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 технологической 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 C-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 C-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.