ИНФОРМАТИКА

2014 **январь-март** **№ 1**

УДК 519.681.3/004.41

В.В. Сергейчик, А.А. Иванюк

ОСОБЕННОСТИ ОБФУСКАЦИИ VHDL-ОПИСАНИЙ И МЕТОДЫ ОЦЕНКИ ЕЕ СЛОЖНОСТИ

Формализуется лексическая и функциональная обфускация. Обфускация – совокупность методик и средств, направленных на затруднение анализа исходных кодов программ. Дается краткий обзор способов лексической обфускации и изучаются их недостатки применительно к описаниям на языке VHDL. Рассматриваются методы оценки сложности описаний на языке VHDL. Приводится оценка сложности для различных вариантов VHDL-описаний одного цифрового устройства.

Введение

В настоящее время быстрыми темпами растет объем производства цифровых устройств, и в связи с этим особую остроту приобретает проблема нарушения авторских прав. По некоторым данным, ущерб от пиратства и других угроз в области производства аппаратного обеспечения составляет около 1 млрд долл. в день [1], что примерно в 10 раз превышает ущерб от пиратства в области ПО [2]. Кроме пиратства, появляются и новые виды угроз. Например, известны атаки на аппаратные реализации криптографических алгоритмов (так называемые Side-Channel Attacks [3]). В ходе этих атак осуществляется измерение токов и магнитных полей в некоторых частях схемы в моменты переключения значений, что позволяет получать значения секретных ключей. Также существует угроза внедрения аппаратных троянов, которые могут изменять функционирование устройства, снижать уровень его защиты, нарушать работоспособность и передавать секретную информацию из него. Поэтому сейчас огромное значение имеет разработка методов защиты от подобных угроз и борьбы с пиратством. Одним из таких методов является обфускация.

Обфускация (от англ. *obfuscate* – делать неочевидным, запутанным, сбивать с толку) – широко известная методика защиты исходных кодов программ от обратного проектирования [4]. Основной целью обфускации является затруднение понимания функционирования программы. В идеале сложность и временные затраты на обратное проектирование должны оказаться близки к затратам на разработку системы с нуля. Обфускация очень часто используется совместно с другими методами защиты от пиратства; например, ее применяют для сокрытия водяных знаков и отпечатков пальцев. Следует отметить, что обфускация используется также для защиты внедренного вредоносного кода (malware) от сканеров и при разработке вирусов.

Существует большое число методов обфускации, разработанных для различных языков программирования. Однако эти методы теряют свою актуальность в случае языка VHDL, так как результаты их применения не приводят к изменению конечного результата синтеза, т. е. схемы устройств до и после обфускации выглядят одинаково. Поэтому применительно к языку VHDL следует рассмотреть еще одну разновидность обфускации – функциональную [5]. Ее суть состоит в получении эквивалентной схемы.

1. Лексическая обфускация

Попробуем формализовать лексическую обфускацию для языка VHDL. Для этого представим исходный язык описания как

$$<V> = \{T; S; D; K\},$$

где T – множество терминалов; S – множество выражений; D – множество объявлений; K – множество комментариев.

Схему цифрового устройства *Sch* можно описать следующим образом [5]:

$$Sch = \{IP; OP; B; L\},$$

где IP – множество входных портов; OP – множество выходных портов; B – множество функциональных блоков, из которых состоит схема устройства; L – множество проводящих линий, соединяющих внутренние блоки и порты.

Синтез – процесс интерпретации исходного описания на языке V в схему:

$$DD(V) = DD(\{T; S; D; K\}) = \{IP; OP; B; L\} = Sch.$$

Тогда лексическое эквивалентное преобразование – это замена одного фрагмента кода $V_1 = \{T; S; D; K\}$, результат синтеза которого

$$DD(V_1) = DD(\{T; S; D; K\}) = \{IP; OP; B; L\} = Sch_1,$$

другим фрагментом

$$V_2 = \{T'; S'; D'; K'\},$$

где $V_1 \neq V_2$, результат синтеза которого идентичен предыдущему:

$$DD(V_2) = DD(\{T'; S'; D'; K'\}) = \{IP; OP; B; L\} = Sch_1 = DD(V_1).$$

Сложность описания можно представить как функцию, зависящую от внутренней структуры V:

$$C(V) = C(\{T; S; D; K\}).$$

Лексическое обфусцирующее преобразование – это лексическое эквивалентное преобразование, для которого дополнительно выполняется свойство «сложность результирующего фрагмента V_2 больше, чем сложность исходного V_1»:

$$C(V_2) > C(V_1),\ DD(V_2) = DD(V_1).$$

Лексическая обфускация – процесс применения лексических обфусцирующих преобразований к исходному описанию V с целью получения более сложного V^*, но при сохранении неизменным результата синтеза.

Частными случаями лексических обфусцирующих преобразований будут следующие:

1) преобразования, удаляющие или добавляющие избыточные конструкции в описание:

$$V' = V \pm V_r,$$

где V_r – избыточные конструкции; V – исходное описание; V' – результирующее описание.

При этом $DD(V') = DD(V \pm V_r) = DD(V) \pm DD(V_r) = Sch \pm \varnothing = Sch$. Примером таких преобразований будет внедрение комбинаций сигналов, минимизируемых при синтезе; удаление комментариев; разрушение форматирования; переименование идентификаторов.

2) преобразования, переупорядочивающие операции и не оказывающие влияние на семантику:

$$V = \{p_1, \ldots, p_n\};$$
$$V' = permutation(\{p_1, \ldots, p_n\});$$
$$DD(V) = DD(V') = Sch.$$

Примером служат преобразования переупорядочивания параллельных выражений.

2. Функциональная обфускация

Функциональность устройства Sch можно определить как зависимость значений на выходных портах OP от значений на входных портах IP, внутренних блоков B и проводящих линий L [5]:

$$OP = F_{Sch}(IP; B; L).$$

Пусть существует другое устройство Sch', схема которого описывается выражением

$$Sch' = \{IP; OP; B'; L'\},$$

при этом множество блоков B' и множество линий L' не совпадают с аналогичными множествами устройства *Sch*.

Устройства *Sch* и *Sch*' будут функционально эквивалентными, если выполняется следующее равенство [5]:

$$F_{Sch}(IP; B; L) = F_{Sch'}(IP; B'; L').$$

Функциональное эквивалентное преобразование – это замена одного фрагмента кода $V_1 = \{T; S; D; K\}$, результат синтеза которого

$$DD(V_1) = DD(\{T; S; D; K\}) = \{IP; OP; B; L\} = Sch_1,$$

другим фрагментом $V_2 = \{T'; S'; D'; K'\}$, результат синтеза которого

$$DD(V_2) = DD(\{T'; S'; D'; K'\}) = \{IP; OP; B'; L'\} = Sch_2.$$

При этом Sch_2 не совпадает с Sch_1. Однако наблюдаемое поведение [4] обеих схем идентично:

$$F_{Sch1}(IP; B; L) = F_{Sch2}(IP; B'; L').$$

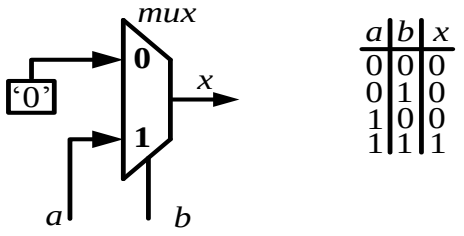
Тогда преобразование функциональной обфускации – это функциональное эквивалентное преобразование, для которого дополнительно выполняется свойство «сложность результирующей схемы Sch_2 выше, чем сложность исходной Sch_1»:

$$C_{Sch1}(IP; OP; B; L) < C_{Sch2}(IP; OP; B'; L').$$

Функциональная обфускация – процесс применения функциональных обфусцирующих преобразований к схеме *Sch* с целью получения более сложной для понимания схемы *Sch**, имеющей эквивалентную функциональность:

$$Obfusc(Sch) = Sch^*, F_{Sch}(IP; B; L) = F_{Sch^*}(IP; B^*; L^*).$$

Примером функциональной обфускации может быть мультиплексор, первый вход которого подключен к константе 0 (рис. 1). Такая схема эквивалентна вентилю and. С помощью некоторых ухищрений можно добиться того, что синтезатор не распознает в данной схеме and.



a	b	x
0	0	0
0	1	0
1	0	0
1	1	1

Рис. 1. Обфусцированная схема вентиля and

Возникает необходимость в оценке качества проводимых функциональных преобразований. В связи с этим можно обратиться к методике [6], которая учитывает вносимые преобразованиями издержки в результирующую схему: увеличение использования аппаратуры и уменьшение скорости работы. Кроме того, методика позволяет оценить сложность обфусцированного описания. Однако в случае функциональных преобразований необходимо также оценивать не только сложность описания, но и сложность обфусцированной схемы по сравнению с исходной.

Понятие «сложность для понимания» подразумевает наличие человека и поэтому является достаточно субъективным и размытым. В связи с этим можно использовать подходы для оценки сложности логических цепей [7]: число вентилей в цепи, число вентилей на самом длинном пути. Очевидным недостатком упомянутых подходов является то, что они не учитывают число прово-

дящих линий и число соединений между блоками; кроме того, они не приспособлены для последовательностных цепей. Эти методы также не учитывают разную квалификацию специалистов: одна и та же схема может иметь разную сложность для разных людей. Для простоты можно определить сложность схемы *Sch* для понимания как некоторую функцию $C_{Sch}(IP; OP; B; L)$, подразумевая, что сложность определяется только внутренним устройством схемы.

3. Обзор существующих методов обфускации

В настоящее время разработано большое количество методов лексической обфускации. Оригинальная классификация описана в работе [4]. Ниже приводятся основные группы преобразований с примерами согласно этой классификации: преобразования размещения, преобразования потока управления, преобразования данных. Исследуется также их применимость для VHDL.

1. Преобразования размещения.

Удаление комментариев.

Переименование идентификаторов – замена имен в программе на последовательности символов, не несущие смысловой нагрузки. Для усложнения задачи анализа кода человеком имеет смысл генерировать длинные имена, близкие по написанию [5]. Этого можно достичь, используя похожие символы: I, l, 1, o, 0. Пример имен: oo01ioool1o001oioii1ol00o101ol001 и oo01ioool1o001oioillol00o101ol001.

Разрушение форматирования – удаление отступов, табуляции, разграничений смысловых частей.

Данные преобразования являются достаточно слабыми даже для обычных языков программирования. Их можно считать предельным случаем плохого стиля кода.

2. Преобразования потока управления.

Преобразования агрегирования.

Преобразования функций направлены на разрушение абстракций, создаваемых функциями, а также на порождение ложных абстракций. Примеры: встраивание функций, выделение функций, объединение функций.

Преобразования переупорядочивания циклов применяются для оптимизации в компиляторах. Примеры: разворачивание циклов, расщепление циклов, объединение циклов.

Приведенные преобразования агрегирования обладают достаточно малой стойкостью.

Преобразования упорядочивания.

Переупорядочивание выражений направлено на разрушение локальности расположения программистом связанных по смыслу операций. Преобразование обладает малой силой.

Вычислительные преобразования.

Преобразование потока управления из сократимого в несократимый основано на различии в описательной силе исходного языка программирования и представления, в которое он компилируется. Например, в java нет оператора goto, а в байт-коде он есть. Поэтому возможно представление с помощью байт-кода потока управления, который описывается на java на другом уровне абстракции, что создает трудности для деобфускаторов при попытке воссоздания исходного кода [4, 8].

Интерпретация таблиц – замена части кода исходного языка на код для виртуальной машины другого языка. Преобразование обладает высокой стойкостью, но требует больших затрат времени выполнения. Рекомендуется использовать его только для самых ценных частей кода.

3. Преобразования данных.

Изменение кодирования. Примером может служить замена счетчика цикла *i* на выражение $i' = c_1 + i*c_2$, где c_1, c_2 – некоторые константы. Преобразование обладает низкой стойкостью, легко удаляется оптимизирующим компилятором (это так называемые аффинные преобразования индексов [9]).

Преобразование статических данных в процедурные заменяет строки автоматами, генерирующими такие строки.

Объединение скалярных переменных. Суть данного преобразования заключается в замене нескольких переменных $var_1 \ldots var_n$ на одну – *var*, которая включает объединенные диапазоны $var_1 \ldots var_n$.

Изменение структуры массивов разрушает абстракции, создаваемые массивами. Примеры: разделение массива на несколько независимых массивов, объединение двух и более массивов в один, сворачивание массива – увеличение числа измерений массива, сплющивание массива – уменьшение числа измерений массива. Данные преобразования используются в компиляторах для оптимизаций [9], преобразования обладают низкой стойкостью.

Прочие преобразования.

Динамическая мутация кода [10] – основана на замене кода обфусцируемой процедуры на вызов шаблона процедуры. В шаблоне часть инструкций заменена на бессмысленный набор байтов; кроме того, присутствует вызов редактирующего механизма, который конструирует процедуру из данного шаблона и последовательности байтов инструкций исходной процедуры. Затем происходит передача управления этой сконструированной процедуре. Метод обладает очень высокой стойкостью, так как процесс конструирования происходит во время выполнения программы.

Внедрение сигналов, минимизируемых при синтезе. Представляет собой специфичное для HDL-языков преобразование. Суть этого лексического преобразования заключается во внедрении в описание комбинаций сигналов, которые будут заведомо удалены при синтезе. Преобразование можно считать разновидностью метода введения избыточного кода (*Dead Code Insertion*). Данная трансформация реализуется следующим образом: сначала выбирается выражение, всегда вычисляемое в константу. Например:

$$(A + \bar{A}) * (B + \bar{B}) = \text{‘1’} \Rightarrow A * B + \bar{A} * B + A * \bar{B} + \bar{A} * \bar{B} = \text{‘1’},$$

где A и B – произвольные сигналы.

Выражение всегда принимает значение логической единицы. После раскрытия скобок получается выражение в правой части. Порядок дальнейшего вычисления показан на рис. 2.

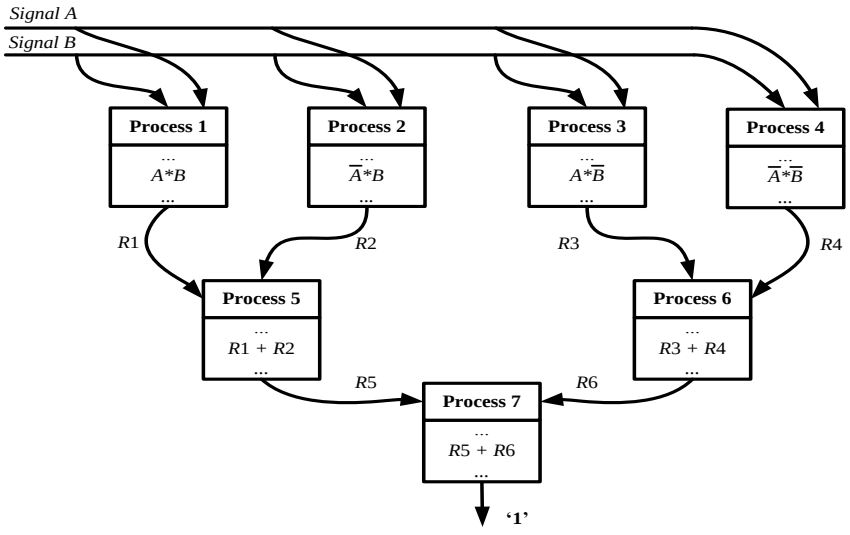


Рис. 2. Порядок вычисления значений методом минимизируемых сигналов

Каждое слагаемое выносится в отдельный процесс и вычисляется в нем, промежуточные результаты вычисления (сигналы $R_1 – R_6$) попарно складываются в других процессах. Для реализации преобразования в виде, показанном на рис. 2, необходимы $2^{x+1}–1$ процессов, где x – число независимых сигналов (A, B в данном примере), хотя возможны и другие реализации. Чтобы преобразование не слишком отличалось от остального кода, имеет смысл для вычислений использовать уже существующие в описании процессы. Это также позволит снизить связность отдельных процессов и увеличить сцепление между ними, что в итоге затруднит понимание.

Все приведенные преобразования в случае языка VHDL обладают низкой стойкостью. Причина этого заключается в том, что результат синтеза (схема) до и после обфускации остается неизменным. Именно поэтому простейшим способом атаки на лексические методы будет

синтез. Однако некоторую ценность эти методы все же представляют. Они ставят первый барьер на пути понимания системы, разрушая исходные высокоуровневые абстракции, известные идиомы, скрывая назначение кода за различными ложными операциями и абстракциями. Кроме того, обфускация добавляет дополнительную маскировку для других методов защиты, например для водяных знаков и отпечатков пальцев. В конечном итоге лексические методы приводят к необходимости привлечения более квалифицированных специалистов, увеличению времени, необходимого для обратного проектирования, а следовательно, снижают экономическую эффективность последнего.

4. Пример лексической обфускации

Рассмотрим пример мультиплексора. Существует много вариантов его реализации на VHDL (рис. 3). Описания 1 – 4 отражают поведение устройства. Они являются эквивалентными, однако трудно выделить среди них наиболее сложное. Поэтому в данном случае нельзя сказать, что замена одного из описаний на другое в проектном коде приведет к лексической обфускации.

Самым низкоуровневым будет представление в виде логических элементов (описание 5). Такое описание немного труднее для восприятия, что связано с раскрытием низкоуровневых деталей реализации и разрушением абстракции переключателя. По этой причине его можно считать не просто очередным эквивалентным описанием, а лексической обфускацией описания.

Кроме того, мультиплексор можно представить в виде элемента памяти, адресуемого входными сигналами (описание 6). Данное описание сложнее для понимания, чем предыдущие. Его можно считать примером лексической обфускации, так как в нем создается фальшивая абстракция ROM и осуществляется работа с этой абстракцией.

Возможность реализации функций вместе с возможностью их перегрузки (в том числе и операторов, таких как «+», «–», «*») позволяет создавать сбивающие с толку абстракции, которые «подражают» знакомым и хорошо известным, но отличаются поведением. Иллюстрацией служит описание 7.

1)
```
process (a, b)
begin
    if (sel = '1') then
        x <= a;
    else
        x <= b;
    end if;
end process;
```

2)
```
process (a, b)
begin
    case sel is
        when '1' => x <= a;
        when others => x <= b;
    end case;
end process;
```

3)
```
x <= a when sel = '1' else b;
```

4)
```
with sel select
    x <= a when '1',
    b when others;
```

5)
```
x <= (sel and a) or (not sel and b);
```

6)
```
process(a,b,sel)
    variable index : integer;
    variable temp : std_logic_vector(3 downto 1);
    variable rom : std_logic_vector (7 downto 0) := "11001010";
begin
    temp := sel & a & b;
    index := conv_integer(temp);
    x <= rom(index);
end process;
```

7)
```
process (a, b, sel)
   subtype pseudo_std_logic is std_logic range 'X' to '-';
    FUNCTION "+" (v : std_logic_vector; selector : pseudo_std_logic)
    RETURN std_logic IS
        variable res : std_logic;
    BEGIN
        if (selector = '1') then
            res := v(v'high);
        else
            res := v(v'low);
        end if;
        return res;
    END "+";
begin
    x <= (a&b) + sel;
end process;
```

Рис. 3. Некоторые варианты описаний мультиплексора

В описании 7 используется фальшивая абстракция операции сложения. Кроме того, здесь задействовано преобразование агрегирования (объединение переменных в вектор).

Результат применения лексических обфусцирующих преобразований размещения к описанию 7 показан на рис. 4. Из рисунка видно, что переименование идентификаторов и разрушение форматирования усложняют процесс анализа кода.

```
process (o10ll01l0oo0o0oi0o10o0oooi0iillll0,o10ll01l0oo0o0oio0i0o0oooi0iillll0
,o10ll01l0oo0o0oiooloo0oooi0iillll0)subtype o10ll01l1oo0o0oio0i0o0oooi0iillll0
to '-';function "+"(o10ll01l0oo0o0oiooiooo0ooi0iillll0:     is std_logic range 'X'
in std_logic_vector ;o10ll01l0oo0o0oiooioo0ooi0iillll0:in
o10ll01l1oo0o0oio0i0o0oooi0iillll0)return std_logic is Variable
o10ll01l0oo0o0oiooioo000oi0iillll0:std_logic;begin if
(o10ll01l0oo0o0oiooioo0oooi0iillll0='1')then o10ll01l0oo0o0oiooioo000oi0iillll0
:=o10ll01l0oo0o0oiooioooo0ooi0iillll0(o10ll01l0oo0o0oiooioooo0ooi0iillll0 'high);
else o10ll01l0oo0o0oiooioo000oi0iillll0:=o10ll01l0oo0o0oiooioooo0ooi0iillll0
(o10ll01l0oo0o0oiooioooo0ooi0iillll0'low);end if;return
o10ll01l0oo0o0oiooioo000oi0iillll0;end;begin o10ll01l0oo0o0oiooi0o0oooi0iillll0
<=(o10ll01l0oo0o0oi0o10o0oooi0iillll0&o10ll01l0oo0o0oio0i0o0oooi0iillll0)
+o10ll01l0oo0o0oiooloo0oooi0iillll0;end process ;
```

Рис. 4. Результат лексической обфускации описания 7

Результат лексической обфускации описания 1 (рис. 5) в дальнейшем будем рассматривать как описание 8. В нем использованы преобразования разрушения форматирования, перестановки параллельных операторов, переименования идентификаторов и внедрения комбинаций сигналов, минимизируемых при синтезе. Для осуществления последнего преобразования дополнительно введены еще девять сигналов и шесть процессов, взаимодействия между которыми также придется изучать, чтобы понять код. Часть действий с исчезающими сигналами производится внутри процесса, существовавшего в исходном описании, за счет внедрения в него дополнительных выражений.

```
process(ol0iol01l1ooli000i0011o0111lll11i,ol0iol0lll0oli000i0011o0111lll11i,ol0iol01i10oli000i0011o0111lll11i)begin
ol0iol01l1ooli000i0011o0111lll11i<=(not ol0iol0lll0oli000i0011o0111lll11i)AND ol0iol01i10oli000i0011o0111lll11i
;end process;process(ol0iol01l1o0ii000i0011o0111lll11i,ol0iol0lll0oli000i0011o0111lll11i,ol0iol01i10oli000i0011o0111lll11i)begin
ol0iol01l1o0ii000i0011o0111lll11i<=(ol0iol0lll0oli000i0011o0111lll11i)AND ol0iol01i10oli000i0011o0111lll11i;end process
;process(ol0iol01l1o0lioooi0011o0111lll11i,ol0iol01l1o0llo00i0011o0111lll11i,
ol0iol01l1o0li000i0011o0111lll11i)begin ol0iol01l1o0lioooi0011o0111lll11i<=(ol0iol01l1o0llo00i0011o0111lll11i)
OR ol0iol01l1o0li000i0011o0111lll11i;end process;process(ol0iol01l100li000i0011o0111lll11i,ol0iol0lll0oli000i0011o0111lll11i,
ol0iol01i10oli000i0011o0111lll11i)begin ol0iol01l100li000i0011o0111lll11i<=(
ol0iol0lll0oli000i0011o0111lll11i)AND not ol0iol01i10oli000i0011o0111lll11i;end process;process(ol0iol01l1o0li000i0011o0111lll11i
,ol0iol01l1ooli000i0011o0111lll11i,ol0iol01l1o0ii000i0011o0111lll11i)begin ol0iol01l1o0li000i0011o0111lll11i<=
(ol0iol01l1ooli000i0011o0111lll11i)OR ol0iol01l1o0ii000i0011o0111lll11i;end process;
process(ol0iol01l1o0lio0oio011o0111lll11i,ol0iol01l1o0lio0oi0o11o0111lll11i,ol0iol01l1o0lio0oi00ilo0111lll11i,
ol0iol01liooli000i0011o0111lll11i,ol0iol0lll0oli000i0011o0111lll11i
ol0iol01l1o0lio0oi0011o0111lll11i      ,ol0iol01i10oli000i0011o0111lll11i)begin if (ol0iol01l1o0lio0oi00ilo0111lll11i='1')then
<=ol0iol01l1o0lio0oio011o0111lll11i;else ol0iol01l1o0lio0oi0011o0111lll11i<=ol0iol01l1o0lio0oi0o11o0111lll11i;end
if;ol0iol01liooli000i0011o0111lll11i<=(not ol0iol0lll0oli000i0011o0111lll11i)AND not ol0iol01i10oli000i0011o0111lll11i;
end process;process(ol0iol01l1o0llo00i0011o0111lll11i,ol0iol01liooli000i0011o0111lll11i,ol0iol01l100li000i0011o0111lll11i
)begin ol0iol01l1o0llo00i0011o0111lll11i<=(ol0iol01liooli000i0011o0111lll11i)OR ol0iol01l100li000i0011o0111lll11i; end process;end;
```

Рис. 5. Результат лексической обфускации описания 1

Синтез всех приведенных описаний дает одинаковый результат. Отсюда следует простейшая атака на лексические методы – синтез. Кроме того, у специалиста может быть в распоряжении только результат синтеза, в таком случае он даже не узнает, что обфускация имела место.

Описания различаются по сложности для восприятия. Для оценки сложности описаний можно воспользоваться различными методами. Интуитивным подходом будет оценка сложности по числу терминалов, входящих в состав описания, или по числу операций. Однако многие идиомы описаний на VHDL могут содержать достаточно много терминалов и при этом быть простыми для понимания и узнаваемыми.

Метод, использующий одновременно несколько метрик оценки и весовые коэффициенты, представлен в работе [6], где приведена методика оценки обфусцирующих преобразований для VHDL-описаний с учетом издержек, вносимых обфусцирующими преобразованиями в проект.

Более подробно методы оценки сложности рассмотрены в следующем разделе.

Приведенным эквивалентным преобразованиям можно найти еще одно полезное применение помимо запутывания. Их можно использовать для внедрения водяных знаков в исходный код. Например, если выбрать 2^n различных описаний и каждому из них поставить в соответствие индекс n-битовой последовательности символов, водяной знак можно кодировать заменой описаний мультиплексоров в коде на эквивалентные, но с номерами, выбранными согласно требуемым битам водяного знака. Подобный метод для последовательностей эквивалентных ассемблерных команд приведен в работе [11]. Такой водяной знак будет достаточно слабым в случае VHDL: его легко удалить переупорядочиванием параллельных операторов. В операторах наподобие if из описания 1, где возможны несколько эквивалентных равнозначных вариантов (if(*sel* = ‘1’) или if (*sel* = ‘0’)), можно дополнительно закодировать 1 бит информации так: если указано сравнение с 1, то бит водяного знака равен 1, иначе 0. Способ имеет низкую стойкость.

5. Метрики оценки сложности

Для оценки сложности исходных кодов используются различные метрики. Метрики оценки сложности показывают качество исходного кода. Эти же метрики очень удобно использовать для оценки эффективности обфусцирующих преобразований, однако в случае обфускации целью является не минимизация значений метрик, а, наоборот, максимизация [4]. Рассмотрим несколько наиболее известных метрик и способов оценки сложности.

Метрика Маккейба показывает сложность потока управления программы [12]. В случае VHDL этого недостаточно: метрика никак не учитывает сложность параллельных взаимодействий.

Метрика Холстеда [13] оценивает сложность потока данных, игнорируя сложность потока управления.

Метрика числа строк кода (LOC) [13] в случае обфускации мало применима из-за преобразований, разрушающих форматирование (обычно они представляют все описание в виде одной строки кода). Вместо числа строк кода можно воспользоваться числом операторов.

Связность – степень направленности методов класса на решение только одной проблемы [14]. Эта метрика носит, скорее, экспертный характер и поэтому трудна для автоматического вычисления.

Сцепление – число зависимостей между классами либо модулями [13]. Чем выше сцепление, тем выше энтропия и тем труднее понять систему и ее поведение. В случае языка VHDL сцепление можно понимать двояко. Во-первых, в качестве модулей можно применять пакеты или архитектуры. Во-вторых, можно рассмотреть сцепление параллельных операторов посредством одинаковых сигналов: чем больше параллельных операторов зависят от одного и того же сигнала, тем сложнее понять их взаимодействие.

Пространственная сложность – сложность, связанная с расположением элементов в описании или программе, их расстоянием от места объявления и предыдущего использования [15]. В том же источнике предложены несколько метрик, нацеленных в первую очередь на объектно-ориентированное ПО. Например, метрика пространственной сложности атрибута класса (*CASC*) и общая пространственная сложность класса (*TCASC*) вычисляются следующим образом:

$$CASC = \frac{1}{N} \cdot \sum_{k=1}^{N} Dist(k); TCASC = \frac{1}{q} \cdot \sum_{i=1}^{q} CASC(i),$$

где $Dist(k)$ – расстояние от места данного использования атрибута до предыдущего, выраженное в строках кода; N – число использований атрибута; q – число атрибутов. Вариации подобных метрик могут быть применены и для оценки обфусцирующих преобразований в случае VHDL. Величину «расстояние», которая там определена как число строк кода, для оценки обфусцирующих преобразований можно вычислить через число операторов или выражений.

Для оценки сложности приведенных эквивалентных преобразований предлагаются следующие метрики:

M_1 – число операторов;

M_2 – среднее число операторов в параллельном выражении;

M_3 – число параллельных выражений;

M_4 – число сигналов и переменных;

M_5 – сцепление параллельных операторов (связанность параллельных операторов посредством одинаковых сигналов в списке чувствительности);

M_6 – средний размер списка чувствительности параллельных операторов;

M_7 – число объявлений (число объявленных типов, сигналов, переменных и т. д.);

M_8 – *TCASC*.

Для расчета общей сложности описания используем формулу

$$C(V) = \sum_{i=1}^{n} a_i \cdot M_i,$$

где a_i – весовой коэффициент, выбранный экспертами для метрики; M_i – рассчитанное значение i-й метрики.

В таблице приведены результаты расчета метрик для описаний мультиплексора из предыдущего раздела при $a_i = 1$ для всех метрик M_i.

Результаты оценки уровня сложности

Метрика	Вариант							
	1	2	3	4	5	6	7	8
M_1	4	4	3	3	5	8	13	28
M_2	3	3	3	3	5	7	12	3
M_3	1	1	1	1	1	1	1	7
M_4	4	4	4	4	4	7	5	13
M_5	1	1	1	1	1	1	1	1,84
M_6	3	3	3	3	3	3	3	3,42
M_7	0	0	0	0	0	3	2	9
M_8	1,875	1,875	1,75	2	2,5	2,67	5,52	7,73
$C(V)$	17,875	17,875	16,75	17	23,5	32,67	43,52	71,69

Результаты оценки показывают, что уровень сложности повышается при разрушении исходных абстракций. При замене абстракций более низкоуровневыми сложность возрастает, но не очень сильно. Наибольший прирост сложности наблюдается у преобразований, создающих новые абстракции. Количественно это проявляется в увеличении числа объявлений типов и структур данных, а также способов взаимодействия между ними (например, выражений, функций или порядка адресации). Дополнительно следует отметить, что минимальные значения сложности $C(V)$ могут быть свидетельством хорошего качества и стиля VHDL-кода.

Заключение

Обфускация представляет собой одно из новых, но перспективных направлений в защите программных кодов и описаний. В случае языка VHDL традиционные методы обфускации теряют свою силу в связи с возможностью синтеза, при котором результаты преобразований будут удалены. Язык VHDL позволяет описывать одни и те же схемы различными способами, поэтому возникает необходимость в оценке уровня сложности проводимых преобразований. Функциональная обфускация позволяет преодолеть недостатки лексической обфускации: не только исходное описание, но также результат синтеза до и после обфускации выглядят по-разному, однако поведение схемы остается одинаковым. Недостатком функциональной обфускации является внесение дополнительных аппаратных и временных издержек в схему устройства. Имеет смысл

применять лексическую и функциональную обфускацию совместно, чтобы обеспечить высокий уровень защиты на всех стадиях жизненного цикла устройства.

Список литературы

1. Hardware Security Mechanisms for Authentication and Trust // GLSVLSI 2011 [Electronic resource]. – Mode of access : http://www.glsvlsi.org/archive/glsvlsi11/Koushanfar_ MeteringGLS-VLSI.pdf. – Date of access : 9.04.2013.

2. Active Hardware Metering for Intellectual Property Protection and Security // Usenix [Electronic resource]. – Mode of access : https://www.usenix.org/conference/16th-usenix-security-symposium/active-hardware-metering-intellectual-property-protection. – Date of access : 5.04.2013.

3. Majzoobi, M. Introduction to hardware security and trust / M. Majzoobi, F. Koushanfar, M. Potkonjak. – N. Y. : Springer, 2011. – 427 p.

4. Collberg, C. A Taxonomy of Obfuscating Transformations / C. Collberg, C. Thomborson, D. Low. – Auckland : Department of Computer Science, 1997. – 36 p.

5. Иванюк, А.А. Проектирование встраиваемых цифровых устройств и систем : монография / А.А. Иванюк. – Минск : Бестпринт, 2012. – 337 с.

6. Brzozowski, M. Obfuscation quality in hardware designs / M. Brzozowski, V.N. Yarmolik // Zeszyty Naukowe Politechniki Bialostockiej. Informatyka. – 2009. – № 4. – P. 19–29.

7. Circuit_complexity // Wikipedia [Electronic Resource]. – Mode of Access : http://www.en.wikipedia.org/wiki/Circuit_complexity. – Date of Access : 8.09.2013.

8. Hou, T. Three control flow obfuscation methods for Java software / T.W. Hou, H.Y. Chen, M.H. Tsai // Software IEE Proceedings. – 2006. – Vol. 153(2). – P. 80–86.

9. Компиляторы: принципы, технологии, инструменты / А. Ахо [и др.]. – 2-е изд. – СПб. : Вильямс, 2008. – 1184с.

10. Software Protection Through Dynamic Code Mutation / M. Madou [et al.] // Information Security Applications : 6th International Workshop WISA–2005. – Jiju Island, Korea, 2005. – P. 194–206.

11. Ярмолик, В.Н. Криптография, стеганография и охрана авторского права / В.Н. Ярмолик, С.С. Портянко, С.В. Ярмолик. – Минск : Изд. центр БГУ, 2007. – 240 с.

12. Software Complexity Measurement / J. Kearney [et al.] // Communications of the ACM. – 1986. – Vol. 29. – P. 1044–1050.

13. Sheng, Y. A Survey on Metric of Software Complexity / Y. Sheng, Z. Shijie // Information Management and Engineering (ICIME). – Chengdu, China, 2010. – P. 352–356.

14. Макконелл, С. Совершенный код / С. Макконелл. – СПб. : Питер, 2005. – 893 с.

15. Gupta, V. Object-oriented cognitive-spatial complexity measures / V. Gupta, K. Chhabra // International J. of Computer Engineering & Science. – 2009. – Vol. 3. – P. 122–129.

Поступила 4.11.2013

Белорусский государственный университет
информатики и радиоэлектроники,
Минск, ул. П. Бровки, 6
e-mail: vovasq@mail.ru

V.V. Sergeichik, A.A. Ivaniuk

FEATURES OF OBFUSCATION OF VHDL-DESIGNS AND ITS COMPLEXITY EVALUATION METHODS

Lexical and functional obfuscation is formalized. Brief survey of methods of lexical obfuscation is given and their drawbacks are investigated when applied to specifications in VHDL language. Complexity evaluation methods for specifications in VHDL language are considered. Complexity evaluation for different variants of VHDL specifications of a given digital device is presented.