Конференция РусКрипто'2005
4 - 6 февраля 2005 г.

Анализ современных подходов к использованию цифровых водяных знаков в программном обеспечении

Доля А.В.

Ростовский Государственный Университет
dolya@math.rsu.ru

В настоящее время цифровые водяные знаки (ЦВЗ) широко используются для защиты мультимедиа-данных. ЦВЗ позволяют предупредить кражу интеллектуальной собственности, а, если это произойдет, доказать свои права на ее владение. В данной области стеганографии проводится большое количество академических исследований, разработан целый ряд теоретических моделей [1]. Уже несколько лет технологии внедрения ЦВЗ в статические изображения используются в бизнесе.

Применение ЦВЗ для защиты ПО (software watermarking) имеет свою специфику, связанную в первую очередь со свойствами носителя информации, а также с предъявляемыми требованиями к помеченному контейнеру. В целом, данное направление стеганографии является еще очень молодым: серьезные теоретические основы были заложены лишь в 1998 году, а первая формальная модель опубликована в 1999 году [2]. Тем не менее, уже сегодня в промышленности применяются различные технологии, позволяющие встраивать ЦВЗ в ПО [6,18].

В первой части статьи рассматриваются общие понятия, используемые для описания различных теоретических подходов к использованию ЦВЗ в ПО. Вторая часть посвящена существующим и разрабатываемым технологиям внедрения ЦВЗ в ПО. Наконец, в третьей части анализируются преимущества и недостатки рассмотренных систем.

1. Введение

Задачу внедрения ЦВЗ в ПО можно сформулировать следующим образом. Необходимо встроить структуру данных ***W*** в программу ***P*** так, чтобы ***W*** можно было обнаружить в ***P*** и извлечь из нее, даже если ***P*** подверглась некоторой модификации (трансформации, оптимизации, упаковке и т.д.). При этом структура ***W*** может иметь большой размер и должна быть размещена в ***P*** скрытно; программа ***P*** с уже встроенной ***W*** не должна терять в производительности, а ***W*** должна обладать математическим свойством, позволяющим утверждать, что ее наличие в ***P*** является результатом предумышленных действий.

Следует отметить, что большое практическое значение имеют системы идентификационных номеров (digital fingerprinting). В таких системах каждый внедряемый ЦВЗ является уникальным. Применение «цифровых отпечатков пальцев» на практике позволяет не только доказать право собственности на программный продукт, но так же идентифицировать того клиента, который участвовал в создании пиратской версии продукта или допустил утечку своей копии.

Важно заметить, что ЦВЗ и идентификационные номера можно встраивать не только в программный продукт целиком, но также в его отдельные модули. Это особенно актуально в случае распространения ограниченных конфигураций продукта [17].

Исторически практика использования ЦВЗ связана с защитой программ, написанных на языке Java. Целый ряд ранних работ [4,5] посвящен разработке систем ЦВЗ, встраиваемых в байт-код Java. Как следствие и сегодня большая часть практических реализаций систем ЦВЗ создается для среды Java Virtual Machine (JVM) [11].

1.1. Атаки на системы ЦВЗ в ПО

Согласно [2] эффективность любой стеганографической системы зависит от размера встраиваемых данных (то количество информации, которое можно разместить в конкретном контейнере), скрытности (характеризующей, насколько внедренные данные незаметны для наблюдателя) и гибкости (определяющей способность заполненного контейнера противостоять атакам противника). Следует отметить, что перечисленные выше стеганографические характеристики зависят друг от друга. Например, гибкость системы можно увеличить, если задействовать большее число избыточных битов контейнера (вплоть до того, чтобы встраивать одно и то же сообщение несколько раз), однако это приведет к снижению пропускной способности.

Чтобы определить эффективность системы ЦВЗ, необходимо знать, как она противостоит различным типам атак. Считается (для систем ЦВЗ в мультимедиа-контейнерах [19]), что не существует системы ЦВЗ, которая может противостоять всем типам атак. Ниже будет также рассмотрен вопрос о невозможности создания преобразования, который бы мог трансформировать заполненные ЦВЗ контейнеры и делать их стойкими ко всем возможным атакам [16].

При дальнейшем рассмотрении различных сценариев атак используются следующие обозначения. Алиса встраивает ЦВЗ ***W*** в объект ***O*** с помощью ключа ***K***, а затем продает ***O*** Бобу. Перед тем, как Боб сможет продать ***O*** Дугласу, он должен убедиться, что ЦВЗ ***W*** стал «бесполезным». В противном случае Алиса сможет доказать, что ее права собственности на объект ***O*** были нарушены (рис. 1).

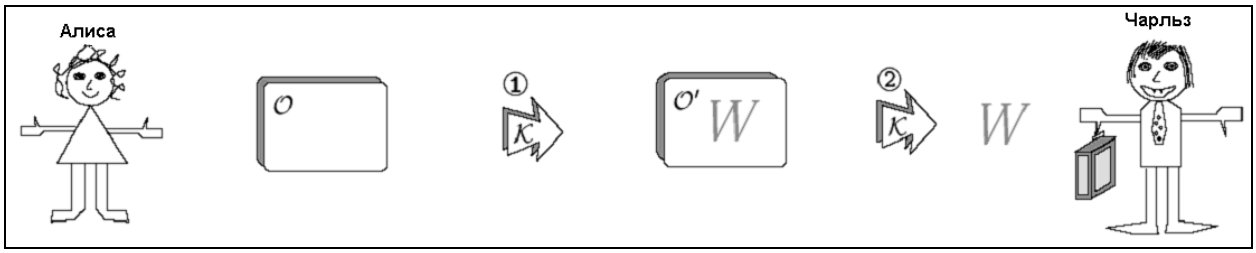

Рис. 1. Алиса создает объект ***O'***. Для этого она встраивает ЦВЗ ***W*** в объект ***O*** с помощью своего ключа ***K***. На втором шаге Боб крадет копию ***O'***, но Чарльз извлекает ***W*** из ***O'*** и, таким образом, доказывает, что ***O'*** принадлежит Алисе.

Таким образом, существуют следующие атаки на системы ЦВЗ:

- Атака, направленная на **удаление ЦВЗ**. Если Боб сможет определить наличие и местоположение ЦВЗ ***W***, он может попытаться удалить (извлечь) ***W*** из ***O***. Данная атака будет успешной в том случае, если объект, получившийся после удаления ЦВЗ, будет по-прежнему представлять ценность для Боба.
- Атака, направленная на **искажение ЦВЗ** (в терминологии [1] – геометрическая атака). Если Боб не может определить местонахождение ЦВЗ ***W***, но может каким-либо образом трансформировать объект ***O***, то он может попытаться изменить ***O*** так, чтобы Алиса не смогла определить наличие ***W*** в трансформированном объекте ***O***. Атака будет успешной, если модифицированный объект ***O*** по-прежнему представляет ценность для Боба.

- Атака, направленная на **добавление ЦВЗ**. Боб может попытаться вставить в объект ***O*** свой собственный ЦВЗ ***W'*** (или несколько таких ЦВЗ). Атака будет успешной, если Бобу удастся перезаписать ЦВЗ ***W***, вставленный Алисой (следовательно, ЦВЗ ***W*** больше нельзя будет распознать), или вставить свой собственный ЦВЗ ***W'*** так, чтобы Алиса не смогла доказать, что ее ЦВЗ ***W*** был встроен раньше.
- **Атака сговора**. Данная атака имеет смысл в основном для систем идентификационных номеров, где каждый встраиваемый ЦВЗ уникален. Атака подразумевает, что Боб попытается собрать несколько копий помеченных объектов, сравнить их и проанализировать. Атака будет успешной, если в результате Бобу удастся создать «абсолютно чистый» объект ***O***. Следует отметить, что в данной атаке могут принимать участие несколько противников, имеющих свои помеченные копии объекта ***O***. В работе [1] показано, что теоретически атака сговора, когда количество анализируемых помеченных объектов стремится к бесконечности, всегда успешна. Тем не менее, на практике противник часто ограничен экономическими и другими факторами.

Далее (рис. 2) приведена стандартная схема, иллюстрирующая различные типы успешных атак на системы ЦВЗ.

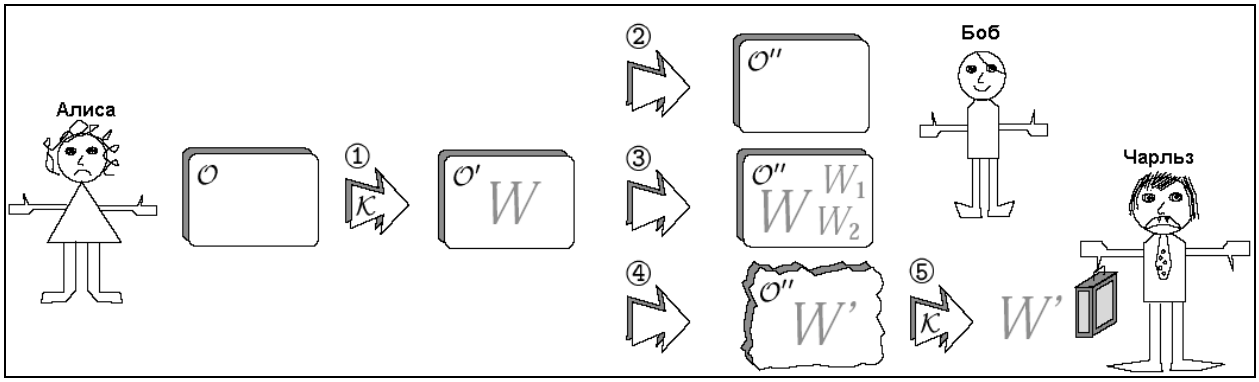

Рис. 2. Второй шаг иллюстрирует успешную атаку, направленную на удаление ЦВЗ. Боб успешно удаляет ***W*** из контейнера ***O'***. Третий шаг демонстрирует успешную атаку, направленную на добавление ЦВЗ. Боб добавляет новые ЦВЗ W_1 и W_2 и, таким образом, затрудняет работу Чарльза, который пытается извлечь ЦВЗ ***W*** и доказать право собственности Алисы. Четвертый шаг иллюстрирует успешную атаку, направленную на искажение ЦВЗ. На пятом шаге Чарльз пытается извлечь ЦВЗ ***W*** из трансформированного сообщения. Ему либо вообще не удается извлечь ЦВЗ из ***O''***, либо он извлекает искаженный ЦВЗ ***W'***.

Отдельного рассмотрения заслуживает атака сговора. Большое внимание к системам идентификационных номеров обусловлено их огромным практическим значением, так как уникальные ЦВЗ позволяют не только доказать право собственности, но и отследить владельца-нарушителя. На следующем рисунке (рис. 3) представлена схема успешной атаки сговора.

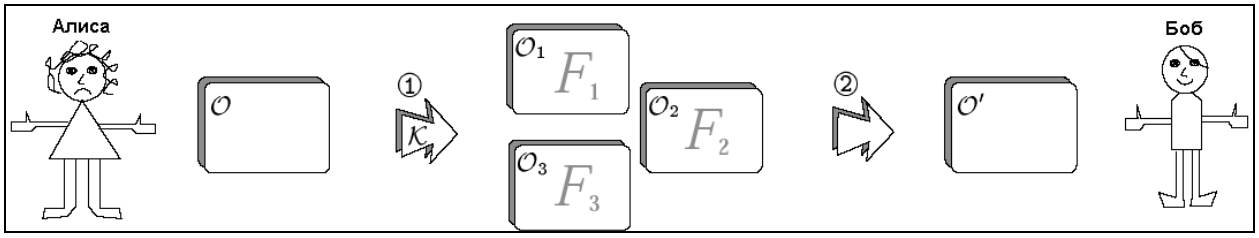

Рис. 3. На первом шаге Алиса создает несколько версий ***O***, каждая из которых содержит уникальный ЦВЗ, например, серийный номер, ***F***. На втором шаге Бобу удается успешно провести атаку сговора (сравнить объекты O_1, O_2 и O_3, удалить ЦВЗ) и получить «чистый» контейнер.

1.2. Повышение устойчивости ЦВЗ в ПО

Существует целый ряд преобразований (рис. 4), позволяющих увеличить стойкость системы ЦВЗ к определенному типу атак (tamper-proofing transformations). В некоторых случаях в качестве таких преобразований выступают скрывающие преобразования (obfuscating transformations).

Следует отметить, что скрывающие преобразования используются уже очень давно. Они позволяют усложнить анализ программы (динамический – отладку, статический – дизассемблирование). Острая потребность в создании скрывающих программ-преобразователей (obfuscator) возникла с появлением языков, компилирующихся в промежуточный код для виртуальной машины. Например, Java для JVM и .NET-языков для .NET Framework. Скрывающий преобразователь можно рассматривать, как компилятор, который на входе получает какую-то программу ***P***, а на выходе выдает новую программу ***P'***, которая сохраняет функциональность ***P***, но с некоторой точки зрения сложнее для исследования (атаки). В работе [16] доказано, что даже с самой малой степенью формализации невозможно теоретически построить скрывающее преобразование, которое бы превращало исходную программу ***P*** в «черный ящик» (включая скрывающие преобразования, которые вычисляются не обязательно за полиномиальное время, не полностью сохраняют функциональность исходной программы ***P*** и подходят только для определенных моделей вычислений). Тем не менее, на практике скрывающие преобразования позволяют несколько повысить устойчивость систем ЦВЗ к определенным видам атак.

Заметим, что формальное определение скрывающего преобразования и классификация таких преобразований даны в работе [8]. Более подробно скрывающие преобразования, актуальные в контексте систем ЦВЗ в ПО, будут рассмотрены далее (при описании статических ЦВЗ).

Есть также целый класс преобразований, не являющихся скрывающими, но позволяющими увеличить стойкость системы ЦВЗ. Такие преобразования так же будут рассмотрены в соответствующем разделе (при описании динамических ЦВЗ).

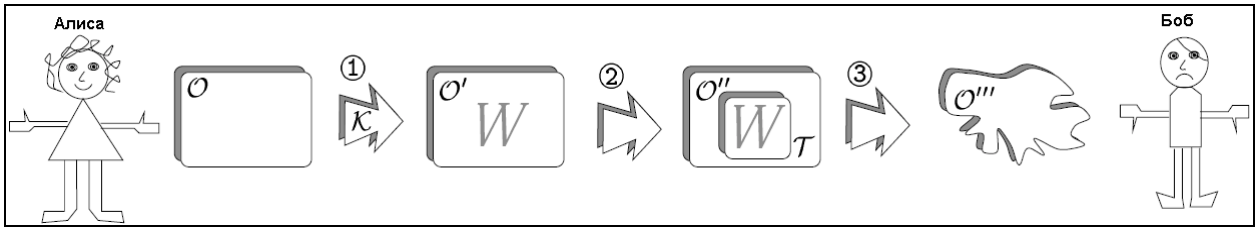

Рис. 4. На втором шаге Алиса применяет преобразование ***T***, повышающее стойкость ЦВЗ ***W*** к атаке с целью удаления ЦВЗ. Третий шаг демонстрирует неудачную атаку Боба, который пытался удалить ***W*** из дополнительно защищенного с помощью ***T*** контейнера ***O''***. В результате Боб получил объект ***O'''***, не защищенный ЦВЗ, но также и не представляющий никакой ценности.

При рассмотрении той или иной системы встраивания ЦВЗ в ПО необходимо также учитывать возможные преобразования, позволяющие повысить стойкость данной системы к какому-то определенному виду атак.

1.3. Формальная модель встраивания ЦВЗ в ПО

Формальная математическая модель ЦВЗ в ПО была построена и опубликована в 1999 году в работе [2]. Заметим, что общая модель стеганографической системы, представленная в [1], слишком сложна для систем ЦВЗ в ПО и не отражает их специфики. Приведем основные положения формальной модели встраивания ЦВЗ в ПО.

Определение 1 (ЦВЗ). Пусть ***W*** **-** это множество структур данных (ЦВЗ), а p – предикат такой, что $\forall w \in \boldsymbol{W} : p(w)$. p и ***W*** выбираются так, чтобы вероятность $p(x)$ для случайного $x \notin \boldsymbol{W}$ была мала.

Определение 2 (Программы). Пусть ***P*** – множество программ. P_w - это программа $P \in \boldsymbol{P}$ с встроенным в нее ЦВЗ $w \in \boldsymbol{W}$. Пусть dom(P) – набор входных значений, которые принимает программа P. Пусть out(P, I) – выходные значения программы P при входных значениях I. Пусть $S(P, I)$ - внутреннее состояние программы P (элемент множества всех состояний ***S***) при входных значениях I, а $|S(P, I)|$ - размер этого состояния.

Гибкость ЦВЗ w, встроенного в программу P_w, определяется в терминах атак, которые могут быть предприняты против P_w. Атаки – *трансформации (преобразования) программы*. Они могут учитывать *семантику* (если атаке подвергаются входные и выходные данные) и *состояние* (если атаке подвергается внутреннее динамическое состояние программы во время исполнения), а также не учитывать семантику (атаки с целью удаления ЦВЗ).

Определение 3 (Преобразование программы). Пусть **T** – множество всех преобразований программ в программы. $\mathbf{T}_{sem} \subset \mathbf{T}$ – множество преобразований, учитывающих семантику. $\mathbf{T}_{stat} \subset \mathbf{T}$ – множество преобразований, учитывающих состояние программы. $\mathbf{T}_{crop} \subset \mathbf{T}$ – множество преобразований, не учитывающих семантику. Тогда:

$$\mathbf{T}_{\mathrm{sem}} = \{t : \mathbf{T} \quad | \quad P \in \mathbf{P}, I \in \mathtt{dom}(P), \mathtt{dom}(P) = \mathtt{dom}(t(P)), \mathtt{out}(P, I) = \mathtt{out}(t(P), I)\}.$$

$$\mathbf{T}_{\mathrm{stat}} = \{t : \mathbf{T} \quad | \quad P \in \mathbf{P}, I \in \mathtt{dom}(P), S(P, I) = S(t(P), I)\}.$$

$$\mathbf{T}_{\mathrm{crop}} = \{t : \mathbf{T} \quad | \quad \exists P \in \mathbf{P}, \exists I \in \mathtt{dom}(P), (I \notin \mathtt{dom}(t(P)) \vee \mathtt{out}(P, I) \neq \mathtt{out}(t(P), I))\}.$$

Если преобразование учитывает семантику программы, то оно учитывает также и ее состояние. Существуют преобразования (например, оптимизация кода), которые учитывают семантику, но не состояние. Другими словами $\mathbf{T}_{stat} \subset \mathbf{T}_{sem}$.

1.3.1. Распознавание ЦВЗ в ПО

Эффективность системы ЦВЗ определяется тем набором преобразований, который может воздействовать на помеченный контейнер и после которого ЦВЗ по-прежнему можно будет распознать.

Определение 4 (Распознавание ЦВЗ). ЦВЗ $w \in \boldsymbol{W}$ в программе $P_w \in \boldsymbol{P}$ можно распознать после воздействия преобразований $T \subset \mathbf{T}$, если существуют оператор распознавания (recognizer) $\mathcal{R}_T$: (**P** x **S**) → **W** и входящие значения I такие, что:

$$\forall t \in T : p(\mathcal{R}_T(t(P_w), S(t(P_w), I))) = p(w).$$

Имея формальное определение оператора распознавания, мы можем выделить несколько подклассов таких операторов: $\mathcal{R}_{\emptyset}(P_w, S(P_w, I))$, $\mathcal{R}_{\mathbb{T}_{\text{sem}}}(P_w, S(P_w, I))$, $\mathcal{R}_{\mathbb{T}}(P_w, S(P_w, I))$, $\mathcal{R}_T(P_w, \emptyset)$, $\mathcal{R}_T(\emptyset, S(P_w, I))$.

- Оператор **тривиального** распознавания не гарантирует распознавание w, если к программе P_w применено хоть какое-то преобразование. В терминах формальной модели оператор тривиального распознавания имеет вид: $\mathcal{R}_{\emptyset}(P_w, S(P_w, I))$.
- Оператор **сильного** распознавания достаточно гибок, чтобы противостоять любым трансформациям (преобразованиям) программы, учитывающим ее семантику. В терминах формальной модели оператор сильного распознавания имеет вид: $\mathcal{R}_{\mathbb{T}_{\text{sem}}}(P_w, S(P_w, I))$.
- Оператор **идеального** распознавания достаточно гибок, чтобы противостоять вообще любым трансформациям программы. В терминах формальной модели оператор идеального распознавания имеет вид: $\mathcal{R}_{\mathbb{T}}(P_w, S(P_w, I))$.
- Оператор **статического** распознавания учитывает лишь текст (код) программы P_w, но не учитывает ее динамическое состояние во время исполнения. В терминах формальной модели оператор статического распознавания имеет вид: $\mathcal{R}_T(P_w, \emptyset)$.
- Оператор только **динамического** распознавания учитывает лишь динамическое состояние программы во время ее исполнения, но не ее код (текст). В терминах формальной модели оператор только динамического распознавания имеет вид: $\mathcal{R}_T(\emptyset, S(P_w, I))$.

1.3.2. Гибкость ЦВЗ в ПО

В дальнейшем нас будет интересовать гибкость ЦВЗ, размещенных в динамическом состоянии программы во время ее исполнения. Для того чтобы атаковать такие ЦВЗ, противник должен иметь возможность записывать свои собственные данные в динамическое состояние программы. Если ЦВЗ может быть стерт противником, который может увеличить размер динамического состояния программы на величину *r*, то будем говорить, что ЦВЗ имеет гибкость величины *r*.

Определение 5 (Гибкость величины *r*). ЦВЗ $w \in \boldsymbol{W}$ в программе $P_w \in \boldsymbol{P}$ имеет гибкость величины r по отношению преобразований $T \subset \mathbf{T}$, если существуют оператор распознавания $\mathcal{R}_T$ и входящие значения I такие, что

$$\forall t \in T : (p(\mathcal{R}_T(t(P_w), S(t(P_w), I))) \neq p(w)) \Rightarrow$$

$$\frac{|S(t(P_w), I)|}{|S(P_w, I)|} \geq r$$

Следует сразу отметить, что любой ЦВЗ обладающий гибкостью размера 1 имеет оператор сильного распознавания. Чем больше r (при $r > 1$), тем слабее система ЦВЗ.

Выше дано формальное определение гибкости величины r для систем ЦВЗ, использующих динамическое состояние программы. Однако ЦВЗ, размещенные прямо в коде программы, так же могут быть уязвимы для атак, увеличивающих статический размер кода.

Определение 6 (Гибкость размера *r*). ЦВЗ $w \in \boldsymbol{W}$ в программе $P_w \in \boldsymbol{P}$ имеет гибкость размера r по отношению преобразований $T \subset \mathbf{T}$, если существуют оператор распознавания $\mathcal{R}_T$ и входящие значения I такие, что

$$\forall t \in T : (p(\mathcal{R}_T(t(P_w), S(t(P_w), I))) \neq p(w)) \Rightarrow$$

$$\frac{|t(P_w)|}{|P_w|} \geq r$$

Многие атаки на помеченную с помощью ЦВЗ программу P_w увеличат время ее исполнения (runtime). Если время исполнения возрастет слишком сильно (хотя бы для одного входного значения из dom(P)), то такая атака не очень опасна.

Определение 7 (Гибкость времени исполнения размера *r*). ЦВЗ $w \in \boldsymbol{W}$ в программе $P_w \in \boldsymbol{P}$ имеет гибкость времени исполнения размера r по отношению преобразований $T \subset \mathbf{T}$, если существуют оператор распознавания $\mathcal{R}_T$ и входящие значения I такие, что

$$\forall t \in T : (p(\mathcal{R}_T(t(P_w), S(P_w, I))) \neq p(w)) \Rightarrow$$

$$\exists i \in \mathtt{dom}(P) \frac{\mathrm{Time}(t(P_w), i)}{\mathrm{Time}(P_w, i)} \geq r$$

1.3.3. Скрытость ЦВЗ в ПО

Некоторые системы ЦВЗ уязвимы к атаке статистического анализа. Если статические или динамические характеристики программы P_w значительно отличаются от характеристик, присущих программам данного типа, то ЦВЗ следует размещать в наиболее типичных (часто встречающихся) конструкциях.

Определение 8 (Скрытость ЦВЗ). ЦВЗ $w \in \boldsymbol{W}$ является статически скрытым в программе P по отношению к статистической мере M, если разность $M(P) - M(P_w)$ не значительна. Аналогично, ЦВЗ будет являться динамически скрытым, если разность $M(S(P,I)) - M(S(P_w,I))$ не значительна.

1.3.4. Размер ЦВЗ в ПО

ЦВЗ в ПО должен содержать столько информации, сколько это возможно без увеличения размера самой программы (ее кода) и ее динамического состояния.

Определение 9 (Эффективность кодирования ЦВЗ). $H(w)=\log_2|\boldsymbol{W}|$ - энтропия ЦВЗ w в битах, когда w извлекается из множества $\boldsymbol{W}$ по равномерному закону. Пусть $|P|$, $P \in \mathbf{P}$, это размер (в словах) программы P. Пусть $|S(P)|=\max_{I \in dom(P)} |S(P,I)|$ это наименьшая верхняя грань размера всех внутренних состояний программ P. Встраивание w в P обладает высокой статической пропускной способностью, если

$$\frac{H(w)}{\max(1, |P_w| - |P|)} \geq 1$$

Аналогично, встраивание w в P обладает высокой динамической пропускной способностью, если

$$\frac{H(w)}{\max(1, |S(P_w)| - |S(P)|)} \geq 1$$

Следует отметить, что пропускная способность измеряется в количестве «скрытых бит» на «новое» слово, добавленное во время процесса внедрения ЦВЗ.

2. Технологии встраивания ЦВЗ в ПО

Сегодня существуют технологии встраивания в ПО статических и динамических ЦВЗ (рис. 5). Статические ЦВЗ хранятся в тексте программы и не используют тот факт, что сама программа может исполняться и обладать динамическими характеристиками. Статические ЦВЗ могут храниться в коде или данных программы.

Динамические ЦВЗ размещаются в структурах данных, которые программа создает динамически во время своего выполнения. При этом сама динамическая структура данных обычно создается по мере обработки входных значений.

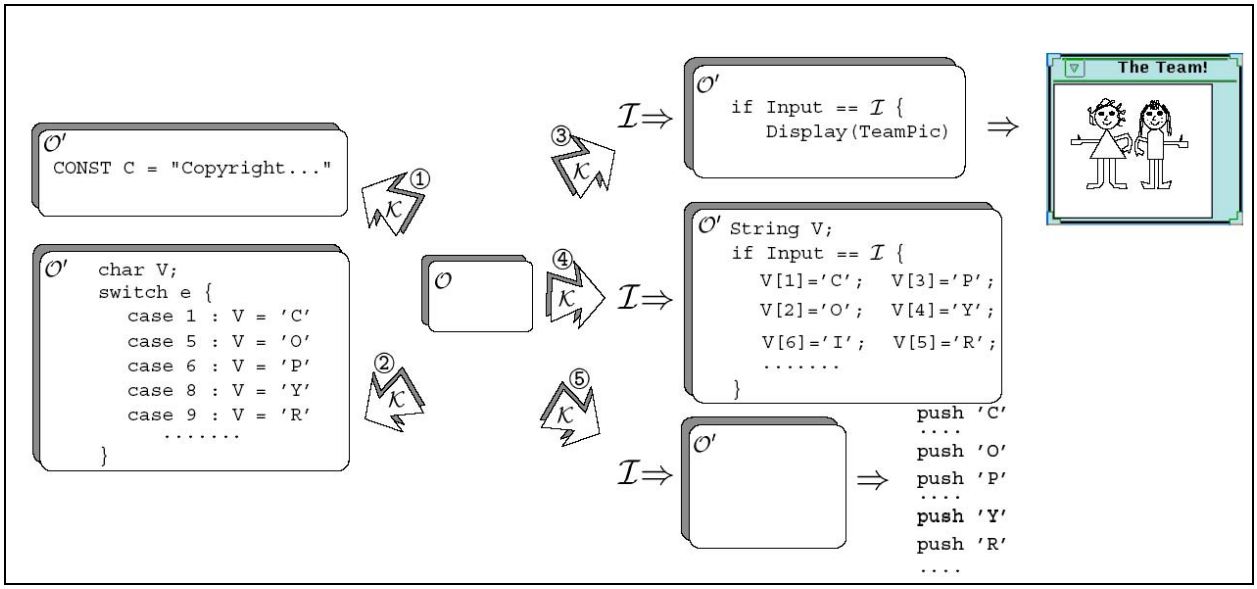

Рис. 5. ЦВЗ, получаемые в случаях 1 и 2, являются статическими, а в случаях 3, 4 и 5 – динамическими. В первом случае Алиса встраивает ЦВЗ в инициализированную строку данных в своей программе. Во втором случае ЦВЗ встраивается в секцию кода. В третьем случае ЦВЗ встраивается в, так называемое, «пасхальное яйцо» - неожиданное поведение программы в случае, если ей на вход подается значение *I*. В четвертом случае ЦВЗ размещается в глобальной переменной *V*, когда программа получает на вход значение *I*. В пятом случае ЦВЗ встраивается в ход выполнения программы, когда ей на вход подается значение *I*.

2.1. Статические ЦВЗ в ПО

В программах, написанных под операционную систему Unix, статические ЦВЗ могут храниться в секции данных (там размещены, например, статические строки), секции текста (там находится исполняемый код) или в символьной секции (там хранится отладочная информация). В программах под Windows статические ЦВЗ обычно размещаются в секциях кода или данных. В случае программ, созданных для исполнения под виртуальными машинами, ЦВЗ может храниться в самых разнообразных таблицах и секциях исполняемого файла.

Системы статических ЦВЗ начали развиваться с того момента, как стеганография оформилась как наука. Исторически это самые первые системы ЦВЗ, некоторые из которых уже запатентованы.

Патент [21] представляет концепцию встраивания ЦВЗ в графическое изображение (или другой мультимедиа контейнер) с помощью одного из существующих алгоритмов, а потом размещение этого изображения в секции статических данных программы.

Патент [22] описывает статические ЦВЗ встраиваемые с помощью конструкций вида:

```
const char c[]="Copyright (c)…"
```

или

```
const struct wmark = {0x12, 0x34, …}
```

Статические ЦВЗ могут также быть встроены в порядок перечисления выражений в конструкции `case`, в порядок не зависящих друг от друга выражений, в порядок инструкций для работы со стеком (`push` и `pop`), а также в граф управляющей логики программы.

Технология использования ЦВЗ на основе порядка следования инструкций `push` и `pop` описана в работе [23]. Этот метод приобрел известность после того, как компания IBM смогла доказать право собственности на PC-AT ROM, копию которого незаконно распространяли пираты.

Меняя порядок m выражений в конструкции `case`, можно встроить $log_2|m!| \approx log_2 \sqrt{2\pi m}\ (m/e)^m) = O(m\ log_2\ m)$ битов данных.

В патенте [24] представлена концепция встраивания ЦВЗ с помощью графа управляющей логики программы.

Следует отметить, что статические ЦВЗ очень сложно сделать более гибкими с помощью каких-либо преобразований, повышающих устойчивость или скрывающих. Так, например, патент [21] описывает метод, согласно которому в изображение (которое размещается в секции статических данных программы) внедряется не просто ЦВЗ, а фрагмент исполняемого кода программы. Во время выполнения программы этот фрагмент извлекается и исполняется. Однако исполнение кода «на лету» демаскирует ЦВЗ и позволяет противнику локализовать его.

2.1.1. Встраивание ЦВЗ на уровне бинарного кода

Некоторые базовые положения по внедрению ЦВЗ в исполняемый *x86*-код были выдвинуты в работе [25] в 1999 году. В 2003 году в статье [26] была сформулирована концепция использования ЦВЗ для защиты ПО от нелегального использования и предложены алгоритмы, использующие в качестве стеганографического контейнера промежуточный байт-код, создаваемый компилятором с языка Java для виртуальной машины JVM. В 2004 году автор [13] реализовал в своей программе *Hydan* эти алгоритмы, но уже для бинарного *x86*-кода, исполняемого преимущественно операционными системами BSD и Linux (был также создан прототип программы *Hydan* для работы под Windows XP), а также рассмотрел потенциальную возможность создания скрытого канала связи на основе исполняемого кода.

Бинарные исполняемые файлы состоят из нескольких частей: заголовка, секций данных, кода, таблиц переадресации и т.д. Мы будем рассматривать встраивание информации лишь в секцию кода (в ней в бинарном виде содержатся инструкции, напрямую исполняемые процессором). Формат PE предполагает лишь одну секцию кода в каждом исполняемом файле (для сравнения с операционными системами UNIX: файлы в форматах COFF и a. out могут так же содержать лишь одну секцию кода, а формат ELF позволяет иметь несколько таких секций). В работе [13] показано, что размер секции кода занимает в среднем 75% всего размера исполняемого файла (вне зависимости от операционной системы).

Внедрение информации в мультимедиа-файлы чаще всего предполагает изменение избыточных битов, модификация которых не будет заметна наблюдателю. Заметим, что определение избыточности зависит от типа самого контейнера. Бинарный код также содержит некоторую избыточность, которая проявляется в том, что одни и те же действия можно производить разными путями. Например, чтобы прибавить число *5* к значению, хранящемуся в регистре *eax*, можно выполнить очевидную команду « *add eax, 5»*, а можно произвести обратное действие, но с противоположным числом « *sub eax, -5 »* (то есть

вычесть -*5*). Другим примером является обнуление значения какого-либо регистра, эту операцию можно так же выполнить двумя способами: « *xor eax, eax* » и « *sub eax, eax* ». Используя две альтернативные формы для выполнения одного и того же действия можно внедрить 1 бит информации. Это легко продемонстрировать на примере команд сложения и вычитания (*add/sub*). Договоримся заранее, что каждая операция сложения будет представлять бит, равный 0, а каждая операция вычитания будет представлять бит, равный 1. Тогда мы сможем встроить значения 00, 01, и 11 в код, представленный ниже (рис. 6). Заметим, что новый код, содержащий скрытую информацию, сохраняет размер и всю функциональность оригинала.

Оригинальный исполняемый код		
83 e8 30	sub	eax, 0x30
83 f8 36	cmp	eax, 0x36
77 e5	ja	-27
83 c0 08	add	eax, 0x8
89 04 24	mov	eax, [esp]

*Код со встроенным значением **00***		
83 c0 d0	add	eax, -0x30
83 f8 36	cmp	eax, 0x36
77 e5	ja	-27
83 c0 08	add	eax, 0x8
83 04 24	mov	eax, [esp]

*Код со встроенным значением **01***		
83 c0 d0	add	eax, -0x30
83 f8 36	cmp	eax, 0x36
77 e5	ja	-27
83 e8 f8	sub	eax, -0x8
89 04 24	mov	eax, [esp]

*Код со встроенным значением **11***		
83 e8 30	sub	eax, 0x30
83 f8 36	cmp	eax, 0x36
77 e5	ja	-27
83 e0 f8	sub	eax, -0x8
83 04 24	mov	eax, [esp]

Рис. 6. Встраивание значений 00, 01, 11 в бинарный код путем вариации инструкций add/sub..

Многие инструкции могут быть использованы в двух разных формах: *<команда регистр/память, регистр>* и *<команда регистр, регистр/память>*. В случае, когда операнд «*регистр/память*» является регистром, можно заменить форму инструкции на другую. Для этого надо исправить опкод инструкции (ее бинарное представление) и поменять местами значения «*регистр/память*» и «*регистр*» (чтобы они указывали на корректные операнды). Ниже (рис. 7) приведен пример такого преобразования.

Оригинальный код		
89 f6	mov	esi, esi
80 38 2f	cmpb	0x2f, (eax)
75 09	jne	0x80480fa
8d 48 01	lea	0x1(eax), ecx
40	inc	eax
80 38 00	cmpb	0x0, (eax)

Модифицированный код		
89 **c4**	mov	**eax**, **eax**
80 **3e** 2f	cmpb	0x2f, (**esi**)
75 09	jne	0x80480fa
8d **52** 01	lea	0x1(**esi**), ebx
46	inc	esi
80 **3e** 00	cmpb	0x0, (**esi**)

Рис. 7. Кодирование информации с помощью разных форм одной и той же инструкции.

Алгоритм, описанный выше, называется алгоритмом функционально-эквивалентных инструкций. Он позволяет встраивать информацию в бинарный код, не меняя его размера и функциональности. Следовательно, программу не требуется перекомпилировать заново. Важен так же и тот факт, что для работы алгоритма не нужен исходный код программы.

На сегодняшний день с помощью данного алгоритма можно в среднем встроить 1 бит сообщения на 110 битов файла-контейнера. Алфавит такого встраивания включает 10 наборов функционально эквивалентных инструкций, описанных в работе [13]. Заметим, что по сравнению с встраиванием информации в графические изображения бинарные файлы обладают гораздо меньшей пропускной способностью (например, согласно [27] утилита Outguess, работающая с файлами в формате JPEG, позволяет внедрять 1 бит сообщения на 17 битов носителя).

В диаграмме ниже (рис. 8) представлены результаты статистического анализа, проведенного автором работы [13].

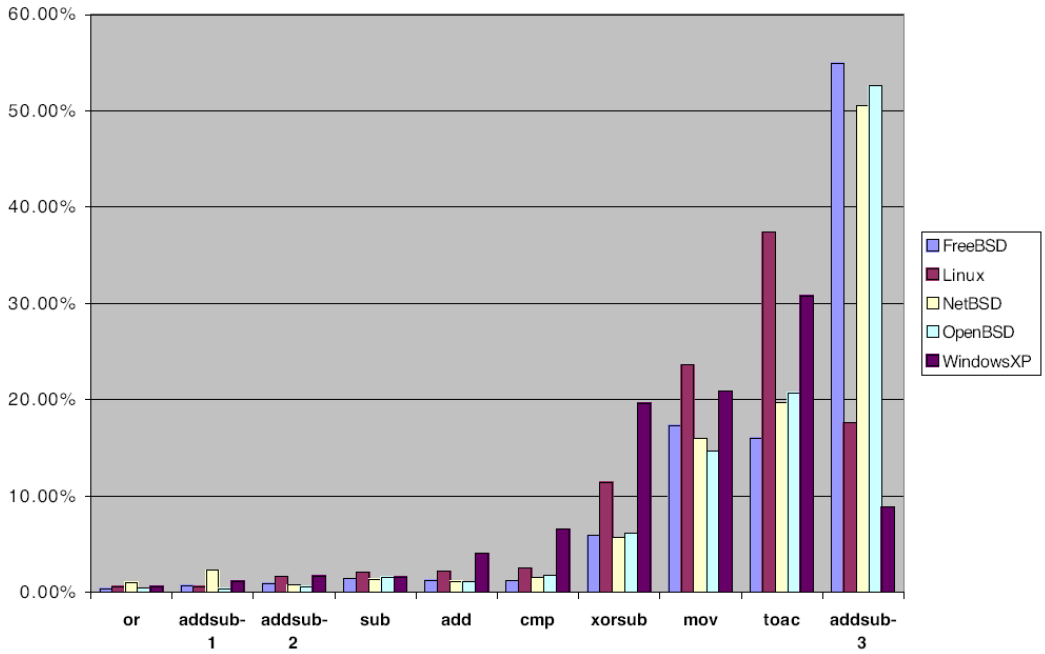

Рис. 8. Распределение функционально-эквивалентных инструкций в исполняемых файлах разных операционных систем. Группа столбцов «toac» означает инструкции «test», «or» и «and».

Эксперимент, представленный на рис. 8, позволил вычислить пропускную способность исполняемого бинарного файла (в среднем $\frac{1}{110}$), а также установить несколько фактов, характеризующих стеганографические свойства бинарных исполняемых файлов. Прежде всего, распределение инструкций внутри программ под различные операционные системы семейства Unix почти одинаково. Это объясняется тем, что для создания программ под Unix используется в основном только один компилятор – GCC. Распределение инструкций в программах под Windows, напротив, сильно отличается от аналогичного под Unix. Данный факт объясняется широким набором различных компиляторов, которые используются для разработки ПО под Windows.

Анализ распределения инструкций, исполняемых в различных операционных системах, показывает, что некоторые функционально-эквивалентные инструкции вообще не встречаются «в диком виде», а некоторые встречаются очень редко. Например, вторым операндом команды *add* всегда является регистр. Если поменять местами операнды, это сразу же вызовет подозрение наблюдателя, так как эта форма инструкции *add* никогда не встречается в «естественных» приложениях. Стоит отметить, что прибавление отрицательного числа встречается также очень редко. Таким образом, при возрастании объема сообщения наблюдателю будет все проще и проще определить наличие скрытой информации в бинарном файле.

Тем не менее, представляется возможным подсчитать скрытую пропускную способность, обеспечиваемую бинарными исполняемыми файлами и не подверженную атаке статистического анализа. Конкретные цифры представлены в таблице ниже (рис. 9).

OS	*Original Encoding Rate*	*Stealthy Encoding Rate*	*KB per bit*
OpenBSD	$\frac{1}{106}$	$\frac{1}{5099}$	66.0
FreeBSD	$\frac{1}{104}$	$\frac{1}{4823}$	61.2
NetBSD	$\frac{1}{95}$	$\frac{1}{846}$	9.81
Windows XP	$\frac{1}{137}$	$\frac{1}{74}$	1.24

Рис. 9. Скрытая пропускная способность исполняемых бинарных файлов в различных операционных системах.

Следует отметить, что скрытая пропускная способность намного меньше, чем просто емкость незаполненного контейнера. Однако ее все равно хватает, чтобы встраивать ЦВЗ в ПО.

Скрытую пропускную способность можно значительно увеличить, расширив алфавит стеганографического преобразования. Это можно сделать за счет новых функционально-эквивалентных инструкций, но намного более перспективный способ состоит в том, чтобы варьировать порядок следования функций и их аргументов. В работе [13] показано, что каждый исполняемый файл в среднем около 400 различных функций. Использование этого ресурса позволяет увеличить общую пропускную способность бинарных файлов до $\frac{1}{80}$, а варьирование порядка следования аргументов еще и до $\frac{1}{36}$.

2.1.2. Встраивание ЦВЗ на уровне исходного и бинарного кода

Концепция использования дизассемблированного листинга программы для встраивания в него ЦВЗ была впервые публично сформулирована в работе [15], в 2004 году. До этого времени подобные подходы негласно применялись в полиморфных вирусах.

Встраивание ЦВЗ на уровне ассемблерного кода удобнее всего проиллюстрировать на примере типичной программы «Hello World». Ниже (рис. 10) представлен код данной программы на языке ассемблера.

В этот код внедряется ЦВЗ, например, идентификационный номер клиента – 4436. ЦВЗ имеет вид:

```
push %eax
movl $44, %eax
movl $36, %eax
pop %eax
```

На следующем рисунке (рис. 11) данный ЦВЗ встроен в ассемблерный код программы (так же к исходному код применено скрывающее преобразование). Авторы работы [15] предполагают использовать дизассемблер IDA Pro [3] для получения ассемблерного кода исполняемой программы. Именно в листинг, получаемым на выходе IDA Pro, планируется встраивать ЦВЗ и вносить скрывающие заглушки. Авторы так же признают, что, несмотря на высокое качество результата, IDA Pro не в состоянии создать дизассемблированный файл, который потом можно автоматически компилировать вновь. Другими словами, требуется некоторая доработка кода программы с внедренным ЦВЗ. Эту доработку в общем случае невозможно провести автоматически. Однако IDA Pro позволяет легко находить ЦВЗ, встроенные по предложенному выше алгоритму, так как в дизассемблер встроены мощные средства поиска инструкций по шаблонам.

```
.file "hello.c"
            .def  ___main;    .scl  2;    .type 32;   .endef
            .text
LC0:
            .ascii "Hello, world\12\0"
            .align 2
.globl      _main
            .def  _main;      .scl  2;    .type 32;   .endef
_main:
            pushl %ebp
            movl  %esp, %ebp
            subl  $8, %esp
            andl  $-16, %esp
            movl  $0, %eax
            movl  %eax, -4(%ebp)
            movl  -4(%ebp), %eax
            call  __alloca
            call  ___main
            movl  $LC0, (%esp)
            call  _printf
            movl  $0, %eax
            leave
            ret
            .def  _printf;    .scl  2;    .type 32;   .endef
```

Рис. 10. Код типичной программы «Hello World» на ассемблере.

Следует отметить, что концепция, преложенная в работе [15], не подвержена атаке статистического анализа, так как инструкции, соответствующие внедренному ЦВЗ, компилируются вместе с исходным кодом программы заново. Таким образом, статистические характеристики получающегося файла не имеют существенных различий с типичными программами подобного типа.

Технология внедрения ЦВЗ в исходный код программы имеет много общего с представленной выше, однако позволяет автоматизировать процесс встраивания ЦВЗ. Основные идеи данного подхода описаны в работе [7].

Предполагается, что ЦВЗ будут внедряться в исходный код программы в виде ассемблерных вставок (блоков). Но алфавит используемых внутри этих вставок инструкций можно легко расширить, например, набором функционально-эквивалентных инструкций, представленных в работе [13].

Ассемблерная вставка может автоматически помещаться в код более высокого уровня, например, C++. Число таких ассемблерных вставок может варьироваться для каждой копии продукта, но в любом случае должно быть не менее 100. Это не позволит атакующему удалить все ЦВЗ вручную. Благодаря тому, что ЦВЗ помещаются на уровне исходного кода и в дальнейшем преобразуются высокоуровневым компилятором, они не нарушают статистической модели носителя. Таким образом, процесс внедрения ЦВЗ удается автоматизировать, но атакующему не так просто написать автоматический детектор, реагирующий на ЦВЗ.

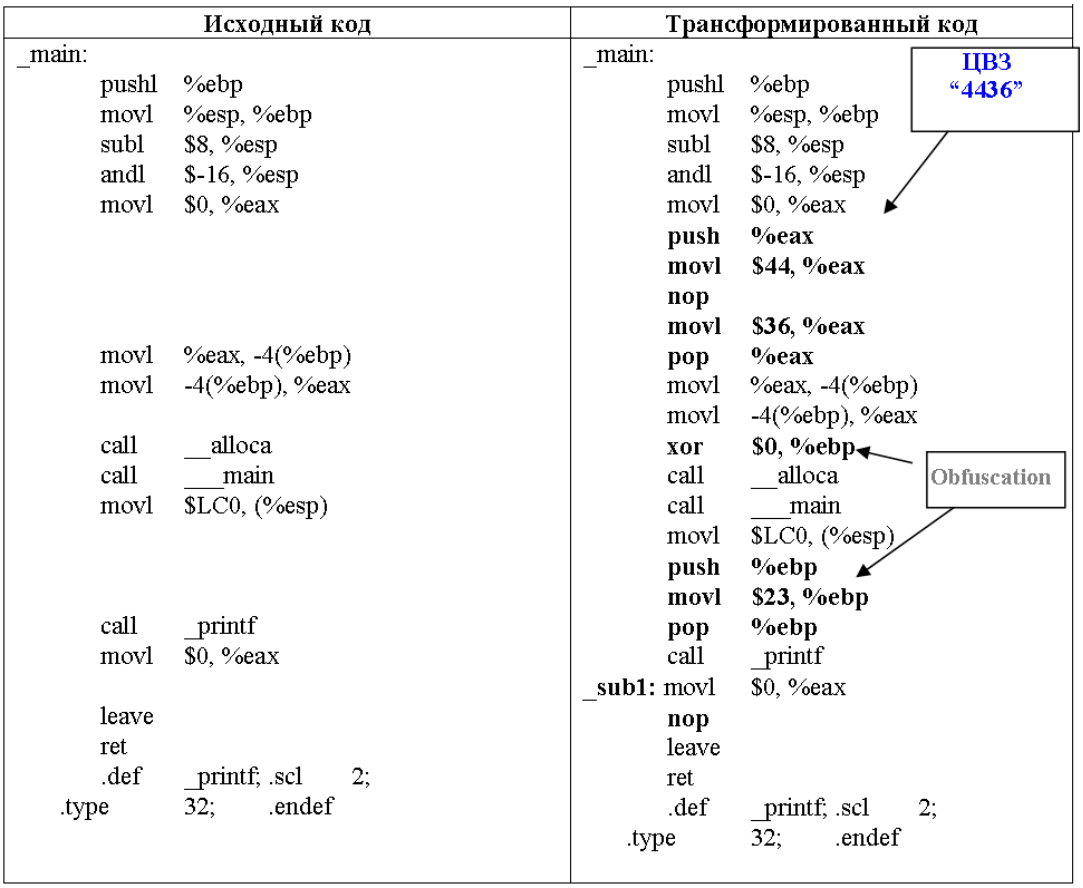

Рис. 11. В левом столбце представлен оригинальный код программы, в правом – трансформированный (с встроенным ЦВЗ и результатом скрывающего преобразования).

2.2. Динамические ЦВЗ в ПО

Динамические ЦВЗ хранятся в динамическом состоянии программы во время ее исполнения. Благодаря этому часто удается найти преобразования, позволяющие значительно повысить стойкость динамического ЦВЗ к определенному виду атаки.

Динамические ЦВЗ можно разделить на три вида: «пасхальные яйца», а также ЦВЗ, встроенные в динамические структуры данных или ход выполнения программы. В любом случае приложение запускается с предопределенной входной последовательностью $I=I_1,\ldots,I_k$, при обработке которой формируется динамический ЦВЗ. Представленные выше типы динамических ЦВЗ отличаются местом хранения ЦВЗ и способом его извлечения.

«Пасхальные яйца» представляют собой часть программы, которая получает управление только в случае получения неожиданных входных данных. Отличительной особенностью таких ЦВЗ является то, что после ввода оригинальной последовательности какое-либо действие, представляющее собой ЦВЗ, происходит немедленно. Таким образом, распознавание ЦВЗ является в данном случае тривиальной задачей. Проблема «пасхальных яиц» в том, что их очень легко обнаружить в коде программы. Существует даже несколько хранилищ подобных ЦВЗ в Интернете. Так же, если противнику удастся обнаружить искомую входящую последовательность, он легко сможет найти и сам ЦВЗ.

ЦВЗ, которые встроены в динамические структуры данных, могут размещаться в стеке, куче, области глобальных переменных и т.д. ЦВЗ обычно извлекается из переменных программы после того, как будет обработан последний элемент входящей последовательности I. Для этого в программу можно включить дополнительный код, а можно изучать программу под отладчиком. Противнику довольно сложно определить, при какой входящей последовательности будет создан ЦВЗ (в случае «пасхальных яиц» момент появления ЦВЗ очевиден). Следует отметить, что противник не может перебирать

все входящие значения *I*, так как в его распоряжении нет способа распознать сам ЦВЗ (предполагается, что код, позволяющий извлечь ЦВЗ, не распространяется вместе с программой, он может лишь присоединяться к программе при ее исследовании).

ЦВЗ, которые встроены в ход выполнения программы, могут храниться в инструкциях, адресах или одновременно и там и там. Следует отметить, что ЦВЗ будет построен, только если программа получит на вход соответствующую входную последовательность *I*. Чтобы извлечь ЦВЗ необходимо наблюдать за некоторыми свойствами исполняемого кода. Чаще всего это статические характеристики, например, последовательность исполняемых инструкций или набор адресов.

2.2.1. ЦВЗ в ПО, построенные на динамических графах

Эта технология подразумевает встраивание ЦВЗ в динамическую структуру данных, представляющую собой граф. Данный тип ЦВЗ выделяют среди всех прочих, так как при конструировании графа можно сразу учесть гибкость ЦВЗ при воздействии некоторых видов атак. Основы систем ЦВЗ на динамических графах были заложены в работах [9, 10, 12].

Основная идея ЦВЗ на динамических графах в том, чтобы встроить ЦВЗ в топологию графа. Следует сразу отметить, что вершины графа связаны между собой указателями, что несколько затрудняет анализ этой структуры данных. Так же существует большое количество преобразований, позволяющих увеличить стойкость графа к воздействию некоторых видов атак.

Главная проблема ЦВЗ, встроенных в динамические графы, состоит в том, что распознать ЦВЗ и извлечь его довольно сложно. Чтобы извлечь w из P_w оператор динамического распознавания должен проанализировать объекты, размещенные в куче, после того, как программа будет запущена с входной последовательностью *I*. После того, как будет обработан последний элемент этой последовательности, можно быть уверенным, что одна из размещенных в куче динамических структур данных представляет собой искомый ЦВЗ *w*. После этого необходимо найти нужный граф среди всего множества объектов в куче.

Существует несколько способов встроить ЦВЗ в топологию графа (алгоритм внедрения зависит от самой топологии). Рассмотрим, например, динамический список, соединенный в кольцо. Пусть в списке будет k элементов. С помощью дополнительного поля, в котором размещен указатель, можно закодировать число, разложив его на степени по основанию k. Если считать, что указатель *null* кодирует значение 0, указатель сам на себя кодирует значение 1, указатель на следующий узел кодирует значение 2 и т.д., то кодирование числа, представляющего собой произведение двух простых чисел, можно проиллюстрировать так (рис. 12):

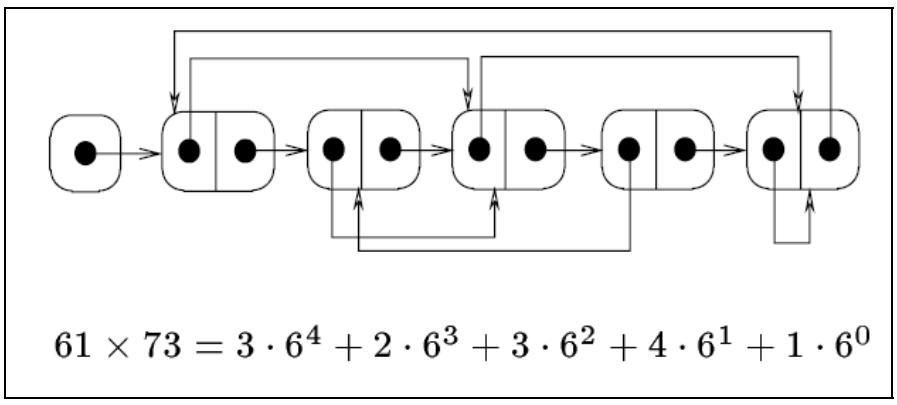

Рис. 12. Правый указатель в каждом узле указывает на следующий узел. Каждый левый указатель кодирует значение.

Есть и другие способы встраивания чисел в топологию различных графов. Многие из них приведены в известной работе Дональда Кнута [28], а также в других справочных материалах, например, [29].

Следует отметить, что атаки против ЦВЗ, размещенных в динамических графах, можно конкретизировать по сравнению с обычными атаками на системы ЦВЗ. Так злоумышленник может добавлять свои собственные дополнительные дуги в граф (если их будет довольно много, то распознать ЦВЗ будет не просто), переименовывать узлы и менять их порядок в графе, добавлять дополнительные уровни (например, расщеплять некоторые узлы на несколько), а также добавлять дополнительные узлы (чтобы нельзя было найти, например, корень дерева). Важно так же заметить, что проведение многих из этих атак требует дополнительных затрат памяти на ПК противника, а так же знания, в какой именно динамической структуре данных размещен ЦВЗ.

ЦВЗ, размещенные в динамических графах, допускают целый ряд преобразований, который повышают устойчивость ЦВЗ по отношению к какой-либо атаке. На рисунке ниже (рис. 13) приведен пример такого преобразования, позволяющего противостоять атаке разбиения вершин графа на несколько вершин. Целый ряд дополнительных преобразований, укрепляющих стойкость ЦВЗ на динамических графах, описан в работе [20].

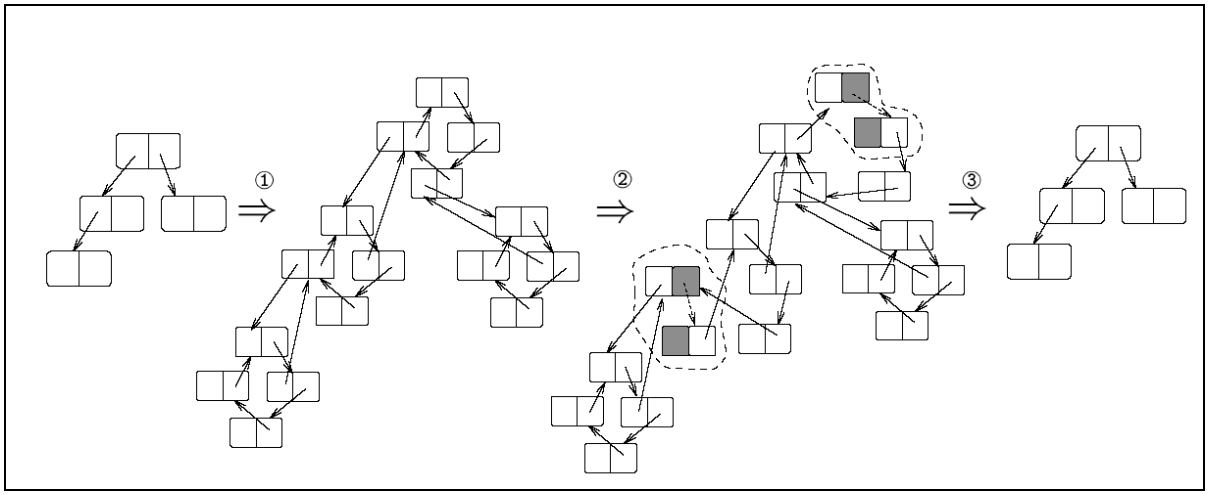

Рис. 13. На первом шаге каждый узел дерева расширяется до 4 узлов, соединенных циклически. На втором шаге противник расщепляет два узла дерева. Структура графа гарантирует, что эти узлы образуют цикл. На третьем шаге узлы, образующие цикл, объединяются в один. В результате получается граф, изоморфный исходному.

3. Анализ

Системы статических ЦВЗ в значительной мере уязвимы для атак, направленных на искажение ЦВЗ. Например, скрывающее преобразование может легко сделать ЦВЗ не распознаваемым. На практике, это означает, что упаковка, оптимизация и другие стандартные преобразования, которые проводятся автоматически, могут легко исказить ЦВЗ. В дополнение к этому к системам статических ЦВЗ очень сложно подобрать преобразования, которые бы могли укрепить стойкость системы без значительного снижения ее скрытости.

Однако статические системы ЦВЗ часто не требуют исходного кода приложения (например, система, построенная на алгоритме функционально-эквивалентных инструкций). Дополнительным преимуществом является простота реализации почти всех алгоритмов встраивания статических ЦВЗ, а также алгоритмов распознавания: не

требуется анализировать динамические объекты, необходимо извлечь лишь статическую характеристику программы.

В работе [14] описан пример удачного использования системы статического ЦВЗ для защиты исходных текстов ПО. Главное отличие этого проекта от существующих сегодня систем статических ЦВЗ для исполняемого кода в значительно более широком алфавите стеганографического преобразования. Для встраивания ЦВЗ в исходные тексты можно использовать целый ряд приемов: изменять регистр букв, локальные идентификаторы, порядок следования функций, стиль отступов и пробелов и т.д. Если вернуться к системам статических ЦВЗ в ПО, то их будущее лежит в расширении алфавита используемых преобразований. В случае алгоритма функционально-эквивалентных инструкций это, безусловно, использование порядка следования функций и аргументов.

На данный момент самым удобным способом встраивания статических ЦВЗ является технология ассемблерных вставок на уровне исходного кода. В отличие от алгоритма функционально-эквивалентных инструкций данный подход требует наличия исходного кода приложения, зато позволяет встроить сколь угодно большое число ЦВЗ без ущерба их скрытости. Следует отметить, что встраивание ЦВЗ на уровне исходного кода можно усилить так, чтобы ЦВЗ были устойчивы к преобразованиям типа оптимизации кода. Например, если ЦВЗ будут содержать лишь «пустые» инструкции (эквиваленты NOP), то оптимизатор, построив граф управляющей логики программы, сможет легко удалить ЦВЗ без ущерба для программы. Однако ЦВЗ можно замаскировать, например, под журнал событий, который заносит в какой-либо файл дату, время и какие-то характеристики программы. Следует однако признать, что все статические ЦВЗ уязвимы к многим скрывающим преобразованиям.

Системы динамических ЦВЗ намного сложнее в реализации (именно поэтому на практике пока используются статические ЦВЗ), но значительно более устойчивы к атакам и скрыты. Более того, динамические ЦВЗ допускают целый ряд преобразований, позволяющих укрепить ЦВЗ по отношению к действию атакующего преобразования. Следует также заметить, что почти все динамические ЦВЗ по умолчанию имеют иммунитет к скрывающим преобразованиям.

Наиболее перспективными на сегодняшний день в теоретическом плане являются системы ЦВЗ, построенные на динамических графах. Сами динамические структуры данных довольно сложно исследовать и противнику, и легальному владельцу продукта. Также развит ряд укрепляющих ЦВЗ преобразований именно для динамических графов. Важно так же и то, что ЦВЗ, построенные на динамических графах, требуют исходного кода приложения, и в общем случае не ясно, как можно автоматизировать процесс внедрения, например, различных идентификационных номеров в различные копии программного продукта. Представляется, что системы динамических ЦВЗ на графах будут широко применяться на практике после того, как будет создано несколько открытых реализаций. Следует отметить, что первая публичная реализация ЦВЗ на динамических графах описана в работе [11] в 2004 году. Так что это относительно новое поле для исследования.

В целом теория и практика технологий защиты ПО с помощью ЦВЗ и идентификационных номеров только развиваются. Однако темпы этого развития и низкие затраты при использовании данных технологий в промышленности указывают на то, что в самом ближайшем будущем ЦВЗ станут использоваться для защиты ПО намного чаще. В долгосрочной перспективе в руках разработчиков ПО окажутся довольно сложные, но необычайно стойкие и скрытые, системы динамических ЦВЗ, построенных на графах. Исследование таких моделей ЦВЗ и их атака на практике потребуют от противника глубоких математических и технических познаний.

Литература

[1] Цифровая стеганография / В. Г. Грибунин, И.Н. Оков, И.В. Туринцев. – М.: СОЛОН-Пресс, 2002. 272 с. ISBN 5-98003-011-5

[2] C. Collberg, C. Thomborson. Software watermarking: Models and dynamic embeddings. In Principles of Programming Languages, pages 311-324, 1999.

[3] The IDA Pro Disassembler and Debugger http://www.datarescue.com/idabase/

[4] A. Monden, H. Iida, K. Matsumoto, K. Inoue, K. Torii. Watermarking java programs. In International Symposium on Future Software Technology '99, pages 119-124, October 1999.

[5] A. Monden, H. Iida, K. Matsumoto, K. Inoue, K. Torii. A practical method for watermarking java programs. In The 24th Computer Software and Applications Conference, pages 191-197, October 2000.

[6] J. Palsberg, S. Krishnaswamy, M. Kwon, D. Ma, Q. Shao, Y. Zhang. Experience with software watermarking. In Annual Computer Security Applications Conference 2000 (ACSAC'02), pages 308-316, 2000.

[7] А.В. Доля. Встраивание цифровых водяных знаков в ПО на уровне ассемблерного и исходного кода. Восьмая московская международная телекоммуникационная конференция «Молодежь и наука». МИФИ, 2004.

[8] C. Collberg, C. Thomborson, D. Low. A Taxonomy of Obfuscating Transformations. Technical report, Dept. of Computer Science, Univ. of Auckland, New Zealand, 2004.

[9] R. Venkatesan, V. Vazirani, S. Sinha. A graph theoretic approach to software watermarking. Information Hiding, LNCS, 2137:157-168, 2001.

[10] C. Collberg, S. Kobourov, E. Carter, C. Thomborson. Error-correcting graphs for software watermarking. In 29th Workshop on Graph Theoretic Concepts in Computer Science, July 2003.

[11] C. Collberg, C. Thomborson, G. Townsend. Dynamic graph-based software watermarking. Technical report, Dept. of Computer Science, Univ. of Arizona, 2004.

[12] C. Collberg, A. Huntwork, E. Carter, G. Townsend. Graph theoretic software watermarks: Implementation, analysis, and attacks. In Workshop on Information Hiding, 2004.

[13] R. El-Khalil, A. Keromytis. Hydan: Information hiding in program binaries. In International Conference on Informaton and Communications Security, 2004.

[14] Д. Скляров. Искусство защиты и взлома информации. – СПб.: БХВ-Петербург, 2004. – 288 с. ISBN 5-94157-331-6

[15] S. Thaker, M. Stamp. Software Watermarking via Assembly Code Transformations. In Proceedings of ICCSA, 2004.

[16] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, K. Yang. On the (Im)possibility of Obfuscating Programs. In Proceedings of the 21st Annual International Cryptology Conference on Advances in Cryptology (CRYPTO '01).

[17] А.В. Доля, Д.К. Надолин. Применение конфигурационных данных в пакете ACELAN. Математическое моделирование, вычислительная механика и геофизика. Труды III Школы-семинара, Ростова-на-Дону, 15-19 ноября 2004, Изд-во «ЦВВР», 2004. 144 с. ISBN 5-94153-088-9

[18] А.В. Доля. Сравнительный анализ систем лицензионной защиты CAE- и CAD-пакетов на примере конечно-элементного комплекса ACELAN. Математическое моделирование, вычислительная механика и геофизика. Труды III Школы-семинара, Ростова-на-Дону, 15-19 ноября 2004, Изд-во «ЦВВР», 2004. 144 с. ISBN 5-94153-088-9

[19] W.Bender, D. Gruhl, N. Morimoto, A. Lu. Techniques for data hiding. IBM Systems Journal. 1996.

[20] C. Thomborson, J. Nagra, R. Somaraju, C. He, Tamper-proofing Software Watermarks. In Proc. Second Australasian Information Security Workshop (AISW2004). 2004.

[21] S. Moskowitz, M. Cooperman. Method for Stega-Cipher Protection of Computer Code. US Patent 5,745,569, January 1996. Assignee: The Dice Company.

[22] P. Samson. Apparatus and Method for Serializing and Validating Copies of Computer Software. US Patent 5,287,408. 1994.

[23] Counsil for IBM Corporation. Software birthmarks. Talk to BCS Technology of Software Protection Special Interest Group.

[24] R. Davidson, N. Myhtvold. Method and system for generating and auditing a signature for a computer program. US Patent 5,559, 884. September 1996, Assignee: Microsoft Corporation.

[25] J. Stern, G. Hachez, F. Koeune, J. Quisquater. Robust object watermarking: Application to code. In Information Hiding Workshop '99, pages 368-378, 1999.

[26] G. Hachez. A comparative study of software protection tools suited for e-commerce with contributions to software watermarking and smart cards. 2003.

[27] N. Provos. Defending Against Statistical Steganalysis. In: Proceedings of the 10th USENIX Security Symposium. 2001.

[28] D. Knuth. Fundamental Algorithms, volume 1 of The Art of Computer Programming. Addison-Wesley, Reading, MA, USA, Third Edition, 1997.

[29] F. Harray, E. Palmer. Graphical Enumeration. Academic Press, New York, 1973.