle V, \mathbf{entry}, \mathbf{exit}, T, B \rangle$ be a PSP and $M = \langle S, s_0, R, \xi \rangle$ be a Kripke structure. A *run* of π on M is a sequence (finite or infinite) of pairs

$$r(\pi, M) = (v_1, s_1), (v_2, s_2), \ldots, (v_i, s_i), (v_{i+1}, s_{i+1}), \ldots \tag{1}$$

such that

1. s_0 is the initial state of M, and $v_1 = \mathbf{entry}$;
2. $s_i = R(B(v_i), s_{i-1})$ and $v_{i+1} = T(v_i, \xi(s_i))$ hold for every i, $i \geq 1$;
3. the sequence $r(\pi, M)$ either is infinite (in this case we say that the run *loops* and yields no results), or ends with a pair (v_n, s_n, Δ_n) such that $v_{n+1} = \mathbf{exit}$ (in this case we say that the run *terminates* and gives a result s_n).

We write $\downarrow r(\pi, M)$ to indicate that the run terminates and denote by $[r(\pi, M)]$ its result assuming that the result is undefined when $r(\pi, M)$ loops. It is worth noting that if $v_i = v_j$ and $s_i = s_j$ for some pair of triples in (1) then $r(\pi, M)$ loops. If a point v_i occurs in some triple of (1) then we say that $r(\pi, M)$ *passes via* v_i.

In what follows when referring to a *model of programs* $\mathcal{M}$ we mean the set of all PSPs over fixed alphabets $\mathcal{A}, \mathcal{P}$ whose semantics is specified by the set $\mathcal{M}$ of Kripke structures.

Definition 3. *Given a model of programs $\mathcal{M}$, we say that PSPs π_1 and π_2 are* equivalent *($\pi_1 \sim_{\mathcal{M}} \pi_2$ in symbols) iff $[r(\pi_1, M)] = [r(\pi_2, M)]$ for every $M \in \mathcal{M}$.*

The equivalence problem for a model of programs $\mathcal{M}$ is to check, given a pair of PSPs π_1 and π_2, whether $\pi_1 \sim_{\mathcal{M}} \pi_2$ holds. The complexity of the equivalence problem "$\pi_1 \sim_{\mathcal{M}} \pi_2$?" depends on the set of structures M which specifies a model of programs. Two examples below illustrate this thesis.

Example 1. *Given a set calA of basic statements, consider a free semigroup* $\langle \mathcal{A}, \circ \rangle$ *generated by* $\mathcal{A}$. *The elements of this semigroup may be thought of as finite sequences (words) of basic statements, whereas binary operation* $\circ$ *may be interpreted as concatenation. The empty sequence lambda stands for the neutral element of the semigroup. Then the equivalence problem for the model of programs* $M' = \{\langle calA^*, \lambda, \circ, \xi \} \ : \ \xi$ *is an evaluation function on* $\mathcal{A}$ *is decidable in time* $O(n \log n)$. *In [9, 29]it was demonstrated that the equivalence problem for* M' *is reducible in linear time to the the equivalence problem for deterministic finite automata; the complexity* $O(nlogn)$ *of the latter was established in [8].*

Example 2. *Given a set* $\mathcal{A} = \{a_1, \ldots, a_n, a_1^{-1}, \ldots, a_n^{-1}\}$ *of basic statements consider Abelean group* $\langle S, \circ \rangle$ *of rank* n *generated by the elements from* $\mathcal{A}$. *The equivalence problem for the model* $M' = \{\langle S, e, \circ, \xi \} \ : \ \xi$ *is an evaluation function on* S, *where* e *is the unit (neutral) element of* S, *was studied in [16]. It was shown that this problem is decidable within exponential space when* $n = 1$, *and it is undecidable when* $n \geq 2$.

Other results on the equivalence problem for some models of programs may be found in [16, 17, 18, 25, 26, 27, 34, 35, 36].

3. Programs with mode switching

In this paper we focus on the equivalence problem for some specific class of programs whose runs can be divided into two stages. In the first stage a program selects an appropriate mode of computation. Several modes may be tried in turn before making the ultimate choice. Every time when the next mode is put to the test, the program brings the data back to the initial state. In the second stage, once a definitive mode is fixed, the final result of computation is generated. In real programs mode switching may be implemented by restart statements or constant assignment statement. In this section we introduce formally the model of such programs in the framework of PSP's syntax and semantics.

We will assume that the set of basic statements $\mathcal{A}$ is partitioned into two subsets $\mathcal{A}_{ord} = \{a^1, \ldots, a^k\}$ (*ordinary actions*) and $\mathcal{A}_{mode} = \{b^1, \ldots, b^N\}$ (*mode switches*). Those points v in a PSP π that are associated with mode switches (i.e. $B(v) \in \mathcal{A}_{mode}$) are called *switching points*. All other points are called *ordinary points*. We write V_{mode} to denote the set of all switching points of a given program π. Without loss of generality, we will assume that **entry** $\in V_{mode}$.

Two principles are used as the basis for the semantics of PSPs with mode switching:

- each ordinary action a is interpreted according to a current mode of computation, and
- each mode switch b abandons any previous intermediate result of computation and brings data into some distinguished state s_b.

Thus, the model of programs with mode switching is characterized by the set of all Kripke structures $\mathcal{M}_0 = \{M_\xi \ : \ M_\xi = \langle S, s_0, R, \xi \rangle\}$ such that

1. $S = \{\varepsilon\} \cup \mathcal{A}_{mode}\mathcal{A}^*_{ord}$, i.e. S includes the empty string λ and all strings $ba_1a_2 \ldots a_n$, where a mode switch $b \in \mathcal{A}_{mode}$ is followed by a string of ordinary actions $a_1a_2 \ldots a_n \in \mathcal{A}^*_{ord}$;
2. $s_0 = \lambda$;
3. an updating function R is defined as follows:

$$R(y, s) = \begin{cases} sy, \text{ if } y \in \mathcal{A}_{ord}, \\ y, \quad \text{if } y \in \mathcal{A}_{mode}. \end{cases}$$

Since the data space S and the interpretation of basic statements R are fixed, each structure $M_\xi \in \mathcal{M}_0$ is completely specified by its evaluation function ξ. PSPs with mode switching π_1 and π_2 are called equivalent ($\pi_1 \sim_{\mathcal{M}_0} \pi_2$ in symbols) iff $[r(\pi_1, M_\xi)] = [r(\pi_2, M_\xi)]$ holds for every structure $M_\xi \in \mathcal{M}_0$. When studying decidability and complexity of the equivalence problem for $\mathcal{M}_0$ we will use the inverse variant of this definition: PSPs π_1 and π_2 are not equivalent iff there exists a structure $M_\xi \in \mathcal{M}$ such that either both runs $r(\pi_1, M_\xi)$ and $r(\pi_2, M_\xi)$ terminate but $[r(\pi_1, M_\xi)] \neq [r(\pi_2, M_\xi)]$, or one of the runs (say, $r(\pi_1, M_\xi)$) terminates, whereas the other (in our case, $r(\pi_2, M_\xi)$) loops.

Given a PSP π, we introduce the *skeleton* G_π of π as a finite directed graph intended for representing the reachability relation between the switching points in π. Formally, $G_\pi = (U, E)$, where $U = V_{mode} \cup \{\mathbf{exit}\}$ is the set of vertices and $E = \{\langle v', v'' \rangle \ : \ v', v'' \in U$, and v'' is reachable from v' in π by a trace which does not pass via any switching point other than v' and $v''\}$ is the set of arcs. A *trajectory* is any directed path in G_π (finite or infinite) which begins in the **entry** point. A trajectory reflects a possible scenario of mode selection in the course of some run of π. We say that a run $r(\pi, M_\xi)$ of a PSP π on a structure M_ξ *traverses the skeleton* G_π *along a trajectory* $v_1, v_2, \ldots, v_n, \ldots$ if $r(\pi, M_\xi)$ passes via switching points $v_1, v_2, \ldots, v_n, \ldots$ in order. The proposition below follows from the definition of $\mathcal{M}_0$ and states the principal property of PSP's runs on the structures under consideration.

Proposition 1. *If a run $r(\pi, M_\xi)$ of a PSP π traverses the skeleton G_π along a trajectory $v_1, v_2, \ldots, v_i, \ldots, v_j, \ldots$ such that $v_i = v_j$ then $r(\pi, M_\xi)$ loops.*

A trajectory $v_1, v_2, \ldots, v_n$ in a skeleton G_π is called

- *repetition-free* if it does not pass twice via the same vertex;
- *complete* if it is repetition-free and ends in the node $v_n =$ **exit**.

Thus, a terminating run of π traverses the skeleton G_π only along a complete trajectory, while a looping run of π may traverse G_π along either some repetition-free non-complete trajectory (in this case we say that the run loops on ordinary points of π), or some infinite trajectory (in this case we say that the run loops on switching points of π). By Proposition 1, in order the latter case to happen a run of π should traverse the skeleton along a trajectory $v_1, v_2, \ldots, v_i, v_{i+1}, \ldots$ such that $v_1, v_2, \ldots, v_i$ is a repetition-free trajectory and $v_{i+1} \in \{v_1, v_2, \ldots, v_i\}$.

The proposition below is but a restatement of the equivalence problem $\pi' \sim_{\mathcal{M}_0} \pi''$ in terms of some properties of trajectories in the skeletons of the PSPs.

Proposition 2. *PSPs π' and π'' are <u>not</u> equivalent on $\mathcal{M}_0$ iff there exists a pair of repetition-free trajectories $w' = v'_1, v'_2, \ldots, v'_n$ and $w'' = v''_1, v''_2, \ldots, v''_m$ in the skeletons $G_{\pi'}$ and $G_{\pi''}$ such that for some structure M_ξ the runs $r(\pi', M_\xi)$ and $r(\pi'', M_\xi)$ traverse the skeletons along the trajectories w' and w'' respectively, and meet one of the following requirements:*

R1: both trajectories are complete and $[r(\pi', M_\xi)] \neq r[(\pi'', M_\xi)]$;

R2: one of the trajectories (say, w') is complete, whereas the other (w'') is traversed by the run which loops on ordinary points;

R3: one of the trajectories (say, w') is complete, whereas the other (w'') is traversed by the run which loops on switching points.

This proposition provides a foundation for the following equivalence-checking strategy: given a pair of PSPs, guess a complete trajectory in the skeleton of one PSP and a repetition-free trajectory in the skeleton of the other; then check if there exists some structure M_ξ to satisfy one of the requirements R1–R3 above. Since the number of repetition-free trajectories in the skeletons is finite, the equivalence problem for PSPs is reduced thus to the analysis of trajectories in their skeletons. Next we will show that the latter can be carried out with the help of DFAs.

4. Using DFAs for the equivalence-checking of programs with mode switching

The problem we deal with in this section is as follows: given a pair of repetition-free trajectories w' and w'' in the skeletons of PSPs π' and π'', check if there exists a structure M_ξ which complies with at least one of the requirements R1–R3 in Proposition 2. We demonstrate that to retrieve an appropriate structure M_ξ one could construct the specific DFAs D_1, D_2 and D_3 and check their emptiness: the requirement Ri, i= 1, 2, 3, can be satisfied by some M_ξ iff $L(D_i) \neq \emptyset$. First we discuss the key ideas of our construction of the DFAs D_i and briefly describe how they operate. Then we present a detailed description of D_1 and explain what minor modifications should be made to convert D_1 into D_2 and D_3.

When guessing a structure M_ξ which makes it possible to traverse the skeletons of π' and π'' along the trajectories w' and w'' one may rely on the basic property of the semantics of PSPs with mode switchings: each mode switch b abandons the achieved data state s and brings data into the predefined state s_b. Thus, the traversing of each arc $\langle u, v\rangle$ begins with some fixed data state which depends only on a mode switch assigned to the point u. This enables us to try all arcs of the trajectories independently in attempt to find for each arc $\langle u, v\rangle$ a specification $Spec_{uv}$ which provides the reachability of the switching point v from the switching point u in the course of some run. The specification $Spec_{uv}$ imposes constraints on a an evaluation function ξ on some data path b, ba_1, ba_1a_2, ..., where $b = B(u)$ and $a_1, a_2, \cdots \in \mathcal{M}_{ord}$. We seek to define the specification so that for any structure M_ξ which satisfies $Spec_{uv}$ the run $r(\pi', M_\xi)$ (or $r(\pi'', M_\xi)$ depending on the trajectory $\langle u, v\rangle$ belongs to) when starting from the point u reaches the point v. As soon as the specifications $Spec_{uv}$ are developed for all arcs of the trajectories, we may compose the objective structure M_ξ.

When building up $Spec_{u_1v_1}$ and $Spec_{u_2v_2}$ for a pair of arcs $\langle u_1, v_1\rangle$, $\langle u_2, v_2\rangle$ it should be seen that the specifications are *consistent*, i.e. impose the same constraints on ξ on the same data states. Two cases are possible depending on the mode switches assigned to u_1 and u_2.

1. The switching points u_1 and u_2 are associated with distinct mode switches, i.e. $B(u_1) = b_1 \neq b_2 = B(u_2)$. This implies that the data paths the specifications $Spec_{u_1v_1}$ and $Spec_{u_2v_2}$ refer to are disjoint. Therefore these specifications are always consistent and may be built up independently.

2. The switching points u_1 and u_2 are associated with the same mode switch, i.e. $B(u_1) = B(u_2) = b$. Then the specifications $Spec_{u_1v_1}$ and $Spec_{u_2v_2}$ may refer to the same data path. Therefore, special care must be taken to coordinate (synchronize) the development of such specification. But as soon as the data paths these specifications deal with diverge, they will never refer to any other common data state and the synchronization may be ceased.

Thus, all specifications $Spec_{uv}$ can be built up in parallel provided that some appropriate synchronization is used to ensure their consistency. This parallel synchronized derivation of the specifications can be implemented by some DFA D. The internal states of D keep only track of the following information:

- A tuple $[v_1, \ldots, v_N]$ of points in π' and π''. Every element v_i is a point in π' or π'' which is currently achieved in attempt to route a trace from the switching point u to the switching point v for some arc $\langle u, v\rangle$ in the trajectories w' and w''.
- Synchronization table H. It is used to provide the consistency of those specifications that refer to the same data states. The synchronization table may be viewed as a finite set of pairs (e_1, e_2) of arcs. Every such pair when being set into the table H indicates that the corresponding specifications $Spec_1$ and $Spec_2$ need coordination to maintain their consistency.

On each computation step D reads as input (guesses) a tuple of conditions $[\Delta_1, \ldots, \Delta_N]$. These conditions give rise to new constraints that should be added to the specifications: each Δ_i is viewed as a possible value of ξ on the currently achieved data state. The synchronization table H is used to check the identity of those constraints that should be added to coordinated specifications. Then DFA D computes the updated tuple of points $[T(v_1, \Delta_1), \ldots, T(v_N, \Delta_N)]$ and modifies the synchronization table H by using the transition functions T and the the binding functions B of both PSPs π' and π''. This results in transition of D to the next internal state. To complete the computation step D checks whether the new internal state satisfies some objective condition; if this is the case then D accepts. The acceptance implies the existence of a structure M_ξ such that w', w'' and M_ξ comply with one of the requirements R1–R3 (depending on the objective condition to be checked).

Now we consider a DFA D_1 intended for checking the satisfiability of the requirement R1 and describe this DFA in more detail.

Let $\pi' = \langle V, \mathbf{entry}', \mathbf{exit}, T, B\rangle$ and $\pi' = \langle U, \mathbf{entry}'', \mathbf{exit}, T, B\rangle$. To simplify the notation we will assume that $V \cap U = \{\mathbf{exit}\}$, and both PSPs have the same transition function T and the same binding function B defined on $V \cup U$.

Let $w' = v_1, v_2, \ldots, v_n, \mathbf{exit}$ and $w'' = v_{n+1}, v_{n+2}, \ldots, v_{n+m}, \mathbf{exit}$ be complete trajectories in the skeletons of π' and π'' respectively.

Let $H(n,m)$ denotes the set of all (unordered) pairs (i,j), $1 \le i, j \le n+m$, $i \ne j$, and $H_0(n,m) = \{(i,j) : \ (i,j) \in H(n,m), B(v_i) = B(v_j)\}$. The set $H_0(n,m)$ indicates all those pairs of switching points in w' and w'' that are associated with the same mode switch.

Then $D_1 = \langle \Sigma, Q, q_0, \mathbf{accept}, \delta\rangle$, where

- $\Sigma = \mathcal{C}^{n+m}$ is the input alphabet;
- $Q = \{\mathbf{accept}, \mathbf{reject}\} \cup (V^n \times U^m \times 2^{H(n,m)})$ is the set of internal states of D_1;
- $q_0 = \langle v_1, \ldots, v_n, v_{n+1}, \ldots, v_{n+m}, H_0(n,m)\rangle$ is the starting state of D_1;
- **accept** is the accepting state of D_1;
- $\delta : Q \times \Sigma \to Q$ is the (partial) state transition function.

Suppose $q = \langle u_1, \ldots, u_{n+m}, H\rangle$ is any internal state of D_1 and $z = \langle \Delta_1, \ldots, \Delta_{n+m}\rangle$ is any tuple of conditions. The state transition function δ is defined according to the following rules (we provide each rule with a brief explanation of its intended meaning).

1. If $\Delta_i \ne \Delta_j$ for some pair $(i,j) \in H$ then $\delta(q,z) = \mathbf{reject}$ (*Comm*: the tuple z does not comply with synchronization request; the constraints imposed on ξ are inconsistent);

2. Otherwise, consider the tuple of points $\langle u'_1, \ldots, u'_{n+m}\rangle$ such that

   $$u'_i = \begin{cases} T(u_i, \Delta_i), & \text{if } u_i \text{ is an ordinary point, or } q = q_0, \\ u_i, & \text{if } u_i \text{ is a switching point and } q \ne q_0, \end{cases}$$ for every $1 \le i \le n+m$.

   Two cases are possible.

   Case 1: all points u'_i, $1 \le i < n$, and u'_j, $n+1 \le j < n+m$, are switching points, and at least one of the points u'_n and u'_{n+m} is either a switching point, or **exit**. Then

(a) if u'_n or u'_{n+m} is a switching point, or if there exists i, $1 \le i < n+m$, $i \ne n$, such that u'_i is a switching point, but $u'_i \ne v_{i+1}$ then $\delta(q,z) =$ **reject** (*Comm*: this means that the automaton D_1 "went of" the trajectory when "laying off" a trace either from v_{n-1} and v_{n+m-1} to **exit**, or from v_i to v_{i+1});

(b) if $u'_i = v_{i+1}$ holds for all $1 \le i < n+m$, $i \ne n$, and $u'_n = u'_{n+m} =$ **exit**, and $(n, n+m) \in H$ then $\delta(q,z) =$ **reject** (*Comm*: this means that D_1 built a specification of a structure M_ξ such that $[r(\pi', M_\xi)] = [r(\pi'', M_\xi)]$);

(c) otherwise $\delta(q,z) =$ **accept** (*Comm*: this means that D_1 built a specification of a structure M_ξ such that both runs $r(\pi', M_\xi)$ and $r(\pi'', M_\xi)]$ traverse the skeletons along the trajectories w' and w'', but $[r(\pi', M_\xi)] \ne [r(\pi'', M_\xi)]$).

Case 2: at least one point u'_i, $1 \le i < n+m$, $i \ne n$, is an ordinary point, or both u'_n and u'_{n+m} are ordinary points. Then $\delta(q,z) = \langle u'_1, \ldots, u'_{n+m}, H' \rangle$, where

$$H' = H - \{(i,j) : \text{ both } u'_i, u'_j \text{ are ordinary points, and } B(u'_i) \ne B(u'_j)\}$$

is a new synchronization table. (*Comm*: this means that the traversing along some arcs in the trajectory is not completed yet; $B(u'_i) \ne B(u'_j)$ implies that data paths in the specifications $Spec_{v_i v_{i+1}}$ and $Spec_{v_j v_{j+1}}$ diverge and the synchronization for these specifications is of no further consequence). □

The DFA D_1 thus defined may be thought of as a one-way finite state machine operating on a tape divided into $n+m$ tracks. On the i-track it simulates some fragment r_i of the runs of PSPs π' and π''. Such fragment $r_i = (v_i^1, s_i^1, \Delta_i^1), \ldots, (v_i^{k+1}, s_i^{k+1}, \Delta_i^{k+1})$ begins with a triple whose point v_i^1 is either a switching point or **entry**, and ends with a triple whose point v_i^{k+1} is either a switching point or **exit**. The synchronization tables of D_1 ensure that only those pairs of runs of π' and π'' are simulated that could be performed on the same structure. The accepting conditions allow D_1 to accomplish successfully a simulation of the runs iff the last fragments of these runs yield different results. Thus we arrive at

Proposition 3. *Let w' and w'' be complete trajectories in the skeletons of π' and π'' respectively. Let DFA D_1 be as specified above. Then $L(D_1) \ne \emptyset$ iff there exists a structure M_ξ such that the runs $r(\pi', \xi)$ and $r(\pi'', M_\xi)$ traverse the skeletons along the trajectories w' and w'', and $[r(\pi', M_\xi)] \ne [r(\pi'', M_\xi)]$.*

It is suffice to introduce only some changes in the acceptance and rejection rules (Case 1) to transform a DFA D_1 intended for checking the requirement R1 into the DFAs D_2 and D_3 that would be responsible for the requirements R2 and R3.

To construct D_2 one has to set off in π'' strongly connected components that are free from switching points. Let

$$\begin{aligned} V_{inf} \quad = \quad & \{v \ : \ \text{there is a cycle in } \pi'' \\ & \text{which contains no switching points and passes via } v\} \end{aligned}$$

Then D_2 operates as follows: after constructing the tuple of points $\langle u'_1, \ldots, u'_{n+m} \rangle$ (see item 2 in the description of D_1), it

- rejects (rule 2(b)) if $u'_i = v_{i+1}$ holds for all $1 \leq i < n+m,\ i \neq n$, and $u'_n = \mathbf{exit}$, and u'_{n+m} is either a switching point or **exit**,
- accepts (rule 2(c)) $u'_i = v_{i+1}$ holds for all $1 \leq i < n+m,\ i \neq n$, and $u'_n = \mathbf{exit}$, and $u'_{n+m} \in V_{inf}$.

It is worthy of notice that when at some computation step D_2 accepts after constructing the tuple $\langle v_2, \ldots, v_n, \mathbf{exit}, v_{n+2}, \ldots, v_{n+m}, u \rangle$ such that $u \in V_{inf}$, this should be interpreted as π'' is presented with an unbound possibility to continue an infinite (looping) run along some cycle on ordinary points.

Similarly to D_2, after constructing the tuple of points $\langle u'_1, \ldots, u'_{n+m} \rangle$ D_3 rejects (rule 2(b)) if u'_{n+m} is either **exit**, or a switching point other than $v_{n+1}, \ldots, v_{n+m}$, and accepts (rule 2(c)) if $u'_n = \mathbf{exit}$ and $u'_{n+m} \in \{v_{n+1}, \ldots, v_{n+m}\}$.

Proposition 4. *Let w' be a complete trajectory and w'' be a repetition-free trajectory in the skeletons of π' and π'' respectively. Let DFA D_2 and D_3 be as specified above. Then*

1. *$L(D_2) \neq \emptyset$ iff there exists a structure M_ξ such that the runs $r(\pi', M_\xi)$ and $r(\pi'', M_\xi)$ traverse the skeletons along the trajectories w' and w'', and $r(\pi', M_\xi)$ terminates, whereas $r(\pi'', M_\xi)$ loops on ordinary points of π''.*
2. *$L(D_3) \neq \emptyset$ iff there exists a structure M_ξ such that the runs $r(\pi', M_\xi)$ and $r(\pi'', M_\xi)$ traverse the skeletons along the trajectories w' and w'', and $r(\pi', M_\xi)$ terminates, whereas $r(\pi'', M_\xi)$ loops on switching points of π''.*

Theorem 1. *The equivalence problem $\pi' \sim_{\mathcal{M}_0} \pi''$ for PSPs with mode switching is decidable in polynomial space.*

Proof. Due to Savitch's theorem [30] we can convert any nondeterministic polynomial space algorithm into deterministic one. Therefore it will suffice to design a nondeterministic polynomial space procedure for checking non-equivalence of PSPs with mode switching. It is as follows. Given a pair of PSPs π' and π'' it builds their skeletons $G_{\pi'}$, $G_{\pi''}$ and guesses a complete trajectory w' in one of the skeletons and a repetition-free trajectory in the other. Then it checks the emptiness of DFAs D_1, D_2 and D_3 corresponding to π', π'' and the selected trajectories. Propositions 2–4 guarantee that $\pi' \not\sim_{\mathcal{M}_0} \pi''$ iff for some pair of trajectories one of the DFAs D_i, $i = 1, 2, 3$, is non-empty. As may be seen from the descriptions of DFAs D_i, these automata has $O(2^{poly(n)})$ states, where $n = |\pi'| + |\pi''|$. Hence, their non-emptiness may be certified within a polynomial space. □

5. Complexity issues

The complexity of decision procedure above can not be improved to a large extent.

Theorem 2. *The equivalence problem for PSPs with mode switching is PSPACE-complete.*

Proof. We demonstrate how to build in linear time, given a family of n DFAs F_i, $1 \le i \le n$, of size $|F_i| = n$, a PSP π of size $|\pi| = n^2$ such that the intersection $\bigcap_{i=1}^{n} L(F_i) = \emptyset$ iff π has no terminating runs, i.e. π_0 is equivalent to the empty PSP.

Without loss of generality, we may assume that each DFA F_i operates on binary input alphabet $\{0,1\}$, i.e. $F_i = \langle Q_i, \delta_i, q_i^0, Q_i' \rangle$, where Q_i is a finite state of states, $\delta_i : Q \times \{0,1\} \to Q$ is a (total) transition function, $q_i^0 \in Q_i$ is the starting state, and $Q_i' \subseteq Q_i$ is the set of final states. Let $\mathcal{A}_{mode} = \{b\}$, $\mathcal{A}_{ord} = \{a\}$, and $\mathcal{P} = \{p_1, p_2\}$. For every DFA F_i we build the PSP $\pi_i = \langle Q_i \cup \{\mathbf{dead}, \mathbf{entry}_i, \mathbf{exit}_i\}, \mathbf{entry}_i, \mathbf{exit}_i, T_i, B_i \rangle$ whose transition function T_i is defined as follows for every $q \in Q$ and $x \in \{0,1\}$:

1. $T_i(\mathbf{entry}_i, \langle x, 0 \rangle) = q_i^0$, and $T_i(\mathbf{entry}, \langle x, 1 \rangle) = \begin{cases} \mathbf{exit}_i, \text{ if } q_i^0 \in Q_i', \\ \mathbf{dead}, \text{ otherwise;} \end{cases}$
2. $T_i(q, \langle x, 0 \rangle) = \delta_i(q, x)$, and $T_i(q, \langle x, 1 \rangle) = \begin{cases} \mathbf{exit}_i, \text{ if } \delta_i(q, x) \in Q_i', \\ \mathbf{dead}, \text{ otherwise;} \end{cases}$
3. $T_i(\mathbf{dead}, \Delta) = \mathbf{dead}$.

It is easy to see from the construction of π_i that a binary string $w = x_1 x_2 \dots x_k$, $k \geq 0$, is accepted by F_i iff $r(\pi_i, M_{\xi_w})$ terminates on a structure M_{ξ_w} such that $\xi_w(b(a)^k) = \langle x_k, 1\rangle$ and $\xi_w(b(a)^i) = \langle x_i, 0\rangle$ for all $1 \leq i < k$.

The PSP π_0 is obtained from the family of PSPs $\pi_1, \dots, \pi_n$ by identifying each point $\mathbf{exit}_i$, $1 \leq i \leq n-1$, with the point $\mathbf{entry}_{i+1}$. It is easy to verify that a binary string w is accepted by every DFA F_i iff $\downarrow r(\pi_i, M_{\xi_w})$ hold for all PSPs π_i iff $\downarrow r(\pi_0, M_{\xi_w})$.

Finally, consider any PSP π_{empty} such that the terminal point **exit** is unreachable from the initial point entry in π_{empty} (PSPs of this kind are called *empty* PSPs). Clearly, an empty PSP has no terminating runs, and therefore $\bigcap_{i=1}^{n} L(F_i) = \emptyset$ iff $\pi_0 \sim_{\mathcal{M}_0} \pi_{empty}$. Thus the Intersection Emptiness Problem for DFAs, which is known to be PSPACE-complete, is reducible in linear time to the equivalence problem for PSPs with mode switching. □

6. Conclusion and Future Work

We demonstrated that the equivalence problem for programs with mode switching is *PSPACE*-complete. Actually, the semantics of these programs makes it inevitable to face the Intersection Emptiness Problem for DFAs. This gives us a decision procedure which is confined to the manipulations with finite automata but at the expense of fairly large complexity. One could find some analogy between our construction of DFAs D_i and the "parallel stacking" technique introduced in [31]. The results obtained pose a number of open problems and we discuss some of them.

One of the most simple class of programs with undecidable equivalence problem was studied in [22]. The programs from this class are composed of four basic statements $\mathcal{A} = \{a_1, a_2, b_1, b_2\}$. In the framework of PSPs the semantics of such programs is specified by the set of structures $\mathcal{M}_2 = \{M' = \langle \mathbb{N}^2, (0,0), R', \xi\rangle\}$, where $\mathbb{N} = \{0, 1, 2, \dots\}$, $R'(a_1, (n,m)) = (n+1, m)$, $R'(a_2, (n,m)) = (n, m+1)$, $R'(b_1, (n,m)) = (0, m)$, $R'(b_2, (n,m)) = (n, 0)$. Notice that constant assignment statements b_1, b_2 are but a sort of a (separate) mode switches. In [7, 22] it was proved that two-head finite state automata can be simulated by the PSPs in the model $\mathcal{M}_2$ and this brings the equivalence problem for $\mathcal{M}_2$ into undecidability.

In essence, $\mathcal{M}_2$ captures two main features of the real program semantics: the effect of commutativity of some statements (a_1 and a_2) and the effect of constant assignments (b_1 and b_2). By divorcing these effects from each other we arrive at the model of programs with commutative statements $\mathcal{M}_1$ and the model of programs with mode switching $\mathcal{M}_0$. In [26, 32] it was shown that the equivalence problem for $\mathcal{M}_1$ is decidable within a time $O(n^2 \log n)$. In our paper we proved that the same problem for $\mathcal{M}_0$ is *PSPACE*-complete. In order to make a boarder between decidable and undecidable cases more precise we wonder: what is the complexity of the equivalence problem for the model $\mathcal{M}_{01}$ which may be placed between $\mathcal{M}_2$ and both $\mathcal{M}_0$ and $\mathcal{M}_1$? In contrast to $\mathcal{M}_2$, which involves two separate mode switches b_1 and b_2, we restrict ourselves in $\mathcal{M}_{01}$ only with a joint mode switch b_0 such that $R'(b_1, (n, m)) = (0, 0)$. Clearly, $\mathcal{M}_{01}$ is less expressive than $\mathcal{M}_2$. The equivalence problem for $\mathcal{M}_{01}$ was studied in [6], but its complexity is still unknown.

The equivalence-checking algorithms worked out in the framework of this model of programs may be used as the basis for the deobfuscation tools aimed at detecting metamorphic viruses. This could stimulate the development of more practical and efficient equivalence-checking procedures for $\mathcal{M}_0$ than that from Theorem 1. Thus, for example, we wonder, if it is possible to check within a polynomial time the equivalence $\pi' \sim_{\mathcal{M}_0} \pi'$ providing that every mode switch occurs in π', π'' at most k times and k is fixed.

References

[1] M. Christodorescu, S. Jha. Static analysis of executables to detect malicious patterns. In *Proceedings of the 12th USENIX Security Symposium (Security'03)*, 2003, p. 169–186.

[2] M. Christodorescu, S. Jha, S.A. Seshia, D. Song, R. E. Bryant. Semantics-aware malware detection. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy (Oakland 2005)*, 2005, (to be published).

[3] Collberg C., Thomborson C., Low D., Manufacturing cheap, resilient and stealthy opaque constructs. *Symposium on Principles of Programming Languages*, 1998, p.184–196.

[4] A.P.Ershov, Theory of program schemata. In *Proc. of IFIP Congress 71*, Ljubljana, 1971, p.93-124.

[5] E.P. Friedman. Equivalence problem for deterministic context-free languages and monadic recursive schemes. *J. Comput. and Syst. Sci.*, **14**, N 3, 1977, p. 344–359.

[6] A.B. Godlevsky. Some special cases of the termination and equivalence problems for automata, 1973, N 4, p. 90-–98 (in Russian)

[7] A.B. Godlevsky. On the one case of special problem of functional equivalence for discrete transducers. Cybernetics, 1974, N 3, p. 32–35 (in Russian)

[8] J.E. Hopcroft, R.M. Karp. A linear algorithm for testing equivalence of finite automata, Technical Report TR 71–114, Cornell University, Computer Science Dep., 1971.

[9] Ianov Iu I., On the equivalence and transformation of program schemes *Communications of the ACM*, 1:10 (1958), 8–12.

[10] G. Karakostas, R.J. Lipton, A. Viglas. On the complexity of intersecting finite state automata and NL versus NP. *Theoretical Computer Science*, **302**, 2003, p. 257–274.

[11] A. Kornai. Vectorized finite state automata. In: *Proceedings of the W1 workshop of the 12th European Conference on Artificial Intelligence*, Budapest, 1996, p. 36–41.

[12] D. Kozen. Lower bounds for natural proof systems. In *18th Annual Symposium on Foundation of Computer Science*, IEEE, 1977, p. 254-266.

[13] A.A.Letichevsky. On the equivalence of automata with final states on the free monoids having right zero. *Reports of the Soviet Academy of Science*, **182**, 1968, N 5 (in Russian).

[14] A.A.Letichevsky. Functional equivalence of discrete transducers. *Cybernetics*, 1970, N 2, p. 14–28.

[15] A.A.Letichevsky, Functional equivalence of finite transducers. III, Cybernetics, 1972, N 1, 1—4 (in Russian)

[16] A.A.Letichevsky, On the equivalence of automata over semigroup, Theoretic Cybernetics, 6, 1970, 3—71 (in Russian).

[17] A.A.Letichevsky, Equivalence and optimization of programs. In Programming theory, Part 1, Novosibirsk, 1973, 166-–180 (in Russian).

[18] A.A.Letichevsky, L.B.Smikun, On a class of groups with solvable problem of automata equivalence, Sov. Math. Dokl., 17, 1976, N 2, 341—344.

[19] L.P. Lisovik, Methalinear schemes with constant assignment, Programming and Computer software, N 2, 1985, 29-–38.

[20] L.P. Lisovik. Hard sets method and semilinear reservoir method with applications. *Lecture Notes in Computer Science*, **1099**, 1996, p. 219–231.

[21] D.C.Luckham, D.M.Park, M.S.Paterson, On formalized computer programs, *J. Comput. and Syst. Sci.*, **4**, 1970, N 3, p.220–249.

[22] G.N.Petrosyan, On one basis of statements and predicates for which the emptiness problem is undecidable. *Cybernetics*, 1974, N 5, p.23–28 (in Russian).

[23] R.I.Podlovchenko, The hierarchy of program models, Programming and Computer Software, 1981, N 2, 3—14.

[24] R.I.Podlovchenko, Semigroup program models, Programming and Computer Software, 1981, N 4, 3-–13.

[25] R.I.Podlovchenko, On the decidability of the equivalence problem on a class of program schemata having monotonic and partially commutative statements, Programming and Computer Software, 1990, N 5, 3—12.

[26] R.I.Podlovchenko, V.A.Zakharov, On the polynomial-time algorithm deciding the commutative equivalence of program schemata, *Reports of the Soviet Academy of Science*, **362**, 1998, N 6 (in Russian).

[27] R.I.Podlovchenko, On program schemes with commuting and monotonic statements, Programming and Computer software, N 5, 2003, 46–54.

[28] K. Rohloff, S. Lafortune. On the computational complexity of the verification of modular discrete-event systems. In *Proceedings of IEEE Conference on Decision and Control*, 2002, Las Vegas, NV, Dec., 2002.

[29] J.D.Rutledge, On Ianov's program schemata, *Journal of the ACM*, **11**, 1964, p.1–9.

[30] W.J. Savitch. Relationships between nondeterministic and deterministic space complexities. *Journal of Computer and System Science*, **4**, N 2, 1970, p. 177–192.

[31] L.G.Valiant, The equivalence problem for deterministic finite-turn pushdown automata, *Information and Control*, **25**, 1974, p.123–133.

[32] V.A. Zakharov, An efficient and unified approach to the decidability of equivalence of propositional program schemes. *Lecture Notes in Computer Science*, **1443**, 1998, p. 247–258.

[33] V.A. Zakharov, On the decidability of the equivalence problem for monadic recursive programs, Theoretical Informatics and applications, 34, N 2, 2000, 157-–171.

[34] V.A. Zakharov, The equivalence problem for computational models: decidable and undecidable cases. *Lecture Notes in Computer Science*, **2055**, 2001, 133–153.

[35] V.A. Zakharov, I.M. Zakharyaschev, An equivalence-checking algorithm for polysemantic models of sequential programs, Proceedings of the International Workshop on Program Understanding (14-16 July, Altai Mountains, Russia), 2003, 59-–70.

[36] V.A. Zakharov, I.M. Zakharyaschev, On the equivalence checking problem for a model of programs related with muti-tape automata, Lecture Notes in Computer Science, 3317, 2005, 293-–305