ГЕНЕРАТОРЫ КОНСТАНТ КАК БАЗОВЫЕ ПРИМИТИВЫ СХЕМНОЙ ОБФУСКАЦИИ

Белорусский государственный университет информатики и радиоэлектроники
г. Минск, Республика Беларусь

Сергейчик В. В.

Иванюк А. А. – д-р. техн. наук, доцент

В настоящее время проблема несанкционированного производства и использование цифровых устройств не только приводит к огромным финансовым потерям, но и ставит под удар безопасность и надежность систем. Среди многообразия угроз следует выделить пиратство, обратное проектирование, внедрение аппаратных троянов, извлечение секретной информации по сторонним каналам. Разрабатываются различные методы борьбы с такими угрозами. Один из методов – обфускация.

Обфускация (от англ. *obfuscate* – делать неочевидным, запутанным, сбивать с толку) служит для запутывания понимания функционирования схемы с целью защиты от обратного проектирования. В идеале сложность и временные затраты на обратное проектирование должны оказаться близки к затратам на разработку схемы с нуля. Кроме того, обфускация применяется для сокрытия авторских водяных знаков и пользовательских отпечатков пальцев.

Для различных языков программирования создано большое количество методов обфускации. Подробная классификация приводится в [1]. Однако в случае HDL-языков такие методы теряют свою актуальность, так как результаты их применения не приводят к изменению конечного результата синтеза, т.е. схемы устройств до и после обфускации выглядят одинаково.

В случае HDL-языков выделяют две разновидности обфускации: *лексическую* и *функциональную* (*схемную*) [2]. Лексическая обфускация затрагивает только уровень исходного кода. Здесь коренится ее основной недостаток: результат синтеза (схема) до и после обфускации остается неизменным. Формально лексическую обфускацию можно описать так:

$$V; V^* = obf(V); O(V) < O(V^*); Sch = DD(V) = DD(V^*),$$

где V – исходный HDL-код, V^* – лексически обфусцированный код, *obf* – обфускация, O – сложность, *DD* – процесс синтеза, *Sch* – схема.

Суть *схемной* обфускации заключается в получении более сложной для понимания схемы, имеющей эквивалентную функциональность:

$$V; V^* = obf(V); Sch = DD(V); Sch^* = DD(V^*); Sch \neq Sch^*; func(Sch) \equiv func(Sch^*); O(V^*) > O(V); O(Sch^*) > O(Sch),$$

где *func* – функциональность схемы.

Существует несколько подходов к схемной обфускации. «Схемы с заиканием» в случае подачи неправильного ключа продолжают функционировать, однако их производительность существенно снижается [3]. Ещё один метод – внедрение проектировщиком в схему «часовых бомб» [4]. «Часовая бомба» прекращает работу схемы после завершения ознакомительного периода до ввода корректного ключа. Следующий метод основан на увеличении пространства состояний конечного автомата путем добавления фиктивных состояний [5]. Схема начинает работу в произвольном состоянии, переход в начальное состояние можно осуществить только зная граф передачи состояний или секретный ключ. При неправильном ключе схема переходит в специальное состояние, в котором пребывает до перезагрузки.

Одним из базовых способов схемной обфускации является внедрение генераторов констант (ГК). Они представляют собой разновидность *непрозрачных предикатов*, значения которых известны на этапе обфускации, но должны быть вычислены при анализе [1]. Способ предполагает замену выводов «0» и «1» схемами (примитивами), генерирующими соответствующие логические значения постоянно:

$$V_{\{0,1\}}; DD(V_{\{0,1\}}) = S_{0,1} \notin \{V_{DD}, GND\}; Func\{S_{0,1}\} \equiv Func\{V_{DD}, GND\},$$

где $V_{\{0,1\}}$ – исходное описание примитива; V_{DD}, *GND* – сигнальные источники логической «1» и «0» соответственно.

Сложность непрозрачного предиката будет определяться сложностью анализа схемного примитива.

Важно добиться того, чтобы схема генератора не оказалась распознана и минимизирована средством синтеза. Различные схемы на основе только комбинационной логики (переключательных функций) не позволили достичь такого результата. Перспективными и эффективными с точки зрения числа используемых ресурсов выглядят последовательностные схемы и схемы, сочетающие последовательностную и комбинационную логику. В качестве дополнительных ухищрений может рассматриваться использование сигналов с хорошо определенным поведением (сигналы сброса, синхронизации и т.д.) на входах ГК.

Примеры ГК приведены на рис. 1. Они синтезированы с помощью XST I.24 Release 8.1i, синтезатор не смог распознать константы и минимизировать эти схемы. Схема *а* обладает двумя недостатками: синтезатор выдаёт предупреждение о комбинационном цикле, что даёт атакующему повод для анализа; если при включении на вход *src* подана «1», то до изменения значения входного сигнала на выходе *q* возможна «1», после изменения всегда будет «0». Устранить второй недостаток можно, подавая на вход сигнал системного сброса. Схема *б*, сочетающая последовательностную и комбинационную логику, лишена указанных недостатков, но требует больше аппаратных ресурсов, а также в ней иногда возможно возникновение паразитных импульсов (glitches).

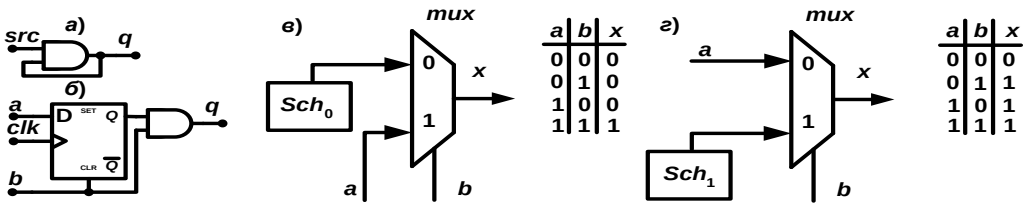

Рис. 1 – *а, б* – генераторы константы «0»; *в* – обфусцированное логическое *И*; *г* – обфусцированное логическое *ИЛИ*.

ГК служат основой для других схемных преобразований. С помощью таких базовых примитивов можно сконструировать составные компоненты для более сложных функций. Например, имея в распоряжении базовый примитив, генерирующий «0», можно реализовать логическое *И* с помощью 2-входового мультиплексора. Это показано на рис. 1(в). Аналогично можно сконструировать логическое *ИЛИ*, рис. 1(б), и другие схемы. Синтезатор не сможет распознать и минимизировать такие составные компоненты, поэтому обратному проектировщику придется их анализировать.

Важнейшим достоинством схемной обфускации является усложнение понимания не только исходного HDL-описания, но и результата синтеза (схемы). Упомянутые примеры демонстрируют важнейшие недостатки схемной обфускации: в некоторых случаях может наблюдаться увеличение аппаратных затрат и ухудшение производительности.

Качество схемных обфусцирующих преобразований можно оценивать по-разному. Во-первых, можно измерять сложность исходного описания с помощью различных метрик оценки сложности кода программ. Однако такой подход будет не полным, т.к. в нем не учитывается сложность, вносимая преобразованием в схему. Во-вторых, необходимо оценивать издержки площади и времени, вносимые в схему. В [6] предложена методика оценки, учитывающая сложность описаний, вносимые издержки и допускающая гибкий подбор метрик сложности. Также при оценке качества необходимо обратить внимание на сложность, вносимую преобразованием в схему. Для этого можно воспользоваться подходами оценки сложности цифровых схем, например, подходом измерения проектной энтропии [7]. Поведенческая энтропия H_B показывает сложность использования компонента, структурная энтропия H_S показывает сложность реализации компонента. Они вычисляются по следующим формулам (основание логарифма в расчетах примем равным 2, так как в идеале схема оперирует сигналами, принимающими одно из двух стабильных значений):

$$H_B(c) = \log(\prod_{i=1}^{n(c)} z(n_i(c)) \cdot \prod_{j=1}^{m(c)} z(m_j(c))); H_S(c) = \sum_{i \in c_b} H_B(i) + \sum_{j \in c_S} H_S(j),$$

где $n_i(c)$ – *i-ый* вход; $m_j(c)$ – *j-ый* выход; $z(n_i(c))$, $z(m_j(c))$ – число возможных состояний *i*-ого входа и *j*-ого выхода соответственно; c_b – экземпляры c; c_S – внутренние компоненты c.

Повышение сложности схемы в таком случае достигается заменой компонент эквивалентными, но с более высокими значениями H_B и H_S. Например, очевидно, что структурная и поведенческая энтропия для V_{DD} (*GND*) равны 0. Нетрудно убедиться, что для ГК значения энтропии выше: для схемы на рис. 1(а) H_B = 3, H_S = 0, а для схемы на рис. 1(б) H_B = 4, H_S = 7.

Лексической обфускации не достаточно для защиты HDL-описаний, потому что такие преобразования не изменяют результат синтеза (схему). Решением может быть функциональная обфускация, которая вносит изменения не только в HDL-описания (выступая здесь в роли лексической), но и в схему. Недостатки функциональной обфускации: увеличение потребляемой мощности, дополнительные аппаратные и временные издержки, вносимые в проект. Для повышения защиты на различных уровнях абстракции имеет смысл использовать лексическую и функциональную обфускацию совместно.

Список использованных источников:

1. Collberg, C. A Taxonomy of Obfuscating Transformations / C. Collberg, C. Thomborson, D. Low – Auckland: Department of Computer Science, 1997. – 36 p.
2. Иванюк, А.А. Проектирование встраиваемых цифровых устройств и систем: монография / А. А. Иванюк. – Минск: Бестпринт, 2012. – 337с.
3. Li, L. Structural transformation for best-possible obfuscation of sequential circuits / L. Li, H. Zhou // Hardware Oriented Security and Trust (HOST). IEEE, 2013. – Austin, Texas, USA, 2013. – P. 55 – 60.
4. Chakraborty, R.S. Hardware IP Protection during Evaluation Using Embedded Sequential Trojan / R.S. Chakraborty, S. Narasimhan, S. Bhunia // Design & Test, IEEE, 2011.
5. Desai, A. Anti-Counterfeit and Anti-Tamper Implementation using Hardware Obfuscation: Thesis: Master of Science / A. Desai. – Blacksburg, 2013. – 73p.
6. Brzozowski, M. Obfuscation quality in hardware designs / M. Brzozowski, V.N. Yarmolik // Proceedings Naukowe Politechniki Bialostockiej. Informatyka, 2009. – № 4. – P. 19 –29.
7. Menhorn, B. Design Entropy Concept – A Measurement for Complexity / B. Menhorn, F. Slomka // Codes ISSS'11. – Taipei, Taiwan, 2011. – P. 285 – 294.