DOI: 10.15514/ISPRAS-2024-36(3)-10

О методах извлечения алгоритмов из бинарного кода

[1,2] *И.И. Кулагин,* ORCID: 0000-0003-2191-1578 <*i.kulagin@ispras.ru*>
[1,2,3] *В.А. Падарян,* ORCID: 0000-0001-7962-9677 <*vartan@ispras.ru*>
[1] *В.А. Кошкин,* ORCID: 0009-0009-1781-9622 <*trickyfox371@ispras.ru*>
[1] *Институт системного программирования им. В.П. Иванникова РАН, 109004, Россия, г. Москва, ул. А. Солженицына, д. 25.*
[2] *Московский государственный университет имени М.В. Ломоносова, 119991, Россия, Москва, Ленинские горы, д. 1.*
[3] *Национальный исследовательский университет «Высшая школа экономики», 101978, Россия, г. Москва, ул. Мясницкая, д. 20*

Аннотация. В работе предложен итеративный метод извлечения алгоритмов из бинарного кода и построения их высокоуровневого представления. Построение алгоритмов предложенным методом реализовано в виде анализа динамических слайсов. Метод базируется на алгоритме отслеживания потока данных в прямом или обратном направлениях и выполняет композицию динамических состояний в функциональные блоки двух уровней представления. Также предложены два уровня представления извлеченных алгоритмов – функциональная схема слайса и схема выполнения алгоритма. Функциональная схема слайса представляет собой структурированное представление слайса и является более низкоуровневым, чем схема выполнения алгоритма. Схемой выполнения алгоритма является представление, состоящее только из резюме функций и их параметров. Предложенный метод построения алгоритмов и способы их представления позволяют повысить продуктивность аналитика при решении задач анализа безопасности кода, улучшить качество полученных результатов анализа. Разработанные способы представления алгоритмов могут быть использованы и для реализации алгоритмов автоматического анализа безопасности кода. Кроме того, авторы выполнили обзор подходов к извлечению алгоритмов из бинарного кода и способов их представления инструментами статического анализа кода, рассмотрены их некоторые недостатки и ограничения.

Ключевые слова: динамический анализ кода; анализ бинарного кода; восстановление алгоритма; построение слайса.

Для цитирования: Кулагин И.И., Падарян В.А., Кошкин В.А. О методах извлечения алгоритмов из бинарного кода. Труды ИСП РАН, том 36, вып. 3, 2024 г., стр. 139–160. DOI: 10.15514/ISPRAS–2024–36(3)–10.

About methods of extracting algorithms from binary code

[1,2] *I.I. Kulagin* ORCID: 0000-0003-2191-1578 <*i.kulagin@ispras.ru*>
[1,2,3] *V.A. Padaryan* ORCID: 0000-0001-7962-9677 <*vartan@ispras.ru*>
[1] *V.A. Koshkin* ORCID: 0009-0009-1781-9622 <*trickyfox371@ispras.ru*>
[1] *Ivannikov Institute for System Programming of the RAS Alexander Solzhenitsyn st., 25, Moscow, 109004, Russia.*
[2] *Lomonosov Moscow State University, GSP-1, Leninskie Gory, Moscow, 119991, Russia,*
[3] *National Research University, Higher School of Economics, 20, Myasnitskaya Ulitsa, Moscow, 101978, Russia.*

Abstract. The paper proposes an iterative method for extracting algorithms from a binary code and constructing their high-level representation. The construction of algorithms using the proposed method is implemented in the form of analysis of dynamic slices. The method is based on an algorithm for tracking data flow in the forward and backward directions. Also, two levels of presentation of the extracted algorithms are proposed: a functional slice diagram and an algorithm execution diagram. The functional slice diagram is a structured slice representation and is a lower-level representation than the algorithm execution diagram. The flowchart of the algorithm is a representation consisting only of function models and their parameters. The proposed method for constructing algorithms and methods of their presentation allow to increase the analyst's productivity in solving problems of code security analysis, to improve the quality of the obtained analysis results. The developed ways of representing algorithms can be used to implement algorithms for automatic analysis of code security. In addition, the authors reviewed the approaches to extracting algorithms from binary code and how they are represented by static code analysis tools and considered some of their shortcomings and limitations.

Keywords: dynamic code analysis; binary code analysis; algorithm recovery; slice building.

For citation: Kulagin I.I., Padaryan V.A., Koshkin V.A. About methods of extracting algorithms from binary code. Trudy ISP RAN/Proc. ISP RAS, vol. 36, issue 3, 2024. pp. 139-160 (in Russian). DOI: 10.15514/ISPRAS-2024-36(3)-10.

1. Введение

Для решения некоторых задач анализа безопасности кода требуется наличие высокоуровневого представления алгоритма. К числу таких задач можно отнести задачи обратной инженерии, повышения понимаемости кода, переход к методам формальной верификации с использованием резюме функций, проверки корректности использования библиотечных интерфейсов и другие. Отсутствие высокоуровневого представления алгоритма существенно затрудняет решение таких задач, а в некоторых случаях делает их решение невозможным. Кроме этого, восстановленный алгоритм может использоваться при формальной верификации программы на этапе построения модели безопасности, а также в качестве вспомогательного источника данных для инструментов автоматического тестирования программного обеспечения. Например, знание об алгоритме разбора или формирования файла или сетевого пакета позволяет восстановить его формат, что в свою очередь может быть использовано при генерации набора тестовых данных для автоматического тестирования. Наличие высокоуровневого представления алгоритма позволяет не только увеличить скорость решения задач анализа кода, но и существенно улучшить качество полученных результатов.

Извлечение алгоритмов из бинарного кода является одной из базовых задач реверс-инжиниринга. Построение высокоуровневого представление алгоритма по бинарному коду в общем случае подразумевает выполнения следующих шагов. 1. *Определение уровней абстракций, на которых осуществляется построение алгоритма*. Данный шаг обусловлен тем, что при разработке программного обеспечения на этапе проектирования вводятся определенные абстракции, необходимые для того, чтобы скрыть излишнюю сложность

реализации той или иной функциональности. В результате такого проектирования программа представляет собой многоуровневую иерархию абстракций. Например, пусть имеется программа, реализующая алгоритм, небольшим фрагментом которого является разбор некоторого сообщения. Всего имеются 5 типов сообщений. Каждый тип сообщения обладает своим набором полей, для разбора каждого из которых используется свой алгоритм, реализованный определенной функцией. На этапе проектирования были введены абстракции (снизу вверх) уровня разбора определенного поля определенного типа ($parseFieldXofMsgTypeB$), уровня разбора сообщения определенного типа ($parseMsgTypeB$) и уровня разбора сообщения любого типа ($parseMsg$). Задача аналитика при извлечении алгоритма и построении его высокоуровневого представления заключается в том, чтобы выбрать такой уровень абстракции, который удовлетворял бы требованиям последующего анализа исследуемого алгоритма. В свою очередь инструмент анализа кода должен учитывать выбранный уровень при извлечении алгоритма и построении его высокоуровневого представления. 2. *Извлечение инструкций из бинарного кода*, принадлежащих анализируемому алгоритму, и локализация всех его входных параметров. Как правило, на данном шаге применяются алгоритмы, основанные на отслеживании потока данных в прямом или обратном направлениях, или алгоритмы построения прямого или обратного слайса. Результатом работы данного шага будет последовательность инструкций, реализующих вычисления извлекаемого алгоритма. 3. *Повышение уровня представления* извлеченной последовательности инструкций. В процессе повышения уровня представления выполняется преобразование полученной подпрограммы к виду, пригодному как для ручного анализа исследуемого алгоритма, так и для автоматического. Таким представлением может быть граф потока управления, программа на высокоуровневом языке, полученная в результате декомпиляции извлеченной подпрограммы, блок-схема и др.

Алгоритм считается извлеченным, если по построенному высокоуровневому представлению в автоматическом или полуавтоматическом режиме может быть получен работоспособный код, который на контрольных входных параметрах выдает ожидаемые значения выходных параметров.

Существующие промышленные программные платформы реверс-инжиниринга и анализа бинарного кода, такие как BAP [1], angr [2], IDA Pro [3], Ghidra [4], Cutter, Rizin [5], Radare2 [6] и др., сталкиваются с рядом ограничений и неразрешимых проблем при выполнении каждого из вышеперечисленных шагов извлечения алгоритма. Так, например, отсутствие информации времени выполнения при статическом анализе кода может привести к невозможности выбрать уровень абстракции, на котором требуется построить представление алгоритма. В свою очередь, используемые алгоритмы отслеживания потоков данных при динамическом анализе кода [7] приводят к тому, что выбранное множество машинных инструкций, представляющих извлеченный алгоритм, будет иметь слишком большие размеры. Изучение свойств извлеченного алгоритма в виде полученного множества инструкций затруднительно, если вообще возможно. Кроме того, часть этих инструкций может быть избыточной, а некоторые инструкции наоборот могут быть утеряны при построении слайса. По причине вышесказанного в работе предлагается итеративный метод извлечения алгоритмов из бинарного кода и построения их высокоуровневого представления. Разработанный метод позволяет извлечь алгоритм из бинарного кода с необходимым уровнем детализации и на заданном уровне абстракции. Предложенные два уровня представления извлеченного алгоритма позволяют выполнять изучение свойств алгоритма как в ручном, так и автоматическом режимах.

2. Методы извлечения алгоритмов из бинарного кода

Извлечение алгоритма и построение его высокоуровневого представления возможно при помощи статического или динамического анализа кода. В текущем разделе проводится обзор

существующих подходов к извлечению алгоритмов из бинарного кода программы методами статического и динамического анализа кода. А также предлагается разработанный авторами итеративный метод восстановления алгоритма путем отслеживания потока данных. Во время анализа методов извлечения алгоритмов подчеркиваются некоторые важные их недостатки и проблемы, а также способы их преодоления.

Основным примером в ходе описания методов извлечения алгоритмов из бинарного кода и построения их высокоуровневого представления будет служить программа $archive$, значимый фрагмент которой показан на Си-подобном псевдокоде на рис. 1. Программа выполняет архивирование и разархивирование данных.

```
1  int main(int argc, char *argv[]) {
2    char *text = readFromFile(argv[2]);
3    int len = strlen(text);
4    char *bufs[4] = {malloc(N), malloc(N), malloc(N), malloc(N)};
5    Handle handles[4];
6    Context ctx[3] = {getContext(), getContext(), getContext()};
7    if (argv[1] == "zip") {
8      strncpy(bufs[0], text, len / 2);
9      strncpy(bufs[1], text + len / 2, len / 2);
10     handles[0] = createHandle(bufs[0]);
11     handles[1] = createHandle(bufs[1]);
12     int size1 = compress(ctx[0], handles[0]);
13     int size2 = compress(ctx[1], handles[1]);
14     copyData(ctx[0], bufs[2]);
15     copyData(ctx[1], bufs[2] + size1);
16     handles[2] = createHandle(bufs[2]);
17     int resultSize = compress(ctx[2], handles[2]);
18     copyData(ctx[2], bufs[3]);
19     saveToFile(argv[3], size1, size2, bufs[3], resultSize);
20   } else if (argv[1] == "unzip") {
21     int size1 = getSize(text);
22     int size2 = getSize(text + sizeof(int));
23     handles[0] = createHandle(ctx[2], text + 2 * sizeof(int));
24     decompress(ctx[2], handles[0]);
25     copyData(ctx[2], bufs[0]);
26     strncpy(bufs[1], bufs[0], size1);
27     strncpy(bufs[2], bufs[0] + size1, size2);
28     handles[1] = createHandle(ctx[0], bufs[1]);
29     handles[2] = createHandle(ctx[1], bufs[2]);
30     size1 = decompress(ctx[0], handles[1]);
31     size2 = decompress(ctx[1], handles[2]);
32     copyData(ctx[0], bufs[3]);
33     copyData(ctx[1], bufs[3] + size1);
33     saveToFile(argv[3], bufs[3], size1 + size2);
34   }
35 }
```

Рис. 1. Фрагмент программы 'archive' – архивирования и разархивирования содержимого входного файла

Fig. 1. Fragment of the program 'archive' for compressing and decompressing the contents of the input file

В качестве входных параметров $archive$ принимает режим функционирования ($argv[1]$), имена двух файлов – входного ($argv[2]$) и результирующего ($argv[3]$). Программа может функционировать в двух режимах: в режиме архивирования содержимого входного файла и режиме разархивирования. Режим архивирования активируется установкой «$argv[1]$» в значение «zip», а режим разархивирования – «$unzip$». Рассмотрим каждый из этих режимов функционирования по отдельности.

В режиме архивирования программа выполняет следующие шаги.

Шаг 1. Считывает из входного файла данные (строка 2). Имя файла передано через аргумент командной строки $argv[2]$.

Шаг 2. Подготавливает контексты архивирования (строка 6). Контексты содержат информацию об используемом алгоритме сжатия данных и владеют буферами, хранящими данные. Контекст создается при помощи абстрактной функции *getContext*, реализация которой может находиться в закрытой динамической библиотеке, которая может быть недоступна на этапе статического анализа.

Шаг 3. Разбивает входной буфер с текстом на 2 части. Предполагается, что текст архивируется по частям, при этом части могут быть сжаты с использованием разных алгоритмов (строки 8-9).

Шаг 4. Копирует буферы в контексты архивирования. Необходимо отметить, что программа не оперирует буферами в явном виде, вместо этого функция *createHandle* возвращает специальный хендл буферов для обращения к ним (строки 10-11).

Шаг 5. Архивирует данные по частям, используя разные контексты архивирования и хендлы буферов (строки 12-13). Алгоритм сжатия неизвестен на данном уровне абстракций, информация о нем хранится в контексте. Результат архивирования хранится в контексте, доступ к нему осуществляется при помощи функции *copyData*.

Шаг 6. При помощи функции *copyData* копирует сжатые данные, хранящиеся в контекстах 0 и 1, во временный буфер (строки 14-15). В результате сжатые данные будут содержаться во временном буфере в объединенном виде.

Шаг 7. Архивирует временный буфер, используя контекст 3 (строки 16-17).

Шаг 8. Копирует результат сжатия буфера из контекста 3 в результирующий буфер и сохраняет результат в выходной файл, название которого хранится в *argv*[3] (строки 18-19).

В результате после выполнения программы в режиме архивирования выходной файл будет содержать следующую информацию: размеры сжатых на шаге 5 буферов и результирующий буфер. Размеры необходимы для того, чтобы в режиме разархивирования корректно разделить буферы на 2 части, и каждую разархивировать по отдельности.

Рассмотрим работу программы в режиме разархивирования входного файла. В этом режиме шаги 1-2 совпадают с соответствующими шагами режима архивирования, разница лишь в том, что входным файлом является результирующий файл режима сжатия, а в выходной файл будут записаны разархивированные данные. Оставшиеся шаги режима сжатия выполняют следующие действия.

Шаг 3. Извлекает из заголовка прочитанных на шаге 1 данных размеры сжатых буферов (строки 21-22).

Шаг 4. Инициализирует входные данные в контексте (строка 23).

Шаг 5. При помощи контекста *ctx*[2] разархивирует данные и копирует результат из контекста во временный буфер (строки 24-25).

Шаг 6. Разбивает временный буфер *bufs*[0] на 2 части на основании размеров, считанных из заголовка входных данных (строки 26-27).

Шаг 7. Инициализирует каждую часть буфера в контекстах 0 и 1 и разархивирует их при помощи соответствующего контекста (строки 28-31).

Шаг 8. Копирует разархивированные части результирующего буфера и объединяет их в один буфер (строки 32-33).

Шаг 10. Сохраняет разархивированные данные в результирующий файл (строка 33).

Далее в работе предполагается, что программа $archive$ скомпилирована любым из промышленных компиляторов (MSVC, Clang, GCC или другим компилятором) с включенным максимальным уровнем оптимизации, а также из кода удалена отладочная информация. Построение высокоуровневого представления алгоритма осуществляется на основе анализа получившегося исполняемого файла.

Из представленного псевдокода удалены все проверки корректности выполнения функций с целью сокращения объема примера. В практической реализации такие проверки должны быть.

Код, реализующий проверки корректности, не вносит новых проблем анализа бинарного кода в процессе извлечения алгоритма и его высокоуровневого построения, принципиально отличающихся от тех, что будут рассмотрены далее в работе. Как правило, такие проверки лишь добавляют в граф потока управления новые пути выполнения и увеличивают общий объем анализируемого кода.

2.1. Извлечение алгоритма методом статического анализа бинарного кода

Большинство существующих инструментов статического анализа кода, такие интерактивные дизассемблеры и программные платформы реверс-инжиниринга, как IDA Pro [3], Ghidra [4], Cutter, Rizin [5], Radare2 [6] и др. позволяют представить функции анализируемой программы в виде графа потока управления (Control Flow Graph – CFG). При наличии интеграции с декомпилятором, например hex-rays [3], RetDec [8], rev.ng [9] и др., функции анализируемой программы могут быть декомпилированы и представлены на языке высокого уровня. То есть, инструменты статического анализа кода позволяют построить представление алгоритма двух уровней: в виде графа потока управления текущей функции и в виде кода на высокоуровневым языке, полученного в результате декомпиляции. И в том, и в другом случае область построения алгоритма ограничена границами текущей анализируемой функции, а выбор уровня абстракций – вызовами внутри этой функции.

Безусловно при наличии всех исполняемых модулей программы имеется возможность построить граф вызовов функций, для каждой функции построить граф зависимостей программы (Program Dependence Graph – PDG), после, на основе получившихся графов, построить граф зависимостей системы (System Dependency Graph – SDG). Однако даже для небольшого синтетического примера получившийся SDG будет иметь внушительные размеры, затрудняющие изучение свойств извлекаемого алгоритма в ручном режиме.

Подход, при котором в качестве целевого способа представления алгоритма выбран высокоуровневый код, полученный в результате декомпиляции, также не лишен недостатков. Распространенной практикой извлечения алгоритма при таком подходе является декомпиляция отдельных функций программы и построение на основе получившегося декомпилированного кода новой программы, реализующей только анализируемый алгоритм. Построение такой программы выполняется практически полностью, за исключением этапа декомпиляции, в ручном режиме, а это значит, что потребуется выполнить внушительный объем работы по анализу кода, представленного не только в виде CFG, PDG или SDG, но и в дизассемблированном виде.

Одним из серьезных недостатков статического анализа кода является отсутствие информации времени выполнения. К такой информации могут относиться, например целевые адреса инструкций передачи управления с косвенной адресацией, код динамических библиотек и системных вызовов, код, динамически порождаемый виртуальными машинами, и др. Преодолеть данный недостаток в той или иной степени возможно при помощи абстрактной интерпретации бинарного кода с целью восстановления переменных и указателей и аппроксимации их значений [10][11]. Однако не все значения переменных и указателей удается восстановить путем абстрактной интерпретации, часть из них остается неизвестными. В контексте получения алгоритма путем выполнения декомпиляции некоторых функций наличие неизвестных указателей и переменных может привести к тому, что декомпилятор на этапе оптимизации ложно распознает какой-то фрагмент кода как недостижимый или мертвый и удалит его как избыточный. А это, в свою очередь, приведет к тому, что извлеченный алгоритм будет являться некорректным, так как в нем будут

отсутствовать некоторые его фрагменты. Или же наоборот, декомпилятор может не найти недостижимый или мертвый код, и тогда извлеченный алгоритм будет содержать избыточные вычисления, что в свою очередь может существенно увеличить размер результирующего алгоритма и затруднить изучение его свойств.

Отсутствие информации времени выполнения делает невозможным выбор уровня абстракций для выполнения построения алгоритма. Это может произойти, если анализируемая программа содержит обращения к интерфейсам библиотек, но на момент выполнения статического анализа не представляется возможным узнать, какая именно реализация того или иного интерфейса будет использоваться. Либо исполняемый модуль, реализующий этот интерфейс, вообще отсутствует, а выбор конкретного модуля осуществляется непосредственно во время выполнения и зависит от текущего окружения.

Важно отметить, что невозможность выбрать уровень абстракций при построении высокоуровневого представления алгоритма является критическим недостатком. Построенный алгоритм должен описывать последовательность преобразований входных параметров в выходные, а отсутствие требуемого уровня абстракций может привести к потере части параметров и преобразований над ними, что является недопустимым при построении алгоритма. Например, на верхнем уровне абстракций использование параметров алгоритма в явном виде может быть невозможным, а доступ к ним осуществляется только при помощи идентификаторов. Сами же параметры используются на более низких уровнях. Однако для извлечения корректного алгоритма его построение необходимо выполнять на более низком уровне, на котором все преобразования осуществляются над самими параметрами, а не над их идентификаторами. Если во время статического анализа кода требуемый уровень абстракций выбрать не удастся, то построение корректного алгоритма невозможно.

Рассмотрим вышеперечисленные проблемы на примере извлечения алгоритма формирования результирующего буфера и построения его высокоуровневого представления из бинарного кода для рассмотренной ранее программы *archive* (рис. 1) в режиме архивирования. Как уже говорилось ранее, первым шагом к построению высокоуровневого представления алгоритма является выбор уровня абстракций, на котором будет выполняться анализ и извлечение алгоритма. И сразу мы стакиваемся с непреодолимым ограничением. Программа *archive* использует высокоуровневые интерфейсы *compress*/*decompress*, которые являются высоким уровнем абстракций. На данном уровне скрыта не только реализация алгоритмов архивирования/разархивирования, но и входные буферы с данными. Доступ к этим параметрам осуществляется при помощи специальных хендлов. Код функций, реализующих алгоритмы архивирования/разархивирования может быть недоступен на момент статического анализа, так как конкретная реализация может быть выбрана из множества доступных в системе непосредственно во время выполнения. В различных окружениях выполнения могут иметься отличающиеся реализации одного и того же алгоритма или разные алгоритмы. То есть, выбор конкретной реализации полностью определяется состоянием окружения в момент выполнения программы. В этих условиях имеется возможность извлечь алгоритм и построить его высокоуровневое представление только на уровне публичных интерфейсов *compress*/*decompress* и хендлов буферов. Стоит также отметить, что уровень абстракций влияет и на количество входных параметров алгоритма. В контексте данного примера, если выбрать уровень, ниже, чем уровень публичных интерфейсов, то к списку входных параметров добавятся параметры, необходимые для работы алгоритмов архивирования и разархивирования. Чем более глубокий уровень анализа будет выбран, тем большим количеством входных параметров будет обладать анализируемый алгоритм.

Следующим шагом на пути к построению высокоуровневого представления является извлечение из программы инструкций, принадлежащих анализируемому алгоритму. На данном шаге, как правило, применяются методы, основанные на отслеживании потока

данных в прямом или обратном направлении, то есть выполняется построение прямого или обратного слайса [12]. При отслеживании потока данных в прямом направлении начальный критерий слайса включает в себя множество (или подмножество) входных параметров алгоритма. Во время обработки очередной, принадлежащей отслеживаемому потоку данных инструкции неизвестно, вычисляет ли она фрагмент результирующих параметров алгоритма или выполняет второстепенные вычисления. При отслеживании потока данных в прямом направлении факт принадлежности каждой инструкции динамического слайса восстанавливаемому алгоритму может быть установлен только во время обработки последней инструкции слайса. Таким образом, использование на данном шаге прямого слайса требует сохранения всех фактов распространения потока данных и не позволяет игнорировать часть инструкций из-за недостатка информации о влиянии их результатов на выходные параметры алгоритма. Поэтому прямой слайс может иметь большие размеры, чем обратный и содержать больше избыточных инструкций.

Для построения обратного слайса начальный критерий должен содержать выходные параметры алгоритма. При отслеживании потока данных в обратном направлении в первую очередь будут рассматриваться инструкции, непосредственно участвующие в вычислении фрагментов результирующих параметров извлекаемого алгоритма. Поэтому для обратного слайса решение о включении или невключении инструкции в результирующий слайс может приниматься сразу, а не в конце работы алгоритма отслеживания потока данных, как в случае прямого слайса. В результате чего обратный слайс может содержать существенно меньше избыточных инструкций, чем прямой слайс. По этой причине целесообразно выполнять построение обратного слайса относительно начального критерия – выходного параметра, а не прямого слайса относительного входных параметров. Кроме того, эмпирическим путем было установлено, что чаще всего количество входных параметров гораздо больше количества выходных.

Отсутствие информации времени выполнения также является существенной проблемой для алгоритма построения прямого или обратного слайса. Как результат, в построенный слайс, содержащий инструкции извлеченного алгоритма, могут попасть избыточные инструкции либо наоборот, часть инструкций будет потеряна.

Продолжая текущий пример – извлечение алгоритма формирования результирующего буфера программой *archive* (рис. 1) в режиме архивирования – необходимо построить обратный слайс. Инструкция вызова функции *saveToFile* на 19-ой строке псевдокода на рис. 1 могла бы являться критерием слайса, так как код, выполняющий архивирование, расположен выше данной инструкции. Результат архивирования сохраняется в буфер *buf*[3], однако использовать его как начальный элемент для построения обратного слайса невозможно, так как во время статического анализа отсутствует такое понятие, как буфер в памяти. При помощи абстрактной интерпретации и моделирования семантики функций работы с памятью в некоторых случаях возможно аппроксимировать состояние динамической памяти и стека [10-11], тем самым локализовав некоторые буферы. Эта информация может быть использована при построении статического слайса для уточнения потока данных. Тем не менее, далеко не все буферы удастся обнаружить подобными методами, и часть из них останется неизвестной.

Подытоживая выше сказанное, используя возможности таких платформ, как IDA Pro, Ghidra, Cutter, полученное представление алгоритма методами статического анализа кода будет соответствовать CFG функции *main*.

Представление алгоритма в виде CFG не только не будет отображать буферные параметры извлекаемого алгоритма, такие как сжатые данные, но и извлеченный алгоритм содержит код как режима архивирования, так и разархивирования. Хотя требовалось извлечь алгоритм работы *archive* только в режиме архивирования.

Помимо извлечения алгоритма путем построения статического обратного слайса, одним из интересных направлений является получение алгоритма с помощью выполнения частичных вычислений. Иначе говоря, в результате частичной специализации исходной программы конкретными значениями некоторых входных параметров. В контексте извлечения алгоритма работы программы *archive* в режиме архивирования необходимо выполнить подстановку входного параметра *argv*[1] = «*zip*». В результате будет сгенерирована специализированная или остаточная программа, выполняющая только сжатие данных.

Среди имеющихся работ, выполняющих частичное выполнение бинарного кода для задач обратной инженерии, особенно хочется отметить работы [13-15]. Рассмотрим основную идею данного подхода в следующем разделе.

2.2. Извлечение алгоритма методом частичного выполнения бинарного кода

Частичное вычисление [16] – это способ специализации программ, который может быть применим как для задач оптимизации программ, так и для задач обратной инженерии, в частности для извлечения алгоритма из бинарного кода, реализующего целый набор алгоритмов (например, извлечение небольшого кодировщика текста, встроенного в веб-сервер). Несмотря на богатый набор подходов к перезаписи бинарного кода с целью выполнения деобфускации [17-18], супероптимизации [19-20] или инструментации проверками безопасности [21-23], подходов к частичному выполнению бинарного кода не так много. Кроме того, алгоритмы частичного выполнения исходного кода [24-26] не дадут удовлетворительных результатов при их прямом применении к бинарному коду.

Данный раздел посвящен алгоритму частичного выполнения бинарного кода, описанному в работах [13-15]. Предлагаемый в этих работах подход к частичному выполнению бинарного кода основывается на классическом двухфазном алгоритме анализа времен связывания (Binding-Time analysis – BTA, BT-analysis, BT-анализ) с последующей специализацией. Входными данными для алгоритма частичного вычисления машинного кода являются бинарный код B и разделение входных параметров для B на статические (S) и динамические (D). Значения для статических входов известны во время специализации; значения для динамических входов неизвестны во время специализации. Алгоритм частичного вычисления генерирует бинарный код B_S, удовлетворяющий уравнению

$$[\![B]\!](S, D) = [\![B_S]\!](D),$$

где $[\![\cdot]\!]$ обозначает значащую функцию для инструкций бинарного кода. Выполнение B_S с входом D дает те же результаты, что и выполнение B с входами S и D. Однако B_S специализирован по отношению к S и может быть значительно быстрее и меньше, чем B.

Рассмотрим пример частичного вычисления. На рис. 2 представлена программа вычисления суммы для архитектуры IA-32, которая принимает значения a, v и n в качестве входных данных и вычисляет $a + b[n]$. Для массива b память выделяется на стеке, и он содержит 5 элементов. В строках 10-16 элементы b инициализируются значением v. n принимает значение от 0 до 4. Входы a, v и n сохраняются в регистрах eax, ebx и ecx, соответственно, в начале программы. Вывод сохраняется в регистре eax в конце программы. Частичное вычисление будем выполнять относительно входного значения $v = 1$.

Для выполнения смешанного вычисления необходимо разделить инструкции на статические и динамические. Такое разделение инструкций выполняется путем BT-анализа, который в свою очередь выполняет построение прямого слайса. В качестве критерия слайса выступят инструкции 2 и 6. Прямой слайс включает только инструкции, выделенные на рис. 2 серым цветом, которые классифицируются как динамические. Остальные инструкции классифицируются как статические.

После того как было получено разделение инструкций, необходимо выполнить все инструкции, помеченные как статические. В результате такого выполнения потребуется перезаписать и часть динамических инструкций, если значение какого-либо из их параметров будет вычислено. Такая перезапись инструкций является одной из многих технических сложностей практических реализаций алгоритмов частичного вычисления. На рис. 2 представлен результат частичного вычисления – остаточная программа рассматриваемого примера.

```
1  lea  esp,[ esp-4 ]          11 jl   L1
2  mov [ esp], eax             12 mov ebx,[ esp+28]
3  lea  esp,[ esp-4 ]          13 mov edx,[ esp+20]
4  mov [ esp], ebx             14 mov [ esp+edx*4 ], ebx
5  lea  esp,[ esp-4 ]          15 sub [ esp+20],1
6  mov [ esp], ecx             16 jmp  L2
7  lea  esp,[ esp-4 ]          17 L1:mov  eax,[ esp+32]
8  mov [ esp], 4               18 mov ecx,[ esp+24]
9  sub  esp, 20                19 mov ebx,[ esp+ecx*4]
10 L2:cmp [  esp+20],0         20 add  eax, ebx
                               21 add  esp,36

0₁ mov esi , esp               13₂ mov [ esi -24 ],1
1₁ lea  esp,[ esi -4 ]         13₃ mov [ esi -28 ],1
2₁ mov [ esp], eax             13₄ mov [ esi -32 ],1
5₁ lea  esp,[ esi -12 ]        13₅ mov [ esi -36 ],1
6₁ mov [ esp], ecx             16₁ mov eax,[ esp+32]
9₁ lea  esp,[ esi -16 ]        17₁ mov ecx,[ esp+24]
9₂ sub  esp,20                 18₁ mov ebx,[ esp+ecx*4 ]
13₁ mov [ esi - 20],1          19₁ add  eax, ebx
```

Рис. 2. Пример частичного вычисления программы, выполняющей $\boldsymbol{a + b[n]}$.
Слева – исходная программа, справа – остаточная программа
Fig. 2. Partial evaluation of the program that performs $\boldsymbol{a + b[n]}$.
on the left – original program, on the right – residual rewritten program

Использование частичной специализации для извлечения алгоритма возможно по двум основным сценариям. При первом сценарии полученная специализированная программа будет являться результатом извлечения алгоритма. При втором – специализированная программа становится исходной программой для последующего анализа, в результате которого может быть получено итоговое представление или может быть принято решение о выполнении повторной специализации уже специализированной программы. Повторная специализация может потребоваться в том случае, если во время анализа будут обнаружены дополнительные входные параметры, для которых можно выполнить подстановку конкретными значениями. Частичные вычисления позволяют сократить количество инструкций анализируемой программы и число возможных путей выполнения, что несомненно положительно сказывается на эффективности последующего анализа в ручном и автоматическом режимах.

Ввиду того, что основной частью специализации программ является BT-анализ, который основан на алгоритме построения прямого слайса, такой способ извлечения алгоритма обладает всеми теми же недостатками, что и построение статического слайса. Кроме того, для специализации требуется точно определить множество входных аргументов извлекаемого алгоритма и выбрать подмножество тех, для которых будет выполнена подстановка конкретных значений. Для проведения такого анализа, как правило, уже требуется иметь высокоуровневое представление исследуемого алгоритма. Задача локализации входных параметров практических программ (например, мессенджеров) может оказаться неразрешимой в условиях отсутствия такого представления.

Подводя итог рассмотрению методов извлечения алгоритмов и построению их высокоуровневого представления путем статического анализа бинарного кода, необходимо выделить следующее:

1) выбор уровня абстракций, на котором будет выполняться построение алгоритма, ограничен из-за отсутствия информации, доступной во время выполнения. Уровень, доступный во время проведения статического анализа кода, может оказаться недостаточным, и тогда получение корректного алгоритма будет невозможным;
2) как правило, выбранный уровень абстракции влияет на количество входных и выходных параметров. Доступных уровней абстракций во время проведения

статического анализа кода может быть недостаточно, и тогда локализовать часть входных параметров будет невозможно, а результирующее представление алгоритма не будет отражать процесс преобразования входных параметров в выходные. В результате алгоритм будет построен некорректно;

3) извлеченный при помощи статического обратного слайса алгоритм, как правило, содержит большое число избыточных инструкций. Например, в случае *archive* в построенном обратном слайсе будут содержаться не только инструкции режима архивирования, но и разархивирования. Избыточные инструкции затрудняют изучение свойств анализируемого алгоритма;
4) отсутствие информации времени выполнения может привести к тому, что часть инструкций анализируемого алгоритма будет потеряна в процессе построения обратного слайса или при декомпиляции (в случае, если в качестве результирующего представления извлеченного алгоритма требуется получить программу на высокоуровневом языке). Такой алгоритм является некорректным;
5) абстрактная интерпретация может быть использована для восстановления некоторой информации времени выполнения, например, значений переменных и указателей;
6) использование частичных вычислений для получения программы, специализированной конкретными значениями некоторых входных параметров, требует решения ряда сложных технических проблем, а именно, необходимо реализовать специализатор, анализ времен связывания, алгоритм синтеза специализированных машинных инструкций и др. Кроме того, в основе алгоритма BTA лежит построение прямого статического слайса, а это значит, что недостаток информации времени выполнения также критичен для использования частичных вычислений, как и для построения статического слайса.

Как можно заметить, основная часть проблем, возникающих при извлечении алгоритма из бинарного кода методами статического анализа кода связана с недостатком информации времени выполнения. Построение высокоуровневого представления алгоритма методами динамического анализа кода позволяет избежать подобных проблем. По этой причине авторами данной работы был предложен метод извлечения алгоритмов и построения их высокоуровневого представления путем динамического анализа кода, которому и посвящена оставшаяся часть работы.

2.3. Извлечение алгоритмов методом динамического анализа бинарного кода

Динамический анализ бинарного кода лишен недостатков, связанных с отсутствием информации времени выполнения. Кроме того, во время динамического анализа имеется возможность выразить такое понятие, как «буфер в памяти» и, тем самым, конкретизировать отслеживаемый поток данных. Авторами предложен метод построения высокоуровневых представлений алгоритмов по динамическим слайсам программ.

Предложенный метод требует выполнения некоторых предварительных действий. Этот подготовительный обычно включает в себя следующие шаги:

1) подготовка рабочего окружения выполнения исследуемой программы;
2) подготовка контрольных входных параметров исследуемой программы, которые обеспечат выполнение программы по интересующему сценарию.

Динамический слайс представляет собой подмножество выполненных инструкций анализируемой программы. Такой слайс может быть получен в результате анализа последовательности снимков состояний программы перед выполняемыми инструкциями. Последовательность состояний программы и выполняемых инструкций может быть получена, например, при помощи средств динамического бинарного инструментирования,

различных отладчиков и др. Инструменты детерминированного воспроизведения работы процессора и периферийных устройств на базе полносистемного эмулятора, например QEMU [27], являются негласным промышленным стандартом в данном в этой области и активно используются для получения последовательности таких состояний.

Извлечение алгоритмов из бинарного кода по динамическом слайсу и построение их высокоуровневого представления предложенным методом представляет собой итеративный процесс. На каждой итерации данного процесса формируется алгоритм с заданным уровнем абстракции. Уровень абстракций, используемый при извлечении исследуемого алгоритма, задается при помощи введения *резюме функций* соответствующих уровней. Резюме функции представляет собой высокоуровневое описание функции в виде ее названия, названия ее входных/выходных параметров и описание, при помощи специальных выражений, способа передачи входных параметров в функцию или выходных из нее (через регистры, стек или, в случае неявных параметров, при помощи адреса используемой глобальной переменной). Например, пусть имеется функция $testFunc$, принимающая один входной параметр in – буфер размером 16 байт. Входной параметр передается через стек в виде указателя на начало буфера. При описании параметра резюме функции $testFunc$ можно было бы ограничиться только одним параметром $inPtr = v(esp + 4, 4)$, указывающим на то, что указатель на буфер расположен в стековом кадре по смещению 4 и его размер 4 байта. Но при восстановлении алгоритма необходимо учитывать не указатель, а буфер, который будет использоваться внутри функции. Поэтому входной параметр $inPtr$ помечается как вспомогательный, и добавляется еще один параметр – буфер $in = v(inPtr, 16)$. Таким образом, в процессе построения высокоуровневого представления алгоритма будут использоваться те уровни абстракций, функции которых описаны в терминах резюме функций. В конце каждой итерации построения высокоуровневого представления алгоритма может быть получено его представление двух уровней: низкоуровневое – *функциональная схема слайса* и высокоуровневое – *схема выполнения алгоритма*.

Функциональная схема слайса [28] является декомпозицией динамического слайса на функциональные блоки. Основными элементами такой схемы являются: 1) точка (p_i) – представляет команду на определенном шаге выполнения анализируемой программы; 2) буфер (b_i) – представляет ячейку абстрактной памяти в определенной точке выполнения программы (на определенном шаге выполнения программы). Это может быть ячейка памяти в виртуальном адресном пространстве или неразрывная последовательность нескольких таких ячеек, регистр или его младшая/старшая часть.

Элементы функциональной схемы слайса формируют гиперграф (рис. 3 – слева). Ребра в гиперграфе отображают зависимости по данным. Вершины гиперграфа типа «точка» могут быть объединены в подмножества, называемые фрагментами (f_i), а фрагменты – в суперблоки (s_i).

Вершины-фрагменты соответствуют фрагментам кода динамического слайса, то есть последовательности инструкций, не содержащей инструкций вызова функций и возврата из них. Например, если функция A вызывает только одну функцию B, которая не содержит вызовов функций, то в этом случае будет сформировано три фрагмента кода: 1) F_{A1}, содержащий последовательность команд функции A до вызова функции B, 2) F_B – последовательность команд функции B, 3) F_{A2} – последовательность команд функции A, которые выполнялись после возврата из функции B. Суперблоки соответствуют экземплярам вызова функций и поэтому могут состоять только из фрагментов, принадлежащих экземпляру одной функции. Буферы соответствуют структурам данных программы и определяют интерфейсы взаимодействия между фрагментами и суперблоками. Также они характеризуют значение потока данных на момент входа в фрагменты или суперблоки или выхода из них. Для суперблоков входные и выходные аргументы определяются буферами.

Схема выполнения алгоритма представляет собой блок-схему уровня резюме функций (рис. 3 – справа). Она включает в себя только резюме функций и их входные/выходные параметры, а при помощи ребер задаются отношения между выходными и входными параметрами. Схема выполнения алгоритма может содержать ребра двух типов, показанные на рис. 3 справа сплошной и пунктирной линиями. Первый тип ребер (показанный сплошными линиями) отражает прямое перемещение потока данных, не содержащее никаких преобразований. Второй тип (показанный пунктирными линиями) отражает перемещение потока данных, сопровождаемое неучтенными преобразованиями, т. е. преобразованиями, непокрытыми резюме функций. В процессе итеративного построения высокоуровневого представления алгоритма ребрам второго типа нужно уделять особое внимание и постепенно покрыть резюме функций все неучтенные преобразования или большую их часть.

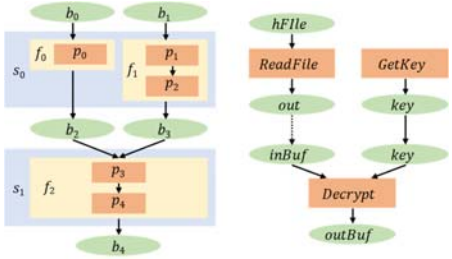


Рис. 3. Функциональная схема слайса (слева) и схема выполнения алгоритма (справа)
Fig. 3. Slice functional diagram (on the left) and algorithm execution diagram (on the right)

Рассмотрим подробно три основных шага, выполняемых на каждой итерации процесса восстановления алгоритма. **На первом шаге** осуществляется построение динамического слайса и построение по нему функциональной схемы слайса. Как уже было сказано ранее, количество выходных параметров алгоритма значительно меньше количества входных, поэтому на данном этапе целесообразно строить именно обратный слайс. Для этого необходимо найти точку выполнения программы, в которой извлекаемый алгоритм завершил свою работу, и имеется возможность описать начальный буфер, хранящий результат вычислений. Чаще всего такими точками являются выполняющие системные вызовы инструкции либо инструкции вызова известных библиотечных функций.

Алгоритм построения прямого или обратного слайса основан на отслеживании зависимостей между процессорными инструкциями. Такие зависимости отражают факт переноса информации между инструкциями и бывают следующих видов: 1) функциональные зависимости, 2) адресные зависимости, 3) зависимости по управлению. Функциональные зависимости отражают зависимости по данным и возникают, когда выходной операнд одной инструкции используется в качестве входного операнда другой инструкцией. Адресная зависимость возникает в случае вычисления одной инструкцией адреса, который используется другой инструкцией для доступа к ячейке памяти. Зависимости по управлению возникают в случае, если факт выполнения одной инструкции зависит от результата выполнения другой. Одной из проблем, возникающих при отслеживании зависимостей, является проблема ложных зависимостей, то есть таких, которые не являются значимыми с точки зрения извлекаемого алгоритма. Иначе говоря, инструкции, порождающие такие зависимости, не участвуют в вычислении результата извлекаемого алгоритма.

Ложные зависимости приводят к тому, что полученный слайс может содержать избыточные инструкции, в результате чего его размер будет существенно увеличен. Отдельным перспективным направлением для научных исследований является разработка методов и алгоритмов для сокращения размера слайса путем исключения из него инструкций с незначимыми зависимостями, то есть «прочистка» зависимостей. Сложность разработки

подобных методов заключается в том, что они являются эвристическими и требуют значительной базы эмпирических знаний, которые могут зависеть от процессорной архитектуры и характера анализируемого приложения. В разработанном методе извлечения алгоритмов реализованы некоторые эвристические алгоритмы «прочистки» слайса, позволяющие существенно сократить его объем, однако в данной работе они не описываются.

Вторая серьезная проблема, возникающая при построении слайса, связана с необходимостью отслеживать информационный поток, а не зависимости. В ряде случаев информационный поток может быть нетривиально выражен через зависимости по управлению. В этом случае распознать информационный поток в автоматическом режиме во время построения слайса невозможно. Подобные ситуации приводят к тому, что в построенном слайсе может отсутствовать часть инструкций, являющихся значимыми с точки зрения извлекаемого алгоритма. Таким образом, построенный алгоритм будет являться некорректным, либо его построение может быть невозможно из-за отсутствия некоторых значимых фрагментов. На рис. 4 представлен пример табличного копирования, который очень хорошо демонстрирует суть вышесказанного. Проблема заключается в том, что в буфер «*dst*» происходит копирование не элемента буфера «*src*» или элемента таблицы «*symbol_table*», а значения индекса найденного элемента в таблице. При построении обратного слайса относительно буфера «*dst*» в отслеживаемое множество не попадает буфер «*src*». Как следствие, в полученном слайсе будут отсутствовать инструкции вычисления буфера «*src*», которые могут быть необходимы для восстановления алгоритма.

```
1  symbol_table[SYMBOL_TABLE_SIZE] = { ... };

2  memcpy By ContorDeps(src, dst) {
3    for  (i   = 0;  i  < len(src); i++) {
4      for  (j =   0; j < SYMBOL_TABLE_SIZE; j++) {
5        if  (src  [i] == symbol_table[j]) {
6          dst[i  ] = j;
7        }
8      }
9    }
10 }
```

Рис. 4. Псевдокод функции табличного копирования
Fig. 4. Pseudocode of the indirect copy function

Для решения данной проблемы в рамках разработанного метода извлечения алгоритмов предлагается использовать резюме функций. На этапе создания резюме функций аналитик явным образом описывает зависимости между входными и выходными параметрами создаваемых резюме функций, тем самым конкретизируя информационный поток.

Построенный обратный слайс даже для синтетических примеров и даже после применения эвристик удаления избыточных инструкций («прочистки» слайса) может иметь внушительные размеры. Выполнять анализ извлеченного алгоритма, представленного в виде последовательности машинных инструкций, затруднительно, а для реальных, сложных программ практически невозможно. По этой причине после извлечения алгоритма выполняется построение его представления в виде функциональной схемы слайса, о которой говорилось ранее. Функциональная схема слайса структурирует построенный слайс в виде иерархических функциональных элементов. Ее иерархическая организация позволяет скрывать избыточную детализацию несущественных частей алгоритма и наоборот, части, являющиеся важными для конкретного анализа, раскрывать до уровня инструкций. Более детально с функциональной схемой слайса и способом ее построения можно ознакомиться в работе [28].

На втором шаге выполняется построение схемы выполнения алгоритма по построенной функциональной схеме слайса. В результате выполнения первого шага будет получена

функциональная схема слайса, часть суперблоков и буферов которой будут соответствовать резюме функций и их параметрам. Для таких суперблоков внутреннее содержимое скрыто, так как предполагается, что аналитик в момент создания резюме функции убедился, что реализация функции корректна и в контексте извлекаемого алгоритма может быть представлена как атомарное преобразование входных параметров в выходные. Функциональная схема слайса формирует гиперграф, на разных уровнях которого хранятся инструкции и резюме функций, связанные между собой ребрами. На ребрах расположены буферы и параметры резюме функций, определяющие зависимости по данным и задающие направления потоков данных. Таким образом, для построения схемы выполнения алгоритма необходимо по графу функциональной схемы слайса распространить информацию о достижимости одними резюме функций других через параметры и буферы. В итоге схема выполнения алгоритма будет содержать только те резюме функций и их параметры, которые присутствуют на функциональной схеме слайса.

Схема выполнения алгоритма содержит ребро прямого перемещения потока данных (обозначаемое сплошной линией) от выходного параметра out_i резюме функции A ко входному параметру in_i резюме функции B, если на функциональной схеме слайса имеется путь от соответствующего параметра out_i к параметру in_i, не содержащий узлов типа точка. Напомним, что узлы типа точка соответствуют инструкциям, выполняющимся на конкретном шаге работы программы. Иначе говоря, поток данных от параметра out_i к параметру in_i не подвергался преобразованиям. Если на функциональной схеме слайса имеется путь от соответствующего параметра out_i к in_i, содержащий узлы типа точка, то схема выполнения алгоритма содержит ребро второго типа – передачи потока данных, сопровождаемое преобразованиями, непокрытыми резюме функций. Отношение между параметрами, задаваемое ребрами схемы выполнения алгоритма, не является транзитивным, т. е. если имеется путь от выходного параметра out_{A_1} резюме функции A до входного in_{B_1} резюме B и от выходного параметра out_{B_1} резюме функции B до входного in_{C_1} резюме C, то схема выполнения алгоритма не будет содержать ребро от out_{A_1} к in_{C_1}.

Так как граф функциональной схемы слайса не содержит циклов, то информация о достигающих параметрах резюме функций может быть вычислена в каждом узле графа за один проход по его узлам в топологическом порядке. На рис. 5 показан псевдокод построения схемы выполнения алгоритма, реализованный в виде обхода узлов графа функциональной схемы слайса в топологическом порядке.

```
Input:    S – граф функциональной схемы слайса
Output:   A – схема выполнения алгоритма
BuildAlgorithmExecutionScheme (S):
1   A ← CreateAlgorithmNodes (S);
2   worklist ← TopoSort (S);
3   foreach (n in worklist):
4       match n with :
5           | Point |=>
6               MarkReachedParamsAsModified (n);
7               PropagateInfo( n);
8           | InParam |=>
9               params = GetReachedParams (n)
10              AddEdgeToAlgorithm (A, fromNodes = params, toNode = n);
11              KillallReachedParams (n);
12              PropagateInfo( n);
13          | OutParam|=>
14              KillallReachedParams (n);
15              AddReachedParam( n);
16              PropagateInfo( n);
17          |*|=>  PropagateInfo (n);
```

Рис. 5. Псевдокод построения схемы выполнения алгоритма
Fig. 5. Pseudocode for constructing the algorithm execution diagram

В процессе построения функциональной схемы слайса может возникнуть такая ситуация, при которой имеется поток данных, ведущий в резюме функции в обход ее входных параметров, или поток данных, выходящий из резюме функции и не проходящий через ее выходные параметры. При возникновении подобных ситуаций необходимо создавать *неизвестные* входные или выходные параметры. На резюме функций, содержащие неизвестные параметры, аналитик должен обратить особое внимание. Наличие неизвестных параметров может служить сигналом того, что аналитик некорректно описал резюме функции, пропустив какие-то входные или выходные параметры. Если же аналитику известно, что данная резюме функции описана корректно, то неизвестные параметры являются признаком того, что текущая реализация резюме функции содержит скрытые (недекларированные) потоки данных, и нуждается в дополнительном анализе, так как, вероятно, в ней реализована незадекларированная возможность.

На третьем шаге, заключительном, выполняется анализ схемы выполнения алгоритма и функциональной схемы слайса. В ходе этого анализа локализуются входные параметры извлекаемого алгоритма, исследуются ребра схемы выполнения алгоритма. В первую очередь рассматриваются ребра второго типа, которые описывают передачу потока данных между выходными-входными параметрами, сопровождаемую непокрытыми резюме функций преобразованиями. При необходимости такие преобразования локализуются на функциональной схеме слайса и покрываются резюме функций. То есть, вводятся новые резюме функций и их параметры.

В некоторых случаях после исследования всех ребер схемы выполнения алгоритма аналитик может перейти к анализу только функциональной схемы слайса и попытаться вручную найти на ней важные с точки зрения извлекаемого алгоритма преобразования и покрыть их резюме функций.

Задача построения резюме функций в данной работе не рассматривается. В процессе ее решения используется широкий спектр инструментов, направленный на решение задачи восстановления типов переменных и агрегированных структур данных [29-30], декомпиляции машинного кода в язык высокого уровня [1, 3, 6] и др. В том числе могут применяться подходы и методы, упомянутые в разделе про методы извлечения алгоритмов путем статического анализа бинарного кода данной работы.

В конце третьего шага аналитик принимает решение прекратить итерационное построение высокоуровневого представления алгоритма или выполнить еще одну итерацию, начиная с шага 1. Аналитик прекращает построение алгоритма, если на текущем шаге не было добавлено ни одной новой резюме функции, и ни одно резюме из уже созданных не было изменено. Кроме того, выбранный уровень абстракций полностью удовлетворяет аналитика, и результирующая схема демонстрирует требуемый уровень детализации извлеченного алгоритма.

Рассмотрим построение высокоуровневого представления алгоритма предложенным методом для уже упомянутого примера $archive$. Выполним построение алгоритма формирования результирующего буфера в режиме архивирования. Предварительный шаг подготовки окружения и входных данных анализируемой программы $archive$ для режима архивирования, пропускается, так как не имеет непосредственного отношения к предлагаемому методу. На первом шаге строятся динамический слайс и функциональная схема слайса. Необходимо задать критерий построения слайса в виде точки выполнения программы и начального буфера для отслеживания потока данных. В качестве стартовой точки выполнения будет служить инструкция вызова функции $saveToFile$ (строка 19 на рис. 1), а начального буфера – $bufs[3]$ длиной $resultSize$. Выбор точки выполнения программы обусловлен тем, что на момент вызова функции записи данных извлекаемый алгоритм полностью отработал, результирующий буфер сформирован. Кроме того, информация об используемых соглашениях вызовов функций позволяет легко локализовать

результирующий буфер $buf s[3]$. После выполнения данного шага будет построена функциональная схема слайса, продемонстрировать ее в рамках данной работы не представляется возможным из-за внушительных размеров.

На первой итерации извлечения алгоритма база резюме функций еще пуста, и схема выполнения алгоритма, построенная после выполнения второго шага, будет пустой, поэтому переходим сразу к третьему шагу.

На третьем шаге первой итерации аналитик просматривает функциональную схему слайса от самого нижнего элемента к верхним и для некоторых суперблоков просматривает их содержимое в поисках интересующих его функций. Иерархия вложенности суперблоков соответствует стеку вызовов функций. Наличие информации о состоянии стека вызовов помогает аналитику осуществить выбор уровня абстракции для получения корректного представления алгоритма. К сожалению, этап построения резюме функций невозможно оптимизировать. Однако, аналитик может накопить базу знаний в виде готового набора резюме функций и использовать ее при извлечении новых алгоритмов. Наличие такого набора резюме функций позволяет уже на первой итерации получить схему выполнения алгоритма, а аналитику потребуется лишь уточнить ее новыми резюме на последующих итерациях. Такой подход позволяет существенно повысить продуктивность работы аналитика.

Предположим, что во время анализа функциональной схемы слайса аналитик нашел суперблоки, соответствующие функциям самого верхнего уровня абстракции: $compress$, $decompress$. Выполняя анализ их содержимого, аналитик может создать резюме функции для любого уровня абстракций, что невозможно было бы сделать при построении алгоритма методом статического анализа кода. Пусть аналитиком были добавлены следующие резюме функций: 1) $fgets$ – функция чтения данных из файла; 2) $strncpy$ – функция копирования подстроки длины n; 3) $ZSTD\_compress$ – функция архивирования данных из библиотеки zstd; 4) $compress$ – функция архивирования данных из библиотеки zlib; 5) $fwrite$ – функция записи буфера в файл. Предположим, что аналитику не важна конкретная реализация, а он пытался выбрать максимально высокий уровень абстракции, на котором появляются буферные параметры, содержащие исходные данные и результат архивирования. Функции $compress$, $ZSTD\_compress$ вызываются во время выполнения программы при архивировании. Данные функции неизвестны при статическом анализе кода.

При создании резюме функций аналитик описал следующие входные и выходные параметры: 1) $fgets$ – выходной параметр buf; 2) $ZSTD\_compress$ – входные параметры: src (буфер с данными для архивирования); выходные: $dest$ (результат архивирования); 3) $compress$ – входной параметр src, выходной $dest$.

После создания резюме функций выполняется вторая, финальная, итерация построения высокоуровневого представления алгоритма. На второй итерации будет построена схема выполнения алгоритма, показанная на рис. 6. При описании резюме функций авторы специально пропустили некоторые входные/выходные параметры, чтобы рассматриваемый пример был более компактным. В реальности же приведенные резюме функций обладают большим числом параметров, однако на метод извлечения алгоритма и построения его высокоуровневого представления это никак не влияет.

Подводя итог, отмечаем, что предложенный метод извлечения алгоритма и построения его высокоуровневого представления лишен недостатков, связанных с отсутствием информации времени выполнения. Извлеченный алгоритм в явном виде содержит параметры, соответствующие выбранному уровню абстракции. Выбор уровня абстракций осуществляется при помощи описания резюме функций. Резюме функций не только задают уровень абстракции, но и уточняют направления потоков данных при помощи указания зависимостей между входными и выходными параметрами. Кроме того, корректные резюме функций помогают обнаружить в алгоритме скрытые потоки данных. Итеративный способ

извлечения алгоритма позволяет построить алгоритм с требуемым уровнем детализации, который может быть использован как для ручного анализа аналитиком, так и для автоматического/полуавтоматического анализа. Также имеется возможность по извлеченному алгоритму построить реализацию прототипа алгоритма. Данный метод не лишен и недостатков, а именно, схема выполнения алгоритма соответствует одной, конкретной траектории выполнения анализируемой программы. Об остальных возможных путях выполнения и состояниях программы в рамках построенной схемы выполнения ничего неизвестно. Отдельной задачей является объединение функциональных схем слайсов и схем выполнения алгоритма, полученных из разных траекторий выполнения программы, т. е. выполнявшихся на разных наборах данных. Однако это направление нуждается в тщательном исследовании.

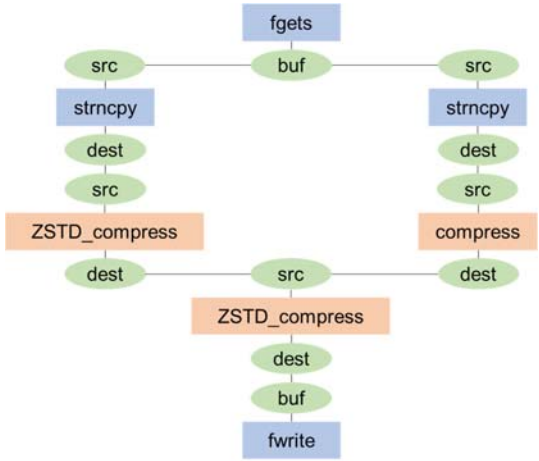


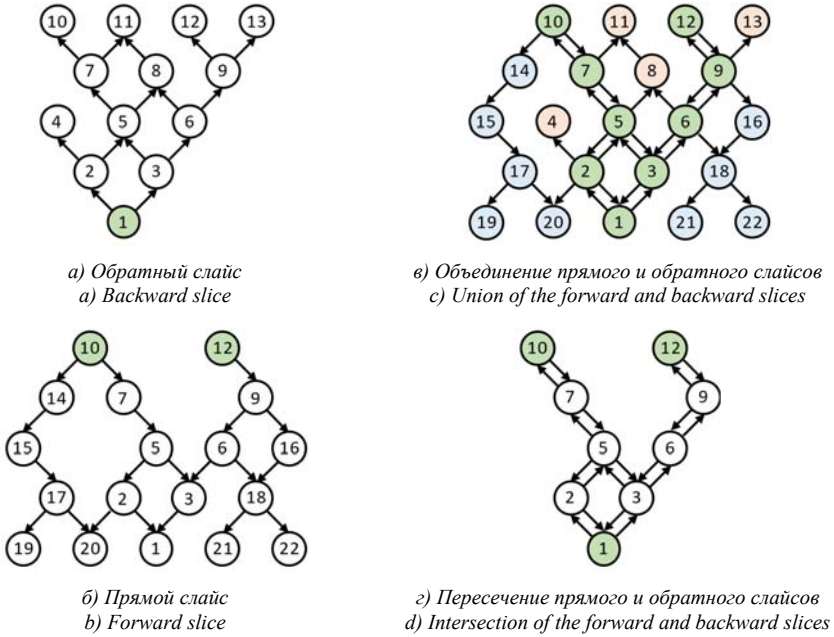
Рис. 6. Схема выполнения алгоритма ‘archive’ в режиме архивирования
Fig. 6. Algorithm execution diagram of the ‘archive’ program in compression mode

3. Сокращение размера функциональной схемы слайса

Анализ функциональной схемы слайса, полученной в результате построения обратного слайса, в ручном режиме осложнен ее значительными размерами. Не все инструкции, попавшие в обратный слайс (а значит и в функциональную схему слайса) являются значимыми в контексте извлекаемого алгоритма. Как уже было сказано в предыдущем разделе, часть инструкций попадают в обратный слайс в результате ложных зависимостей. Разработка методов сокращения числа избыточных инструкций является отдельным перспективным направлением научных исследований. В текущем разделе предлагается метод сокращения размера функциональной схемы слайса путем выполнения операции пересечения схем, построенных в прямом и обратном направлениях.

Функциональная схема слайса строится на основе полученного обратного слайса. В результате выполнения второго шага извлечения и построения алгоритма генерируется схема выполнения алгоритма (промежуточная или итоговая). Полученная схема позволяет локализовать входные параметры алгоритма, то есть найти стояния и соответствующие точки выполнения программы, на которых параметры появляются, а также описать их в виде специальных выражений. Основная идея предлагаемого подхода заключается в построении функциональных схем слайса в прямом направлении относительно входных параметров и вычислении операции пересечения построенных функциональных схем со схемой, построенной как результат обратного слайса относительно выходного параметра алгоритма.

На рис. 7 представлен пример выполнения операций объединения и пересечения над прямым и обратным слайсами. Вершинами графа являются инструкции, попавшие в прямой или обратный слайс, а ребра задают направления потоков данных между ними. Вершина с номером 1 представляет инструкцию, которая использовалась в качестве критерия построения обратного слайса. После того как была построена функциональная схема слайса по обратному слайсу и схема выполнения алгоритма, необходимо локализовать входные параметры извлекаемого алгоритма. Предположим, что входные параметры были локализованы в вершинах 10 и 12 обратного слайса. Это значит, что входные операнды этих инструкций являются входными параметрами извлекаемого алгоритма. Теперь необходимо построить прямой слайс относительно входных параметров алгоритма, в качестве критерия построения слайса будут выступать инструкции в вершинах 10 и 12 (на рис. 7б отмечены зеленым цветом на графе прямого слайса). Результат построения прямого слайса также показан на рис. 7б. На объединении прямого и обратного слайсов (рис. 7в) можно увидеть, что во время прямого слайса были добавлены инструкции 14-22 (помеченные синим цветом), а также инструкции, вошедшие в обратный слайс, но не вошедшие в прямой (вершины 4, 8, 11, 13, помеченные розовым цветом). После выполнения операции пересечения прямого и обратного слайсов в результирующем слайсе останутся только те инструкции, которые содержатся и в прямом, и в обратном слайсах. Результат пересечения показан на рис. 7г. Как можно увидеть, таким способом удалось исключить часть инструкций, не являющихся значимыми с точки зрения извлекаемого алгоритма.



а) Обратный слайс
a) Backward slice

в) Объединение прямого и обратного слайсов
c) Union of the forward and backward slices

б) Прямой слайс
b) Forward slice

г) Пересечение прямого и обратного слайсов
d) Intersection of the forward and backward slices

Рис. 7. Операции объединения и пересечения над прямым и обратным слайсами
Fig. 7. Union and Intersection operations of the forward and backward slices

4. Заключение

Извлечение алгоритмов и построение их высокоуровневого представления может быть выполнено как методами статического анализа кода, так и динамического. При

восстановлении алгоритма путем статического анализа кода приходится сталкиваться с рядом проблем и ограничений, связанных в первую очередь с отсутствием информации времени выполнения. Часть таких ограничений удается преодолеть, например при помощи абстрактной интерпретации, но некоторые проблемы, скажем, невозможность выбора при построении алгоритма уровня абстракции, могут остаться неразрешимыми. В результате извлеченный алгоритм может оказаться некорректным или вовсе его полное извлечение может оказаться невозможным.

Построение алгоритмов методами динамического анализа кода позволяет избежать таких проблем. Поэтому в работе предложен метод извлечения алгоритмов из бинарного кода на основе анализа динамического слайса. Данный метод представляет собой итеративное построение алгоритма, на каждой итерации данного процесса может быть получено представление алгоритма двух уровней – функциональная схема слайса и схема выполнения алгоритма. В работе также предложен способ построения схемы выполнения алгоритма по функциональной схеме слайса.

Построенные предложенными методами функциональная схема слайса и схема выполнения алгоритма могут быть использованы для проведения анализа по самым разнообразным сценариям как в ручном, так и в автоматическом/полуавтоматическом режимах.

Одним из недостатков предложенного метода является наличие избыточных инструкций в извлеченном алгоритме. Предложенный авторами подход к сокращению числа избыточных инструкций выполняет пересечение функциональных схем слайсов, построенных в прямом и обратном направлениях.

Извлеченный алгоритм и построенная схема выполнения алгоритма отражают выполнение анализируемой программы только вдоль одного пути, что несомненно является недостатком предложенного метода. Задача объединения восстановленных алгоритмов по динамическим слайсам, отражающих выполнение программы вдоль разных путей, является перспективным направлением дальнейших работ. Еще одним перспективным направлением работ является объединение методов статического и динамического анализа кода для решения задачи извлечения алгоритмов и построения их высокоуровневого представления. Интерес представляет подход, выполняющий частичное вычисление для получения остаточной, специализированной программы. Однако способы его использования совместно с динамическим анализом кода для решения задачи извлечения алгоритмов сложных программ нуждаются в тщательном исследовании. Важным также является еще одно перспективное направление исследований – разработка инструментария для сокращения размера слайса путем исключения из него избыточных инструкций с незначимыми зависимостями.

Список литературы / References

[1]. D. Brumley, I. Jager, T. Avgerinos, E. J. Schwartz. BAP: A Binary Analysis Platform. In Proceedings of the 23rd International Conference on Computer Aided Verification. Snowbird, UT, 2011.
[2]. Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel,G. Vigna. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. IEEE Symposium on Security and Privacy, 2016.
[3]. The IDA Pro disassembler and debugger. [Online]. http://www.hex-rays.com/idapro/.
[4]. NSA, Ghidra is a software reverse engineering (SRE) framework. NSA. [Online]. https://github.com/NationalSecurityAgency/ghidra.
[5]. The Official Rizin Book. [Online]. https://book.rizin.re/.
[6]. ESIL: Radare2 book. [Online]. https://radare.gitbooks.io/radare2book/content/disassembling/esil.html.
[7]. Андрей Тихонов, Арутюн Аветисян, Вартан Падарян. Методика извлечения алгоритма из бинарного кода на основе динамического анализа. // Проблемы информационной безопасности. Компьютерные системы. №3 2008. стр. 66-71.
[8]. Retargetable Decompiler. [Online] https://retdec.com/.

[9]. Alessandro Di Federico, Mathias Payer, Giovanni Agosta. Rev.ng: a unified binary analysis framework to recover CFGs and function boundaries. In Proceedings of the 26th International Conference on Compiler Construction (CC 2017). Association for Computing Machinery, New York, NY, USA, pp. 131–141.
[10]. Reps T., Balakrishnan G. Improved memory-access analysis for x86 executables // Proceedings of the International Conference on Compiler Construction, April 2008.
[11]. Balakrishnan G., Reps T. DIVINE: DIscovering Variables IN Executables // Proceedings of the VMCAI. Nice, France. Jan. 14-16, 2007.
[12]. M. Weiser. Program Slicing. In Int. Conf. on Softw. Eng. IEEE Comp. Soc., Wash., DC, pp. 439–449, 1981
[13]. Michael Vaughn, Thomas Reps. A Generating-Extension-Generator for Machine Code. 2020.
[14]. Venkatesh Srinivasan and Thomas Reps. Partial Evaluation of Machine Code. SIGPLAN Not. 50, 10 (Oct. 2015), pp. 860–879.
[15]. Venkatesh Srinivasan, Thomas Reps. An improved algorithm for slicing machine code. In Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2016). Association for Computing Machinery, New York, NY, USA, 378–393.
[16]. N. Jones, C. Gomard, P. Sestoft. Partial Evaluation and Automatic Program Generation. Prentice-Hall, Inc., 1993.
[17]. B. Yadegari, B. Johannesmeyer, B. Whitely, S. Debray. A generic approach to automatic deobfuscation of executable code. In S&P, 2015.
[18]. K. Coogan, G. Lu, S. Debray. Deobfuscation of virtualizationobfuscated software: A semantics-based approach. In CCS, 2011.
[19]. S. Bansal, A. Aiken. Automatic generation of peephole superoptimizers. In ASPLOS, 2006.
[20]. E. Schkufza, R. Sharma, A. Aiken. Stochastic superoptimization. In ASPLOS, 2013.
[21]. M. Abadi, M. Budiu, U. Erlingsson, J. Ligatti. Control-flow integrity. In CCS, 2005.
[22]. U. Erlingsson, F. Schneider. SASI enforcement of security policies: A retrospective. In Workshop on New Security Paradigms, 1999.
[23]. A. Slowinska, T. Stancescu, H. Bos. Body armor for binaries: Preventing buffer overflows without recompilation. In ATC, 2012.
[24]. L. O. Andersen. Program Analysis and Specialization for the C Programming Language. PhD thesis, Univ. of Copenhagen, 1994.
[25]. C. Consel, L. Hornof, F. No¨el, J. Noy´e, and N. Volanschi. A uniform approach for compile-time and run-time specialization. Dagstuhl Seminar on Partial Evaluation, pages 54–72, 1996.
[26]. P. Kleinrubatscher, A. Kriegshaber, R. Z¨ochling, and R. Gl¨uck. Fortran program specialization. SIGPLAN Notices, 30(4), 1995.
[27]. П.М. Довгалюк, В.А. Макаров, В.А. Падарян, М.С. Романеев, Н.И. Фурсова. Применение программных эмуляторов в задачах анализа бинарного кода. Труды Института системного программирования РАН, том 26, вып. 1, 2014, стр. 277-296. DOI: 10.15514/ISPRAS-2014-26(1)-9.
[28]. A. B. Bugerya, I. I. Kulagin, V. A. Padaryan, M. A. Solovev, A. Y. Tikhonov, Recovery of High-Level Intermediate Representations of Algorithms from Binary Code," 2019 Ivannikov Memorial Workshop (IVMEM), 2019, pp. 57-63.
[29]. Z. Lin, X. Zhang, D. Xu. Automatic Reverse Engineering of Data Structures from Binary Execution. In Proceedings of the 11th Annual Information Security Symposium, West Lafayette, Indiana, 2010.
[30]. G. Ramalingam, J. Field, F. Tip. Aggregate Structure Identification and Its Application to Program Analysis. In Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, Texas, USA, 1999.

Информация об авторах / Information about authors

Иван Иванович КУЛАГИН – кандидат технических наук, научный сотрудник ИСП РАН. Область научных интересов: построение компиляторов, анализ бинарного кода, оптимизирующие компиляторы, полиэдральная компиляция, генерация кода, модели параллельного программирования.

Ivan Ivanovich KULAGIN – Cand. Sci. (Tech.), Researcher in ISP RAS. Research interests: compiler construction, binary code analysis, compiler optimizations, polyhedral compilation, code generation, parallel programming models.

Вартан Андроникович ПАДАРЯН – кандидат физико-математических наук, ведущий научный сотрудник ИСП РАН, доцент кафедры Системного программирования ВМК МГУ. Сфера научных интересов: компиляторные технологии, анализ бинарного кода, компьютерная безопасность, высокопроизводительные вычисления.

Vartan Andronikovich PADARYAN – Cand. Sci. (Phys.-Math.), leading researcher at ISP RAS, associate professor of the Department of system programming of CMC of Lomonosov Moscow State University. Science Interests: compiler technologies, binary code analysis, cybersecurity, high performance calculating.

Вячеслав Александрович КОШКИН – стажер-исследователь ИСП РАН. Сфера научных интересов: динамический анализ бинарного кода, восстановление высокоуровневого представления алгоритма, анализ помеченных данных, поиск утечек чувствительных данных, восстановление типов структур данных.

Viacheslav Alexandrovich KOSHKIN – research intern at ISP RAS. Science Interests: dynamic binary code analysis, high-level algorithm representation recovery, taint analysis, sensitive data leak detection, data structures type recovery.