Атака по времени

Теоретическая часть

Данная атака на реализацию криптографического алгоритма RSA основана на использовании алгоритма умножения Монтгомери при быстром возведении в степень.

Алгоритм Монтгомери позволяет перейти от проведения умножения по модулю N к более удобному модулю, от которого будет удобнее брать остаток (например, если модуль равен 2^k), что может существенно ускорить возведение в степень. Но в то же время в алгоритме Монтгомери есть один существенный недостаток: в зависимости от поданных на вход чисел одинаковой длины он будет выполняться за разное время (последнее сравнение либо не выполнится и алгоритм завершит работу, или же условие выполнится и произойдет вычитание - оно называется сокращением (редукцией)), что может позволить нам делать некоторые предположения о том, какие числа умножались.

Мы сразу будем рассматривать улучшенный вариант атаки с самокорректированием.

Пусть известны i бит ключа, начиная со старшего: $b = k_0 k_1 \dots k_{i-1}$. На очередном, i-ом шаге алгоритма возведения в степень имеем промежуточный результат $m_{temp} = (m^b)^2$. Рассмотрим 4 возможные ситуации для сообщения m:

1) $(m_{temp} \cdot m)^2$ без редукции, $(m_{temp})^2$ без редукции;

2) $(m_{temp} \cdot m)^2$ с редукцией, $(m_{temp})^2$ без редукции;

3) $(m_{temp} \cdot m)^2$ без редукции, $(m_{temp})^2$ с редукцией;

4) $(m_{temp} \cdot m)^2$ с редукцией, $(m_{temp})^2$ с редукцией.

Такое разделение само по себе ничего полезного нам не дает. Рассмотрим, что может происходить при различных значениях очередного бита ключа: если $k_i = 1$, то в алгоритме вычисляется только $(m_{temp} \cdot m)^2$ (может происходить с редукцией или без), если $k_i = 0$, то вычисляется $(m_{temp})^2$:

1) ($k_i = 1$) $(m_{temp} \cdot m)^2$ вычисляется:
   (1) С редукцией (класс M_1);
   (2) Без редукции (класс M_2);

2) ($k_i = 0$) $(m_{temp})^2$ вычисляется:
   (3) С редукцией (класс M_3);
   (4) Без редукции (класс M_4).

Исходя из первой классификации, произвольное сообщение m в условиях неопределенности бита ключа должно попадать одновременно в два класса (например, M_1 и M_3, или M_1 и M_4). Однако, если m выбрано таким образом, чтобы оно заведомо попадало, например, в класс M_1, мы предполагаем, что оно с равной вероятностью может оказаться в классах M_3 и M_4.

Рассмотрим функцию-оракул $T(m)$, которая для произвольно выбранного сообщения m выдает время, за которое произойдет вычисление i-го шага алгоритма. Эта функция моделирует работу устройства, в котором зашит ключ (неизвестный внешнему наблюдателю), и которое при подаче на вход сообщения выдает отчет о времени обработки этого сообщения. Подавая на вход оракулу произвольные сообщения, мы можем еще до обращения к оракулу определить, к каким классам оно относится, а после получения ответа от оракула вычислить средние значения времени выполнения. Например, для классов M_1 и M_2 обозначим средние через $\mu(M_1)$ и $\mu(M_2)$, где

$$\mu(M_1) = \frac{T(m_1^{(1)}) + T(m_2^{(1)}) + \cdots + T(m_k^{(1)})}{k}$$
$$\mu(M_2) = \frac{T(m_1^{(2)}) + T(m_2^{(2)}) + \cdots + T(m_k^{(2)})}{k},$$

где $m_1^{(1)}, m_2^{(1)}, \ldots, m_k^{(1)}$ – сообщения, попавшие в класс M_1, $m_1^{(2)}, m_2^{(2)}, \ldots, m_k^{(2)}$ – сообщения, попавшие в класс M_2, k – число сообщений в каждом классе. Аналогичным образом мы можем вычислить средние $\mu(M_3)$ и $\mu(M_4)$. Если $k_i = 1$, то существенные различия в средних будут только у классов M_1 и M_2, и $\mu(M_1) > \mu(M_2)$ (это выполняется, поскольку время выполнения с редукцией «в среднем» больше), и при этом мы предполагаем, что $\mu(M_3) \approx \mu(M_4)$ (т.к. в них попали примерно поровну сообщений, которые на самом деле принадлежат M_1 и M_2, и потому существенной разности во времени вычисления «в среднем» нет). Аналогично, если $k_i = 0$, то $\mu(M_3) > \mu(M_4)$ и $\mu(M_1) \approx \mu(M_2)$.

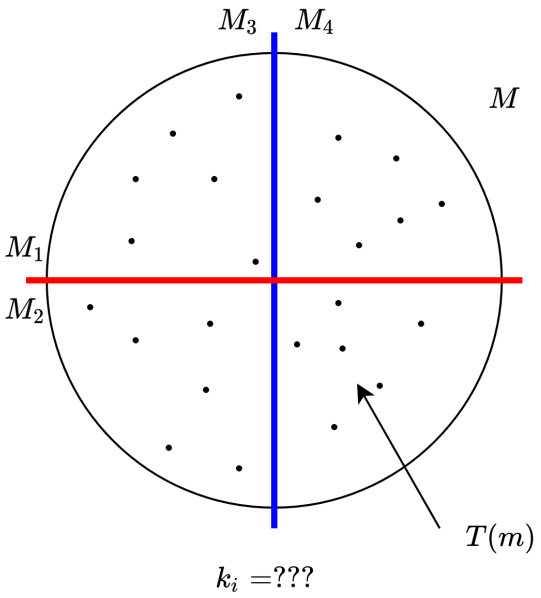

Рис. 1. (Априорное распределение).

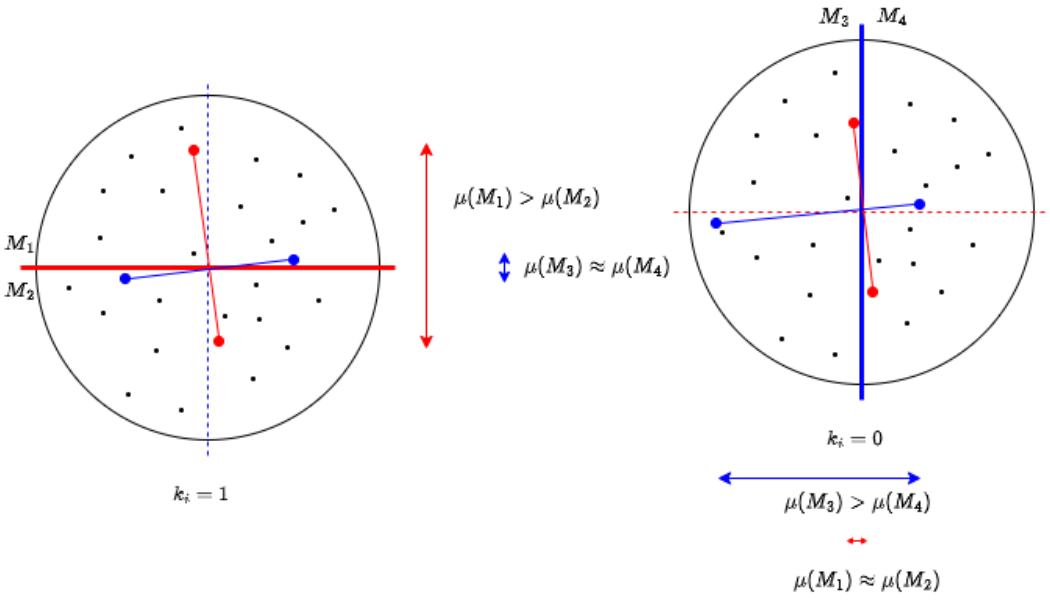

Рис. 2. Ситуации, которые возникают в результате ответа оракула.

Таким образом, описанный статистический метод позволяет эффективно различать одновременно только два класса – M_1 и M_2, либо M_3 и M_4, и наблюдая это различие в виде разности средних, мы можем сделать предположение о значении бита ключа.

Отметим, что нам достаточно оперировать только двумя классами, например

M_1 и M_2, потому что из равенства их средних автоматически следует неравенство средних для классов M_3 и M_4, и наоборот.

Примечание: последний бит не может быть раскрыт этой атакой и, следовательно, должен быть угадан.

Эта атака не страдает от проблем своей упрощенной версии (при которой рассматривается только переполнение при умножении):

1) Во-первых, наблюдаемая нами операция (т.е. квадрат) не включает постоянный фактор, и его поведение кажется гораздо менее предвзятым, чем для умножения.

2) Во-вторых, нам больше не нужно решать, имеет ли смысл разделение или нет: теперь мы должны сравнить два разделения и решить, какое из них является самым значительным.

Обнаружение ошибок

Одним из замечательных свойств нашей атаки является то, что она имеет функцию обнаружения ошибок. Коррекцию ошибок легко обосновать с интуитивной точки зрения: помните, что атака в основном состоит из моделирования вычислений и построения двух критериев для принятия решения (либо $\mu(M_1) > \mu(M_2)$ и $\mu(M_3) \approx \mu(M_4)$, либо $\mu(M_3) > \mu(M_4)$ и $\mu(M_1) \approx \mu(M_2)$), причем только один из них имеет смысл, в зависимости от искомого значения бита k_i. Также обратим внимание, что каждый шаг атаки опирается на предыдущие шаги (они нам нужны для имитации расчетов на данном шаге).

Теперь предположим, что мы приняли ошибочное решение относительно значения бита k_i. На следующем шаге мы уже не будем правильно моделировать вычисления, так что значение m_{temp}, которое мы получим, не будет тем, которое использовалось на шаге $i+1$. Наши попытки решить, будет ли умножение Монтгомери включать дополнительное сокращение или нет, таким образом, не будет иметь смысла, и критерии, которые мы будем строить, будут бессмысленны. Это останется верным для последующих битов.

На практике это приводит к аномально близким значениям для $\mu(M_1) - \mu(M_2)$ и $\mu(M_3) - \mu(M_4)$: пока выбор был правильным, два разделения были в целом легко различимы, одно из них было более значительным, а теперь они кажутся гораздо более похожими (и оба плохими) после того, как неверный выбор был сделан. Этот факт хорошо иллюстрируется на рисунке ниже, показывающим атаку 512-битным ключом на основе 350 000 наблюдений.

Критерий — это просто разница между средним временем для двух подмножеств, а на графике показано абсолютное значение $diff_1$ (разница между M_1 и M_2) минус $diff_2$ (разница между M_3 и M_4). Из графика становится ясно, что ошибка произошло рядом с битом 149.

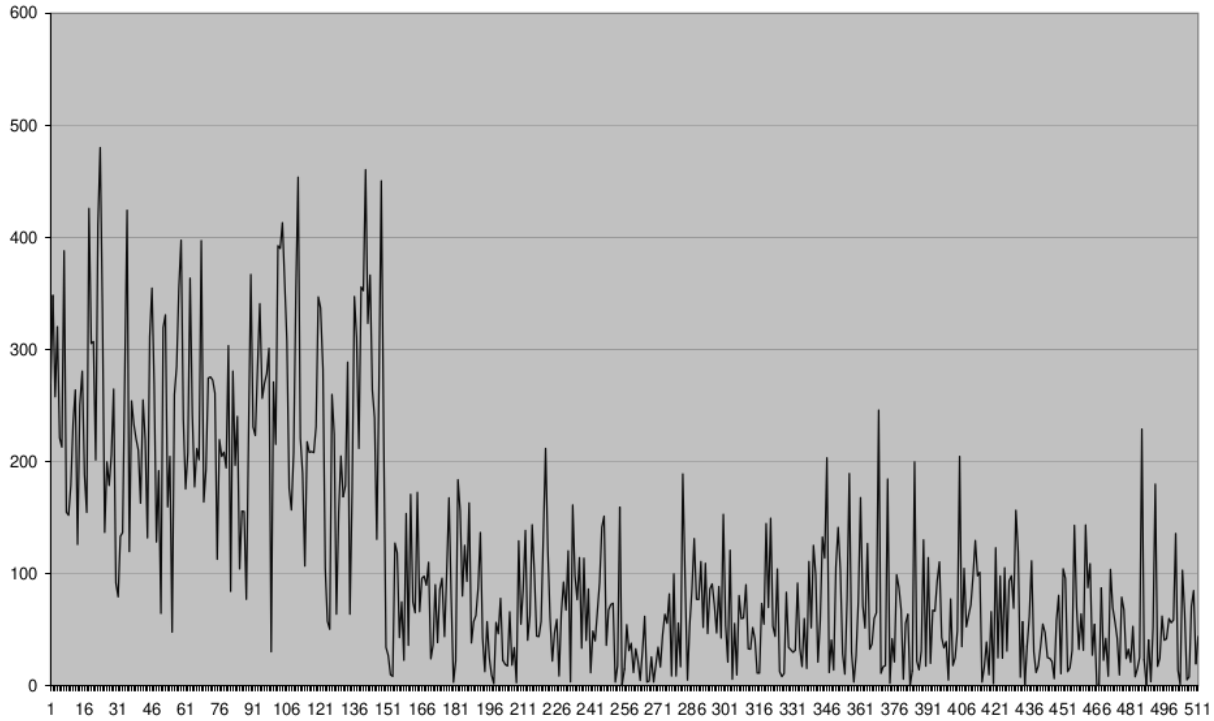

Рисунок 1: Обнаружение ошибки для 512-битного ключа

Как только ошибка обнаружена, ее нетрудно исправить, сделать другой выбор для последнего выбранного бита. После этого сделаем несколько шагов вперед, чтобы увидеть, дела идут лучше; если улучшений нет, то возвращаемся еще на шаг назад, меняем значение бита и снова проверяем. Так итеративно находим находим ошибочный бит и снова получаем нормальную разницу $diff_1 - diff_2$.

На практике такая схема исправления ошибок позволила значительно снизить объем необходимых выборок. Выборки из 10 000 текстов, например, были достаточно для восстановления 128-битных экспонент.

Параметры атаки

Скрипт, который реализует атаку, использует следующие параметры:

1) количество примеров (sampleSize) - зависит от выбранного алгоритма умножения (бывают разные реализации Монтгомери, и чем больше вероятность выполнения вычитания в алгоритме (выясняется экспериментально), тем больше примеров требуется) и от длины закрытой экспоненты

2) граница (treshold) - отвечает за принятие решения о том, произошла ли ошибка (смотри раздел про обнаружение ошибок).

3) lookAhead - количество предыдущих бит, на которые мы возвращаемся назад чтобы посмотреть, не было ли там ошибки.

Источники

1) https://crypto.stanford.edu/ dabo/papers/RSA-survey.pdf

2) P.Kocher. Timing attacks on implementations of Dffie-Hellman, RSA ,DSS , and other systems. InCRYPTO'96, volume1109 of Lecture Notes in Computer Science, pages 104-113. Springer-Verlag, 1996.

3) A practical implementation of the timing attack

4) https://github.com/paul110/RSA-Timing-Attack