УДК 004.056.53

О ПРОБЛЕМАХ ЗАЩИТЫ ИНТЕЛЛЕКТУАЛЬНОЙ СОБСТВЕННОСТИ В ПРОГРАММНЫХ СИСТЕМАХ

И. Ю. Иванов

Киевский национальный университет им. Тараса Шевченко,
03022, Киев, проспект академика Глушкова, 2, корп. 6, факультет кибернетики,
тел. (044) 259 0139, факс (044) 259 0439, e-mail: innokentiy@gmail.com

Рассматривается проблема защиты интеллектуальной собственности, воплощенной в алгоритмах и данных, содержащихся в исполняемом коде программ. Изучается современное состояние проблемы, приводятся основные достижения и открытые вопросы, а также обзор и анализ существующих методов защиты интеллектуальной собственности.

The article considers existing approaches to protection of intellectual property represented by algorithms and data structure contained in software executable. The contemporary state of the problem is studied, most important achievements and actual issues are considered. An overview and analysis of existing methods of intellectual property protection is performed.

Введение

Вследствие всестороннего проникновения программных средств во все сферы деятельности человека проблема обеспечения защиты интеллектуальной собственности (ИС), являющейся их неотъемлемой частью, становится все более актуальной. Особенности архитектуры современных операционных систем стали причиной возникновения ряда проблем, связанных с защитой ИС. В связи с тем, что в большинстве современных ОС отсутствуют какие-либо средства защиты выполняемого процесса от вмешательства со стороны других процессов, теоретически некоторый процесс может получить доступ к коду и данным любого другого процесса. Поскольку эта информация может являться предметом интеллектуальной собственности, возникает проблема защиты её от раскрытия.

С появлением интерпретируемых языков программирования (CLR-языки, Java) проблема защиты данных и алгоритмов, хранящихся в коде, вышла на новый уровень. Поскольку в интерпретируемый код (*байт-код*) записывается информация о структуре исходной программы, задача взломщика намного упрощается, так как ему приходится иметь дело уже не с машинными инструкциями, а с высокоуровневыми языковыми (обычно, объектно-ориентированными) конструкциями. Это делает проблему защиты ИС ещё более важной и требующей пристального внимания.

Следует отметить, что проблема защиты ИС тесно переплетается с другой, не менее важной проблемой – обеспечения целостности выполнения программного кода.

Защита интеллектуальной собственности, воплощенной в программном коде

Сформулируем основные требования к методу защиты программ от кражи интеллектуальной собственности:

- обеспечение максимального сокрытия алгоритмов и информации, используемых в программе;
- сохранение семантики программы;
- несущественное влияние на скорость выполнения программы и количество используемой ею памяти;
- независимость от среды выполнения и применимость к максимально широкому классу программ.

Проблема защиты данных и алгоритмов, используемых в программном коде, несмотря на свою молодость (первые работы в этой области появились в середине 90х гг. прошлого века) достаточно широко освещена в литературе. Предложено большое количество методов для обеспечения такой защиты. Однако многие из них не лишены недостатков – начиная от невозможности практического использования и заканчивая излишней техничностью, что приводит к невозможности оценить обеспечиваемый ими уровень защиты.

В данной статье рассматриваются только методы, для которых существует возможность эффективного практического применения и для которых можно построить оценки обеспечиваемого уровня защиты.

Рассматривая методы, будем исходить из следующей модели. Производитель распространяет программу среди множества пользователей (под термином «пользователь» понимается окружение, в котором может выполняться программа). Пользовательская среда выполнения в любой момент времени работы программы имеет полный доступ к её коду и данным. Некоторые методы также требуют наличия канала связи между конкретным пользователем и производителем программы.

Существующие подходы к защите программного кода можно разделить на две группы: автономные (полностью выполняемые в окружении пользователя) и взаимодействующие с вычислительной средой производителя. Очевидно, что первые выигрывают по быстродействию и удобству для пользователя, в то время

как вторые обеспечивают более высокий уровень защиты. Тем не менее поскольку наша цель состоит в сокрытии оригинальных алгоритмов, а не в подмене бизнес-модели локального использования программы сферой оказания вычислительных услуг, логичными выглядят следующие требования к неавтономным методам:

а) обмен данными между пользовательской средой и средой производителя должен быть минимальным;

б) нагрузка на вычислительную среду производителя должна быть минимальной.

В литературе широко описано семейство автономных методов, получившее название затемнения (другие термины – запутывание, обфускация).

Определение. *Затемнить программу* P [1], т.е. применить к исходной P, состоящей из объектов исходного кода (классы, методы, функции, операторы) $\{S_1,\dots,S_K\}$, множество *затемняющих преобразований* $T=\{T_1,\dots,T_N\}$ с тем, чтобы построить такую программу $P'=\{S_1'=T_{i1}(S_1),\dots,S_j'= =T_{ij}(S_j),\dots\}$, которая удовлетворяет следующим условиям:

- поведение P' полностью совпадает с поведением P (преобразования являются семантически замкнутыми);
- P' обеспечивает сокрытие семантики, т.е. анализ и реверс-инженерия P' будет занимать не меньшее время, чем анализ и реверс-инженерия программы P;
- устойчивость каждого преобразования $T_{ik}(S_j)$ является максимальной, т.е. вычислительно сложно построить программу, которая будет выполнять обратные преобразования, или же выполнение такой программы будет чрезвычайно ресурсоемким;
- незаметность преобразований T_{ik} является максимальной, т.е. по статическим свойствам S'_j будут достаточно близки к S_j;
- стоимость (дополнительное время выполнения или занимаемое пространство, вызываемое преобразованиями) P' является минимальной.

Для оценки эффективности затемняющих преобразований используются метрики сложности программного кода ([2]-[4]), показывающие зависимость сложности программы от особенностей её структуры. При построении затемняющих преобразований за основу берутся особенности среды выполнения и её архитектуры, а также учитываются сильные и слабые места существующих программных средств, выполняющих обратные преобразования (декомпиляторов).

Выделяются следующие виды затемняющих преобразований [5].

Преобразования форматирования (лексические преобразования). Самый примитивный вид затемняющих преобразований. Действие лексических преобразований распространяется только на лексическую структуру программы, т.е. фактически сводится к переименованию идентификаторов (переменных, классов, методов), затрудняя таким образом анализ логики программы на основании имен этих идентификаторов. Лексические преобразования не создают серьезного препятствия для анализа кода, так как не скрывают информацию о ходе выполнения программы.

Преобразования структур данных. Это семейство методов основывается на преобразовании используемых в коде структур и типов данных таким образом, что восстановление по ним оригинальных структур и типов является вычислительно сложной задачей. Существуют следующие преобразования структур данных.

1) Расщепление переменных. Суть преобразования заключается в том, что одна переменная представляется в виде нескольких. Например, целую переменную можно представить в виде суммы двух (или больше) целых переменных:

`Int32 A = 4;`
`Int32 B = 9;`
`Int32 C = A + B;`
	`Int32 A1 = 7, A2 = -3;`
	`Int32 B1 = 1, B2 = 8;`
	`Int32 C1 = A1 * B1;`
	`Int32 C2 = A1 * B2 + A2 * B1 + A2 * B2;`

Рис 1

Аналогичные представления можно ввести и для других типов переменных.

2) Слияние переменных. Преобразование противоположно предыдущему. Несколько переменных объединяются в одну, и все обращения к каждой из этих переменных заменяются обращением к одной. Примером может быть представление двух 32-разрядных целых переменных в виде одной 64-разрядной.

В случае если базовый язык программирования является объектно-ориентированным, дополнительно могут быть использованы следующие преобразования.

3) Усложнение дерева наследования. Увеличение глубины и ширины дерева иерархии усложняет задачу анализа кода ([3]). Добиться этого можно путем добавления промежуточных классов в иерархию;
4) приведение независимых классов к одному родителю (даже полностью абстрактному) позволяет внести ложную информацию про иерархию классов в код программы;
5) объединение функциональности разных классов в один класс/разбиение класса на несколько подклассов также способствуют затруднению анализа кода программы.

Преобразования хода выполнения достаточно хорошо изучены. Такие преобразования обеспечивают самый высокий уровень защиты программного кода. Они оперируют с графом логики выполнения программы и заключаются в изменении его структуры таким образом, что задача восстановления оригинального графа является вычислительно сложной. В литературе описано множество преобразований такого типа. Наиболее эффективным является подход, основанный на непроницаемых предикатах (opaque predicates) [5].

Определение. Предикат P является *непроницаемым* (opaque), если его значение известно до затемнения, но его сложно получить при анализе затемненной программы.

Непроницаемые предикаты позволяют внести мнимые разветвления в граф выполнения программы (рис.2):

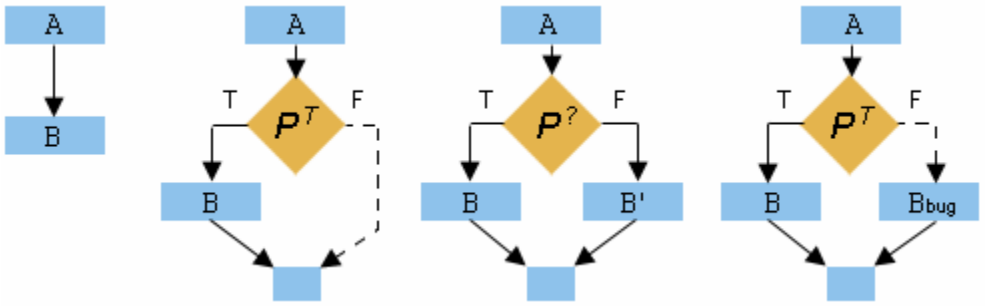

Рис. 2

Несомненным достоинством этого подхода является возможность свести исходную задачу к задаче построения устойчивых непроницаемых предикатов. С некоторыми методами построения таких предикатов можно ознакомиться в [6].

В [7] предложен метод, основанный на вырождении графа логики выполнения. Суть его заключается в том, что множество условных конструкций, используемых в программе, заменяется одним условным оператором switch, как это показано на рис. 3.

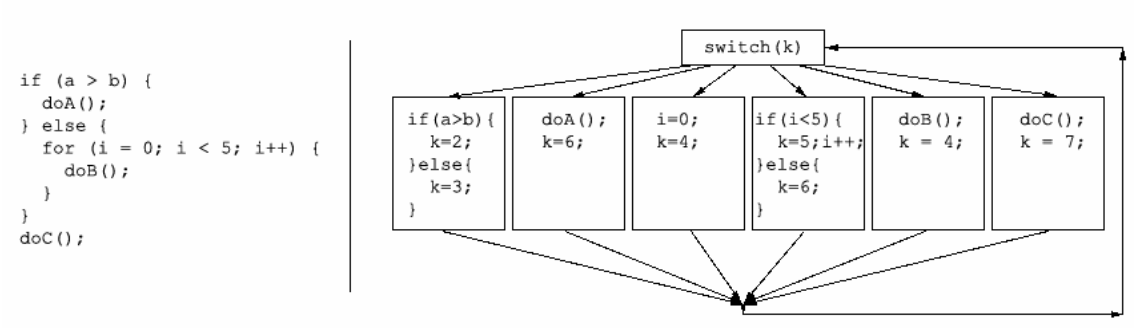

Рис. 3

Кроме того, практикуются методы, основанные на увеличении энтропии кода, в частности внесении избыточного или «мертвого» кода. Избыточный код состоит из вычислений, которых можно было бы избежать (например, вычисление константных значений в виде сложной функции). «Мертвый» код – это код, который выполняется, но не оказывает никакого влияния на результат выполнения программы.

В [8] предложен метод, основанный на вырождении графа логики выполнения программы с использованием динамической адресации функций и переменных. Все переменные и функции динамически

адресуются как элементы глобальных массивов. Такой подход делает затемненную программу устойчивой к статическому анализу.

Несмотря на обилие предложенных методов, универсального затемняющего преобразования не существует [9] и, более того, не для каждой программы можно построить адекватную затемненную программу. Тем не менее для абсолютного большинства практических задач такие преобразования построить можно.

Шифрование данных и кода. Отдельным случаем низкоуровневого затемнения являются методы, основанные на шифровании программного кода (как всего кода программы, так и отдельных его участков). Эти методы используются в основном для затемнения выполняемого машинного (не интерпретируемого) кода.

Ключ для шифрования (и, соответственно, расшифрования) кода должен быть надежно защищен. В идеальном случае, он должен отличаться для каждого пользователя программы, но такой подход является неприемлемым для продуктов с большим количеством пользователей, так как предполагает, что каждому пользователю будет выдаваться отдельная копия продукта, зашифрованная ключом этого пользователя. Поэтому более удобным вариантом является использование общего ключа и его хранение в недоступном месте (например, на смарт-карте или сервере приложений).

Уровень защиты, обеспечиваемой шифрованием кода, достаточно невысок. Это объясняется тем, что зашифрованный код непосредственно перед выполнением подлежит расшифровке, а следовательно, может быть перехвачен злоумышленником.

Секретные вычисления. Кроме затемняющих преобразований, вызывает интерес подход, основанный на секретных вычислениях (или, другими словами, на использовании *зашифрованных функций*). Суть его заключается в следующем. Производитель программы *P* заменяет функцию *f*, реализованную в программе, на некоторую функцию *g* такую, что

- восстановить функцию *f* по функции *g* вычислительно сложно;
- восстановить *f(x)* по *g(x)* $\forall x$, не имея информации про *f(x)*, вычислительно сложно;
- восстановить *f(x)* по *g(x)* $\forall x$, зная связь между *g(x)* и *f(x)*, легко.

Функция *g*, таким образом, является зашифрованным аналогом функции *f* и обозначается *g(x) = E(f)(x).*

Протокол выглядит следующим образом (рис. 4):

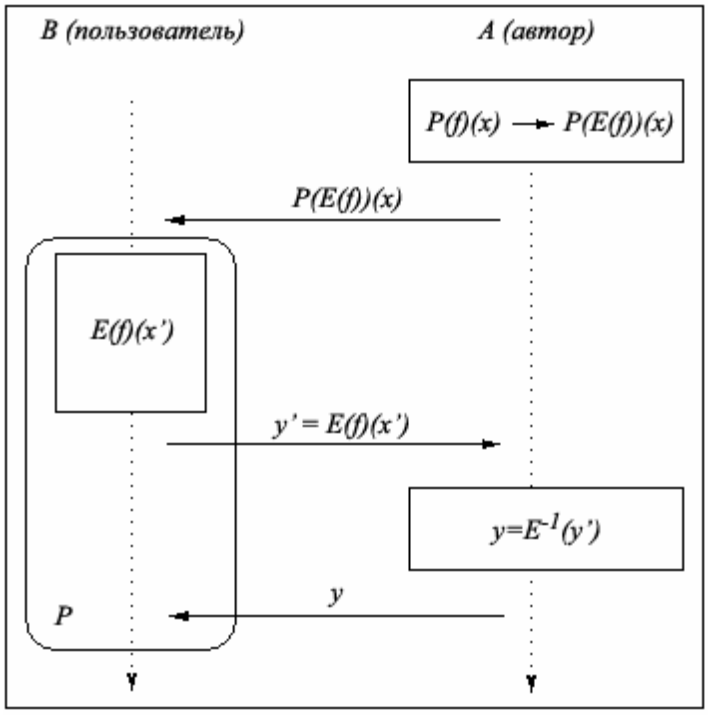

Рис. 4

1. Производитель программы *P(f)(x)* (назовем его *A*) заменяет функцию *f(x)* некоторой функцией *g(x)=E(f)(x)*, т.е. шифрует функцию *f* и посылает результат *P(E(f))(x)* пользователю *B*.
2. В ходе выполнения программы *B* вычисляет *P(E(f))(x)* на некоторых данных *x'* и посылает результат вычислений *y'* автору программы.
3. Производитель выполняет обратное преобразование $y=E^{-1}(y')$ и посылает *y* пользователю.

Следует отметить, что преобразование E нужно выбирать таким образом, чтоб основная вычислительная нагрузка ложилась на B. В обязанности A входит лишь применение обратного преобразования E^{-1} к результату вычисления $E(f)(x)$.

Наиболее интересные результаты в этой области принадлежат [10]. Несмотря на то что множество функций, которыми оперирует метод, описанный в этой статье, сравнительно невелико (и ограничено полиномиальными функциями), в будущем он может быть расширен и для других типов функций.

Выводы

Проблема защиты интеллектуальной собственности, воплощенной в программном коде, остается одной из важнейших проблем современной информатики. В связи с тем, что операционные системы и исполняющие окружения не содержат встроенных средств для предотвращения кражи интеллектуальной собственности, ощущается острая необходимость в программных и математических методах, обеспечивающих её защиту. Обилие существующих методов не решает проблему в целом – каждый из них в определенной мере техничен, а следовательно, неуниверсален. Кроме того, для многих методов сложно оценить уровень обеспечиваемой ими защиты. Поэтому в числе первейших задач, которые необходимо решить в этой области, находятся следующие:

- определить модель угроз (в том числе формализовать понятия «защитить программу», «взломать программу», «атакующий»);
- дать математические оценки сложности конкретных атак, связанных с кражей интеллектуальной собственности;
- дать математические оценки защиты, обеспечиваемой конкретными методами.

Наиболее многообещающим представляется подход, основанный на секретных вычислениях, в первую очередь ввиду его универсальности и возможности математически доказать его стойкость.

1. *Collberg C., Thomborson C., Low D.* Breaking Abstractions and Unstructuring Data Structures // Proc. of the IEEE Intern. Conf. on Computer Languages, 1998. - P. 28.
2. *Henry S., Kafura D.* Software structure metrics based on information flow // IEEE Transactions on Software Engineering. – 1981. – N. 7. – P. 510-518.
3. *Chidamber S., Kemerer C.* A metrics suite for object oriented design // Jbid. – 1994. – N. 20. – P. 476-493.
4. *Munson J., Kohshgoftaar T.* Measurement of data structure complexity // J. of Systems and Software. – 1993. – N. 3. – P. 217-225.
5. *Collberg C., Thomborson C.* Watermarking, Tamper-proofing, and Obfuscation — Tools for Software Protection // IEEE Transactions on Software Engineering. – 2002. - Vol. 28. - N. 8. - P. 735 – 746.
6. *Collberg C., Thomborson C., Low D.* Manufacturing Cheap, Resilient and Stealthy Opaque Constructs // Proc. of 25th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, 1999. - P. 184 – 196.
7. *Wang C., Davidson J., Hill J.* Protection of Software-based Survivability Mechanisms // Proc. of the 2001 Intern. Conf. on Dependable Systems and Networks (formerly: FTCS), 2001. — P. 193 – 202.
8. *Іванов І.* Комбінований метод захисту програмного забезпечення від атак з боку злочинного хоста // Матеріали конференції TAAPSD, 2005. – P. 65 – 69.
9. *Barak B., Goldreich O., Impagliazzo R. et al.* On the (Im)possibility of Obfuscating Programs // Proc. of the 21st Annual Intern. Cryptology Conf. on Advances in Cryptology, 2001. - Vol. 2139. - P. 1 – 18.
10. *Sander T., Tschudin C.* On Software Protection via Function Hiding // Proc. of the Second Intern. Workshop on Information Hiding, 1998. - P. 111 – 123.