Прикладная Криптография: Симметричные криптосистемы Практические аспекты

Макаров Артём

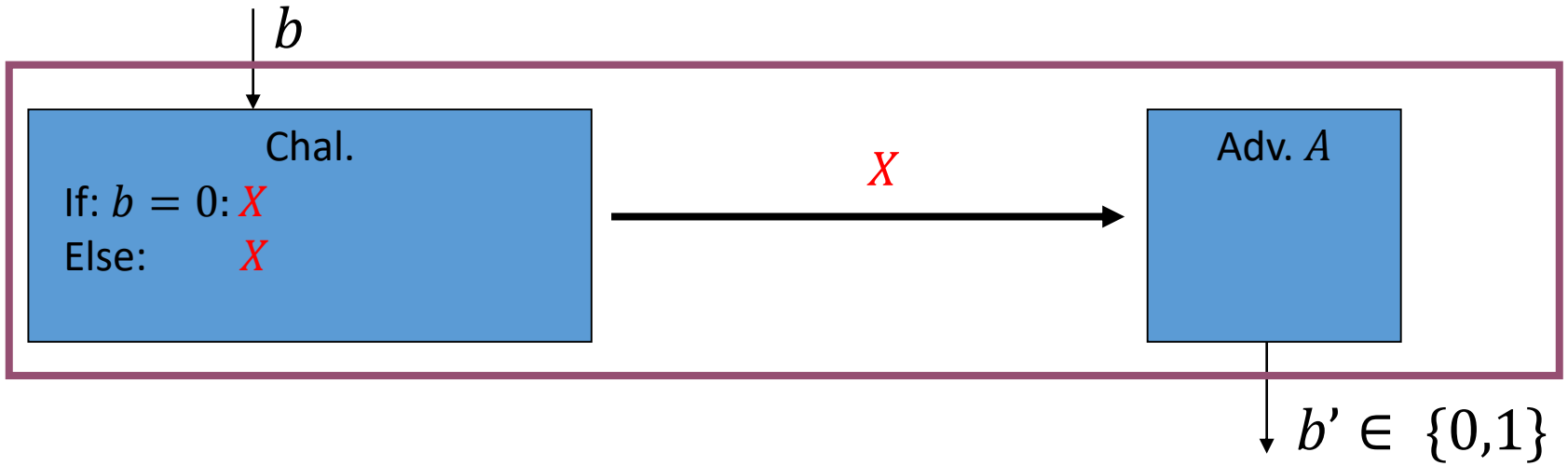
МИФИ 2024

Тест.

- Положить телефон экраном вниз справа от себя
- Не разговаривать с соседями
- Не пользоваться конспектами и электронными устройствами
- Написать номер (по таблице) и ФИО на листочке
- Написать краткий ответ на вопрос
- Дождаться окончания теста

Тест.

Пусть G псевдослучайный генератор на (S, R). Тогда игра на стойкость генератора для эффективного противника A выглядит следующим образом: (перерисовать, вместо X вписать нужное).

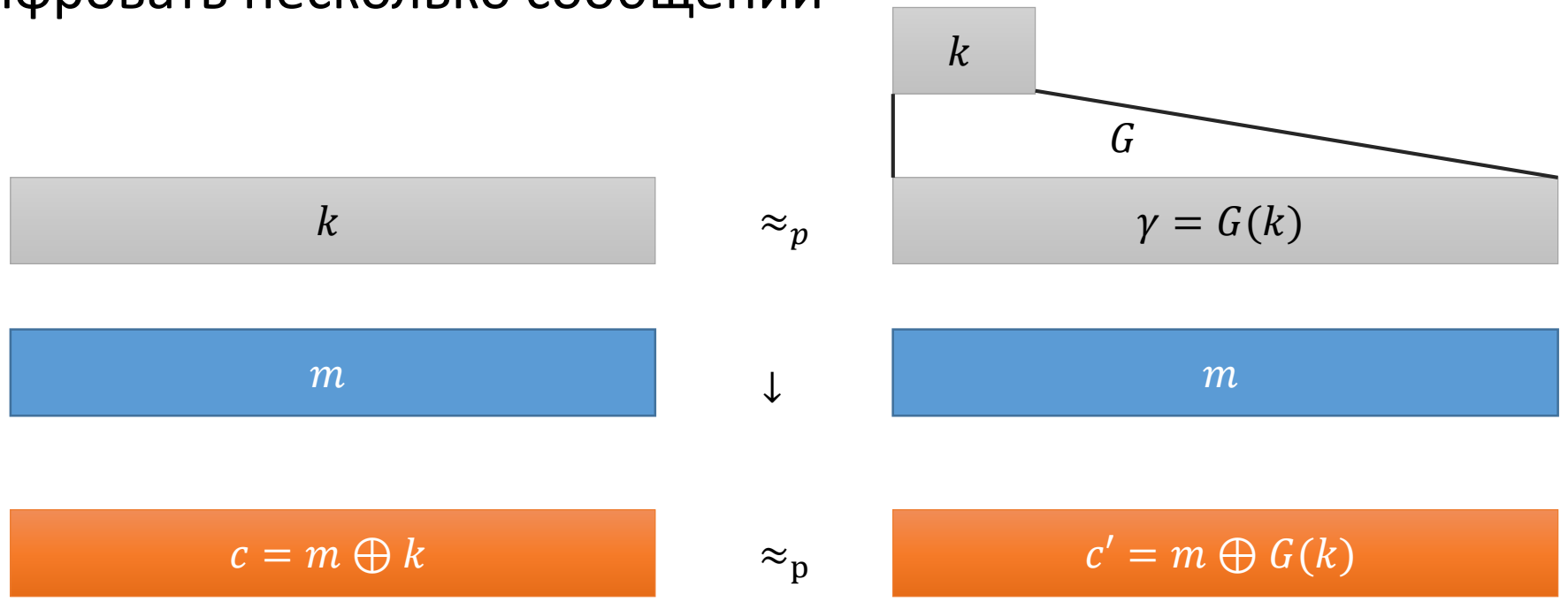


TIME IS UP

Композиция генераторов

PRG $G: S \to R$ позволяют получить псевдослучайный вектор $\gamma \in R$ (с использованием ключа $k \in S$), для использования его для зашифрования сообщения $m \in R$.

Можем ли мы зашифровать несколько сообщений $m_0, \ldots, m_q$?



Параллельная конструкция

Пусть G – стойкий PRG на (S, R).

Построим новый PRG G' на (S^n, R^n) из G следующим образом:

$$G'(s_1, \dots, s_n) = \big(G(s_1), \dots, G(s_n)\big), s_1, .., s_n \in S$$

G' называется $\boldsymbol{n}$**-ой параллельной композицией** генератора G.
Величина n **называется параметром повторения**.

Параллельная конструкция

Теорема 3.1. Пусть Пусть G – стойкий PRG на (S, R). Тогда параллельная конструкция G' построенная с использованием G – стойкий PRG с параметром повторения n.

Т.е. $\forall A$ – противника в игре на различимость против G' $\exists B$ – противник в игре на различимость против G, причём

$$PRG_{adv}[A, G'] = n * PRG_{adv}[B, G]$$

Параллельная конструкция

▻Рассмотрим последовательность из $n+1$ игры:

n. Претендент случайно выбирает $(s_1, \dots, s_n) \xleftarrow{R} S^n$ и отправляет противнику $\big((G(s_1), \dots, G(s_n)\big)$.

n-1. Претендент случайно выбирает $(s_2, \dots, s_n) \xleftarrow{R} S^{n-1}$, $r_1 \xleftarrow{R} R$ и отправляет противнику $(r_1, (G(s_2), \dots, G(s_n),)$.

…

0. Претендент случайно выбирает $(r_1, \dots, r_n) \xleftarrow{R} R^n$ и отправляет противнику $(r_1, \dots, r_n)$.

Обозначим игру H_j: первые j передаваемых элементов случайные, остальные – псевдослучайные.

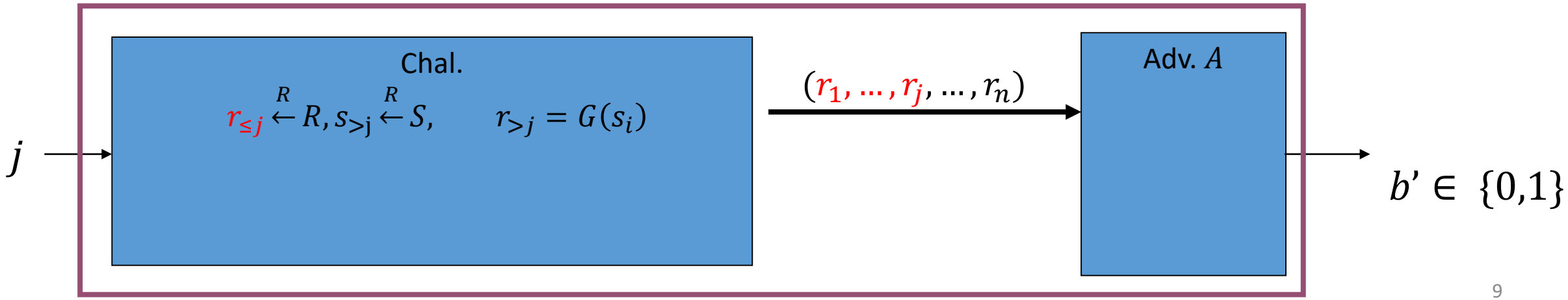
Параллельная конструкция

Пусть A – эффективный противник в игре против G'.

В эксперименте H_j игра выглядит следующим образом:

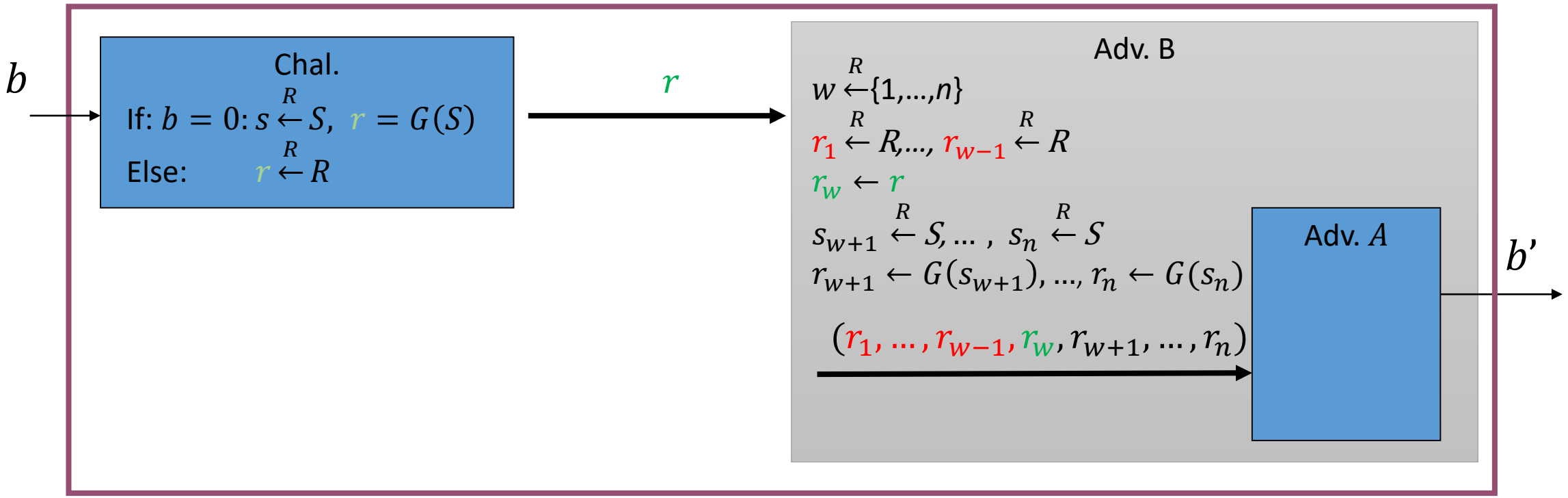
Обозначим p_j - вероятность того, что в эксперименте j величина $b' = 1$.

Тогда $PRG_{adv}[A, G'] = |p_n - p_0|$.



Параллельная конструкция

Построим алгоритм B в игре против G следующим образом:



Параллельная конструкция

Пусть W_b - событие того, что $b' = 1$, в эксперименте b игры противника B.
Заметим, что для $w = j, j = 1 \dots n$

- Эксперимент 0 эквивалентен игре H_{j-1}.
- Эксперимент 1 эквивалентен игре H_j.

Тогда $\Pr[W_0|w = j] = p_{j-1}, \Pr[W_1|w = j] = p_j$.

$$\Pr[W_0] = \sum_{j\overline{\overline{n}}1}^{n} \Pr[W_0|w = j] \Pr[w = j] = \sum_{j\overline{\overline{n}}1}^{n} \frac{1}{n} \Pr[W_0|w = j] = \frac{1}{n} \sum_{j\overline{\overline{n}}1}^{n} p_{j-1}$$

$$\Pr[W_1] = \sum_{j=1}^{n} \Pr[W_1|w = j] \Pr[w = j] = \sum_{j=1}^{n} \frac{1}{n} \Pr[W_1|w = j] = \frac{1}{n} \sum_{j=1}^{n} p_j$$

Параллельная конструкция

$$\Pr[W_0] = \sum_{j=1}^{n} \Pr[W_0 | w = j] \Pr[w = j] = \sum_{j=1}^{n} \frac{1}{n} \Pr[W_0 | w = j] = \frac{1}{n} \sum_{j=1}^{n} p_{j-1}$$

$$\Pr[W_1] = \sum_{j=1}^{n} \Pr[W_1 | w = j] \Pr[w = j] = \sum_{j=1}^{n} \frac{1}{n} \Pr[W_1 | w = j] = \frac{1}{n} \sum_{j=1}^{n} p_j$$

$$PRG_{adv}[A, G'] = |p_n - p_0|.$$

$$PRG_{adv}[B, G] = |\Pr[W_0] - \Pr[W_1]| = \left| \frac{1}{n} \sum_{j=1}^{n} p_{j-1} - \frac{1}{\mathrm{n}} \sum_{j=1}^{n} p_j \right| = \frac{1}{n} |p_n - p_0|$$

$$= \frac{1}{n} PRG_{adv}[A, G'] \blacktriangleleft$$

Последовательная конструкция

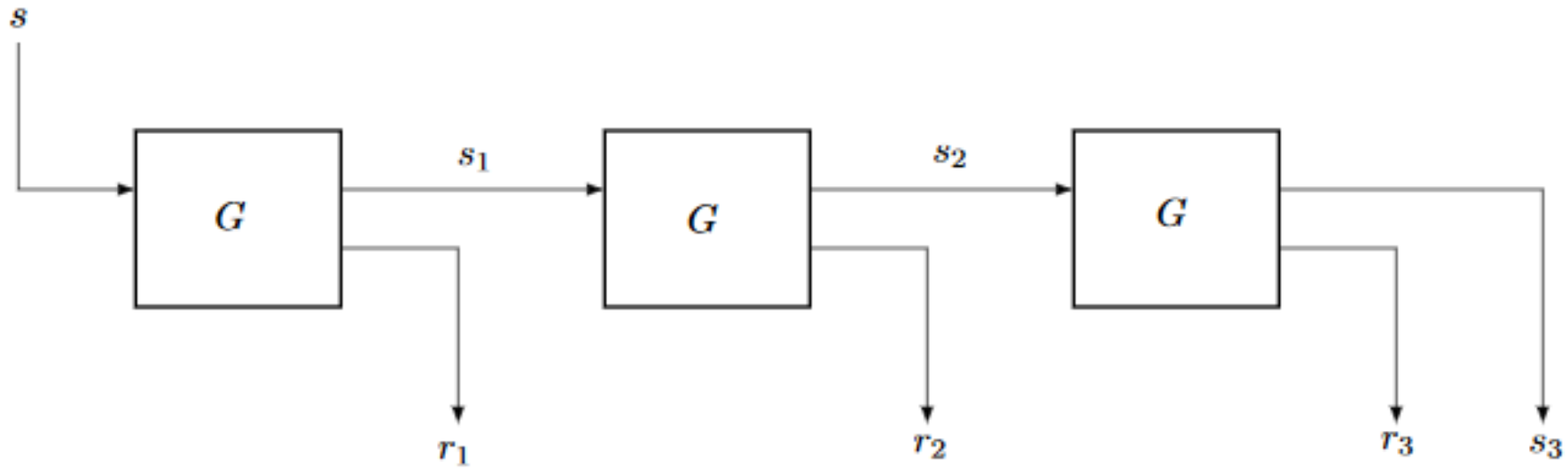
Пусть G – PRG на $(S, R \times S)$. Пусть n – параметр.

Тогда PRG G' на $(S, R^n \times S)$ определённый следующим образом:

$G'(s)$:
$s_0 \leftarrow s$
for $i = 1..n$:
$(r_i, s_i) \leftarrow G(s_{i-1})$
return $(r_1, \dots, r_n, s_n)$

называется **n-ой последовательной композицией** генератора G.

Последовательная конструкция



Последовательная конструкция

Теорема 3.2. Пусть Пусть G – стойкий PRG на (S, R). Тогда последовательная конструкция G' построенная с использованием G – стойкий PRG с параметром повторения n.

Т.е. $\forall A$ – противника в игре на различимость против G' $\exists B$ – противник в игре на различимость против G, причём

$$PRG_{adv}[A, G'] = n * PRG_{adv}[B, G]$$

▻ без доказательства. Идея доказательства – аналогичная **Теореме 3.1** – построение гибридных игр H_j и построение противника B использующего противника A в гибридной игре.◅

LCG (Linear Cong. Generator)

Простой способ построения генераторов – линейный конгруэнтный генератор. Для параметров a, b, p и ключа $k \in \{0, \dots, p-1\}, r[0] = k$.

$r[i] \leftarrow a * r[i-1] \mod p$
$i++$
return $r[i]$

НИКОГДА НЕ ИСПОЛЬЗОВАТЬ ДЛЯ КРИПТОГРАФИИ!

Используются как генераторы общего назначения в стандартных библиотеках многих языков. Пример:

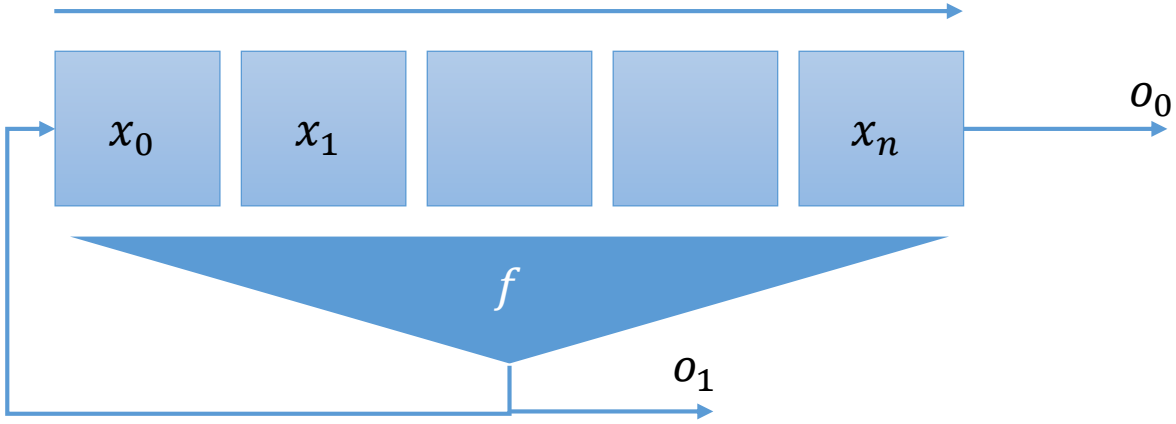
glibc random():

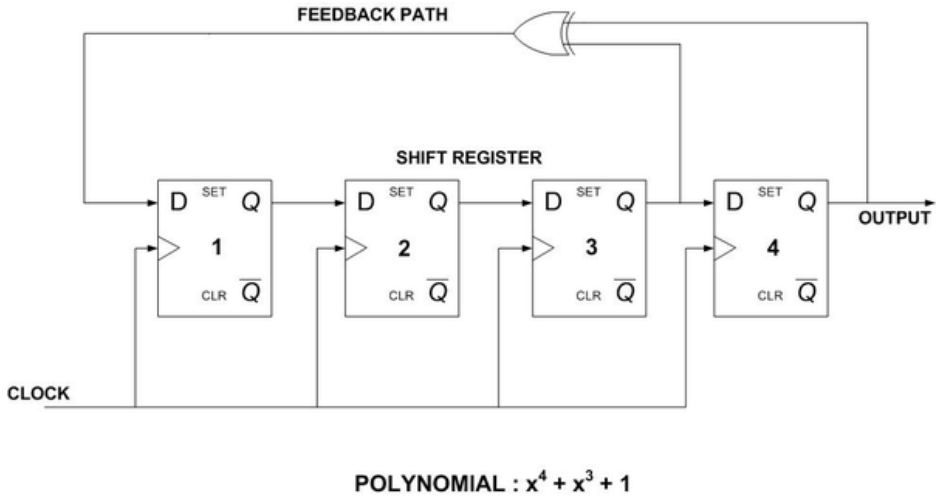
r[i] ← (r[i-3] + r[i-31]) % 2^{32}

output r[i] >> 1

LFSR – Linear feedback shift register

Линейный регистр сдвига с обратной связью (ЛРСОС). Как правило выходом является либо значения функции обратной связи (o_1), либо «выталкиваемый» бит (o_0).





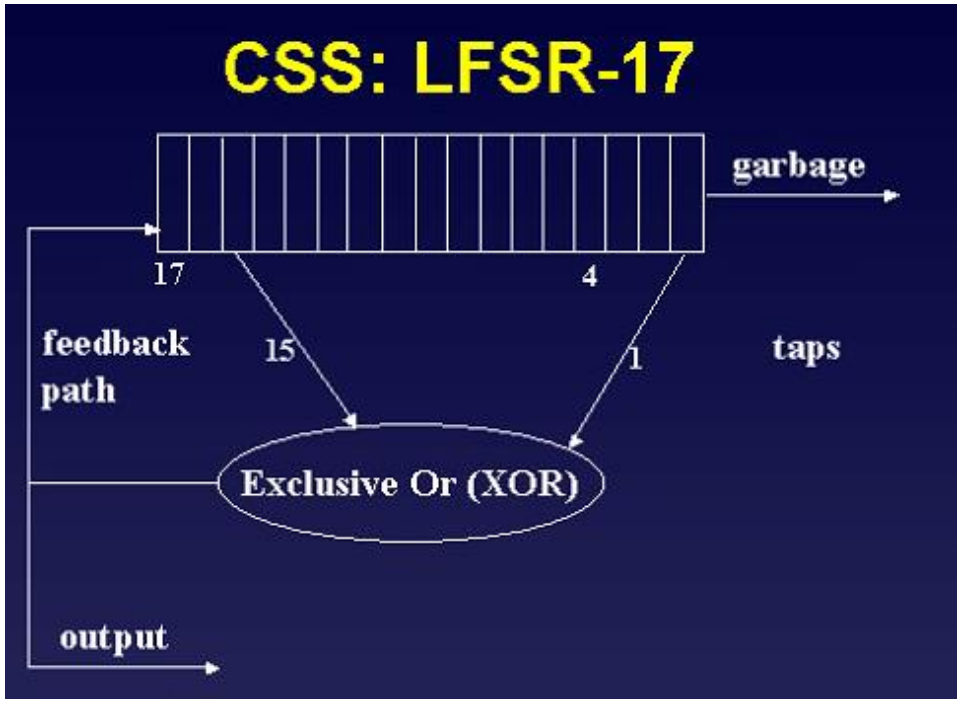
Схемотехническая реализация регистра с функцией обратной связи $f = x_3 \oplus x_4$ с использованием синхронного D триггера.

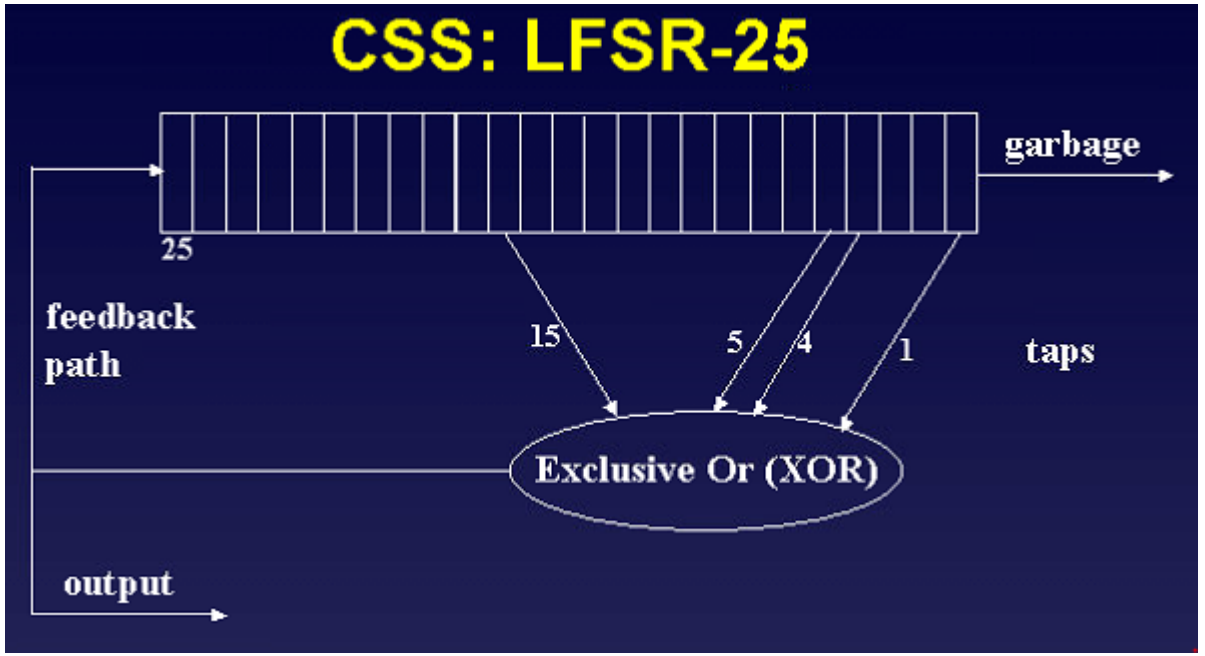
CSS

$S = \{0, \dots 7\}^5$. $s \in S$ – изначальное заполнение регистров LFSR-17 и LFSR-25

$s_1 = (1, s[0], s[1])$ – 17 бит.

$s_2 = (1, s[2], s[3], s[4])$ – 25 бит.



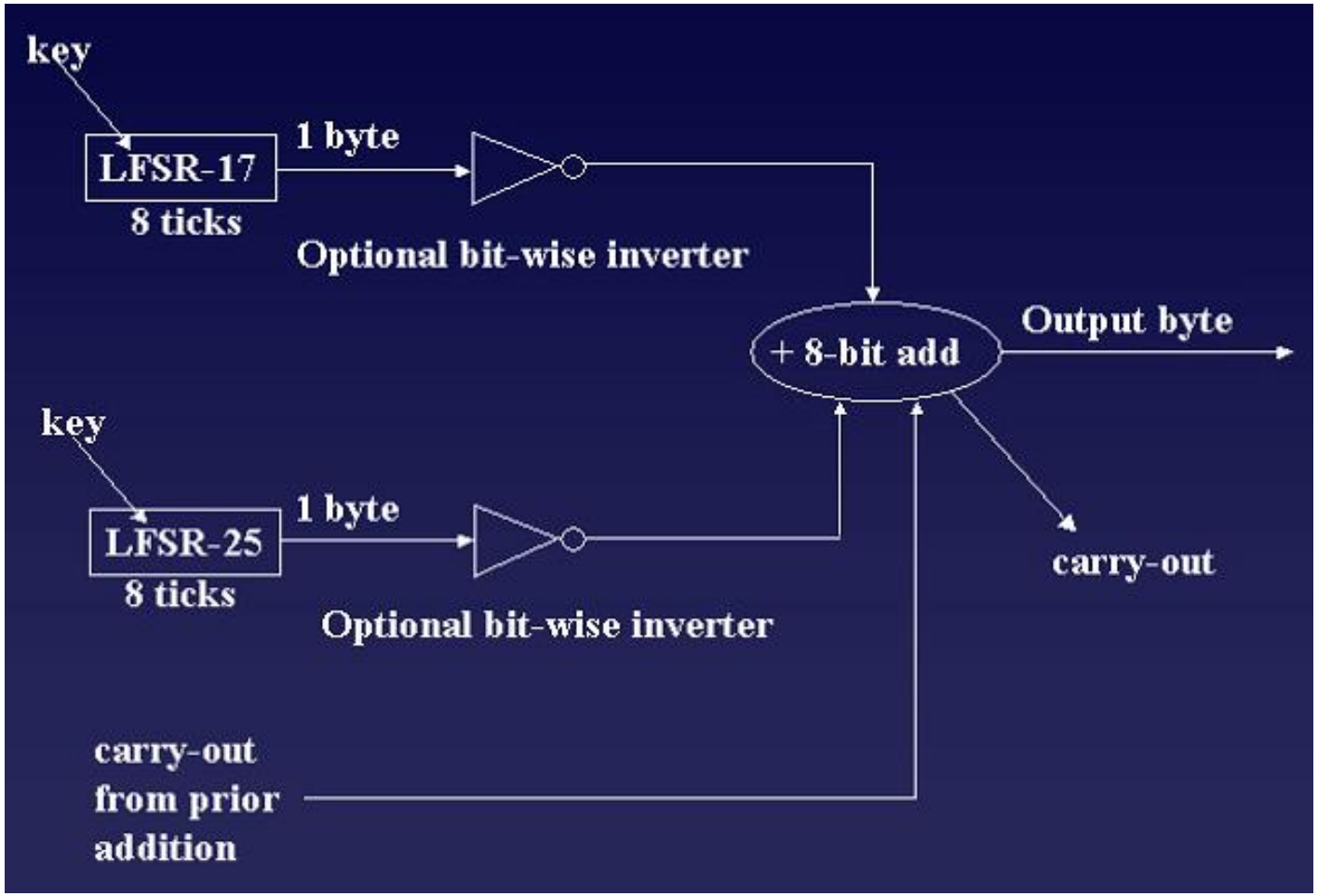


CSS

Возможна атака сложность $\sim 2^{17}$.

Идея атаки – имея выход CSS (γ), предположить начальное состояние LFSR-17, получить предполагаемый выход LFSR-17(γ_{17}), получить предполагаемый выход LFSR-25 (γ_{25}), восстановить начальное состояние LFSR-25 по выходу, проверить корректность следующих байтов выхода CSS.

Подробнее – [https://www.cs.cmu.edu/~dst/DeCSS/Kesden/]



Использование Nonce

Можем ли мы использовать ключ повторно в PRG, избегая двухразового блокнота?

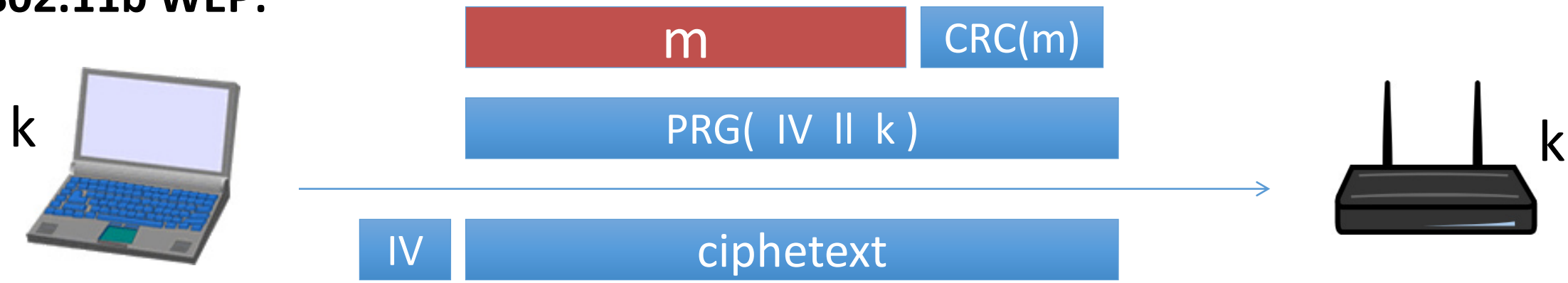
Идея – использование уникальной величины nonce.

RPG $G: S \times R \rightarrow \{0,1\}^n$, где $s \in S$ – ключ, $r \in R$ – nonce (неповторяющаяся величина для фиксированного $s \in S$).

Шифр $E(k, m, r) = m \oplus G(k, r)$, пара (k, r) не должна повторяться.

WEP, или как не надо использовать nonce в поточных шифрах

802.11b WEP:



Длина IV: 24 bits

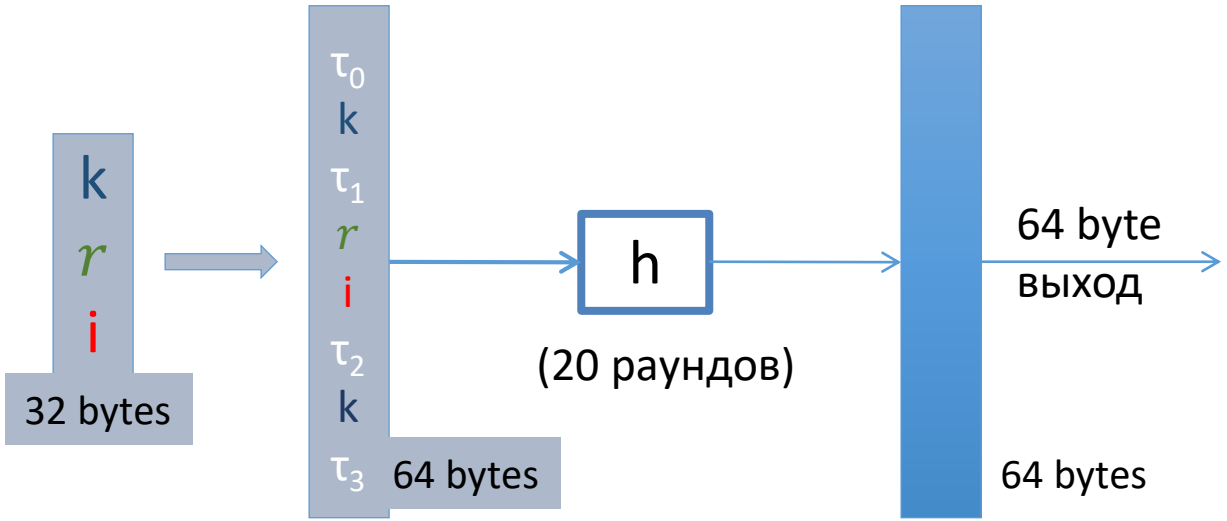
- Повторение IV $2^{24} \approx 16M$ frames
- Начальная инициализация IV на многих устройствах = 0

Salsa20

Salsa 20: $\{0,1\}^{128(256)} \times \{0,1\}^{64} \to \{0,1\}^{n}, n \leq 2^{73}$

r– nonce, k – ключ, i – счётчик.

Salsa $20(k, r) = (H(k, (r, 0)), H(k, (r, 1)), \ldots)$, H – необратимая функция сжатия Roomba.



"expa"	Key	Key	Key
Key	"nd 3"	Nonce	Nonce
Pos.	Pos.	"2-by"	Key
Key	Key	Key	"te k"

D. J. Bernstein

Salsa

"expa"	Key	Key	Key
Key	"nd 3"	Nonce	Nonce
Pos.	Pos.	"2-by"	Key
Key	Key	Key	"te k"

Алгоритм «четверти раунда»:

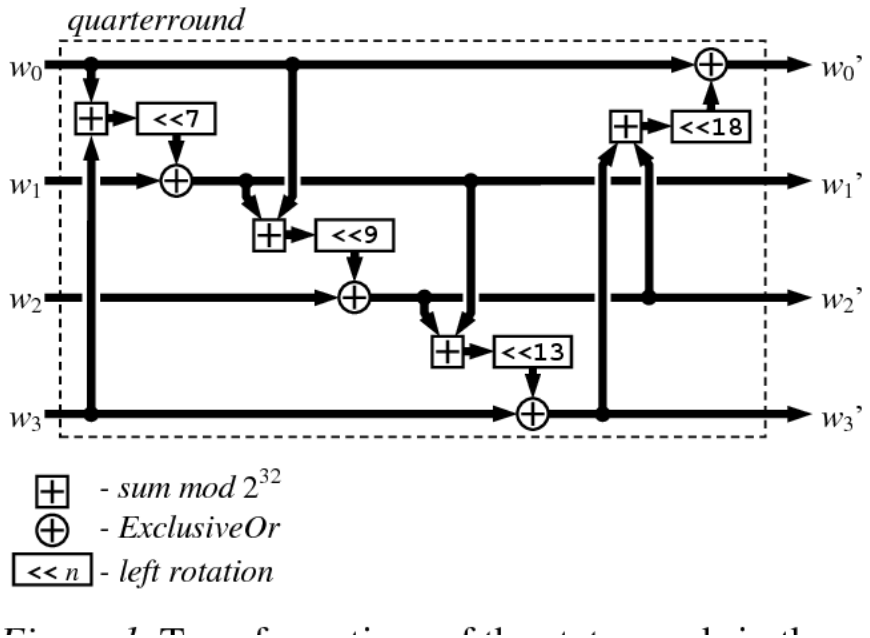
QR(a,b,c,d):

b ^= (a + d) <<< 7;

c ^= (b + a) <<< 9;

d ^= (c + b) <<< 13;

a ^= (d + c) <<< 18;



На нечётных раундах алгоритм применяется ко всем столбцам.

На чётных раундах – ко всем строкам.

Salsa20 и стойкость

Является ли Salsa20 стойким PRG? Никто не знает. (Как никто не знает возможно ли вообще построить стойкий PRG).

При использовании Salsa20 предполагается её стойкость на основе сложности **существующих практических атак**.

Salsa20/7, 128 бит (7 раундов, вместо 20) - 2^{102}

Salsa20/8, 256 бит (8 раундов, вместо 20) - 2^{240}

Salsa20/12, Salsa20/20 – не известны атаки лучше чем перебор ключа

Другие вариации XSalsa20, ChaCha20 (используется Google), XChaCha20.

Генерация случайных чисел

Как получать случайные данные для ключей?

- Использование внешних источников: метеоданные, интенсивность излучений итд.
- Использование аппаратных генераторов – на основе времени выполнения и частоты появления системных прерываний, текущей частоты процессора, времени чтения из памяти итд.
  - Примеры: /dev/random (Unix), RdRand (intel)
  - Важно использовать в коде криптографически стойкие классы генераторов вместо генераторов общего назначения, пример (Python): os.urandom() или secrets вместо random.randint())
  - Плохая идея реализовывать их самостоятельно

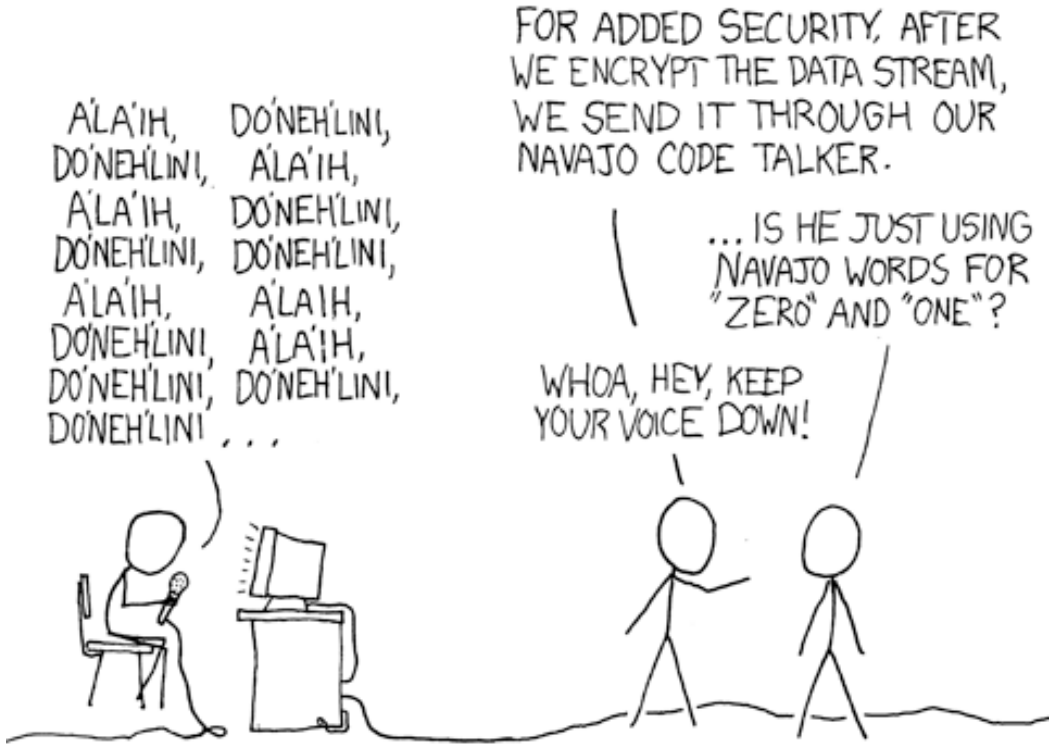
Главное правило реализации криптографии

Никогда не использовать собственные примитивы!

- Использовать только стандартные, широко распространённые примитивы
- Даже если вы уверены что ваши примитивы лучше
- Даже если вы опубликовали статью на eurocrypt
- Правило **Керкгоффса** – противник знает строение и функционирование криптосистемы. Неизвестны только ключи.

Главное правило реализации криптографии

- Не использовать security through obscurity
  - «очень» запутанный алгоритм не означает стойкий алгоритм
  - В местах где необходима безопасность необходимо использовать стойкие криптосистемы, а не то что «усложнит» жизнь противнику запутанностью.
  - Запутанность кода и алгоритма не обеспечивает защищенности



… но тогда я придумаю свой протокол!

Никогда не придумывать протоколы!

- Существует множество существующих стандартов, с большой вероятностью описывающих то, что вам нужно
  - RFC – интернет стандарты
  - …но даже в стандартах есть уязвимости

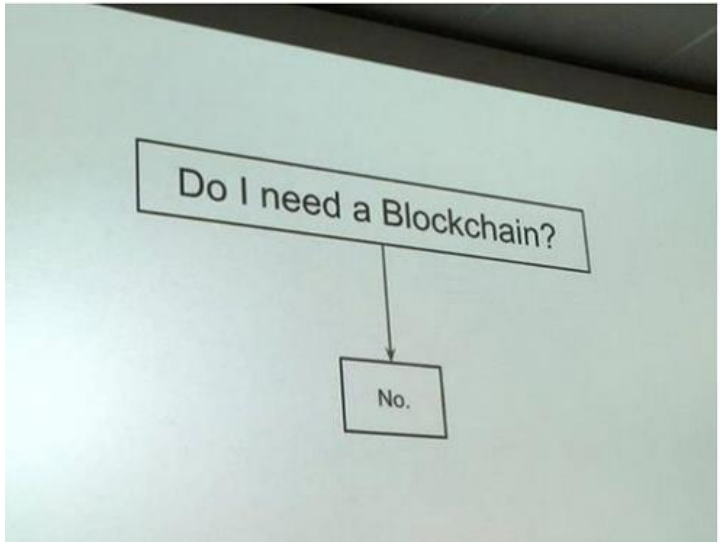
Никогда не придумывать собственные средства защиты информации!

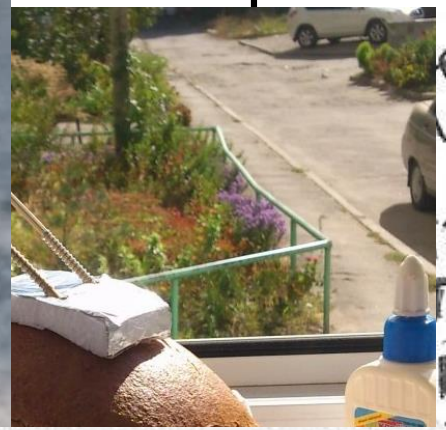
- Пр: Зашифрование жёстких дисков наверняка уже кем то описано и проанализировано, достаточно найти и использовать

… тут недавно выяснил что придумали новый алгоритм

Не нужно усложнять систему!

- Если у проблемы есть простое решение с использованием стандартных средств, не нужно использовать всю криптографию, о которой вы знаете, даже если вам она очень нравится.
- Не нужно везде использовать блокчейн.



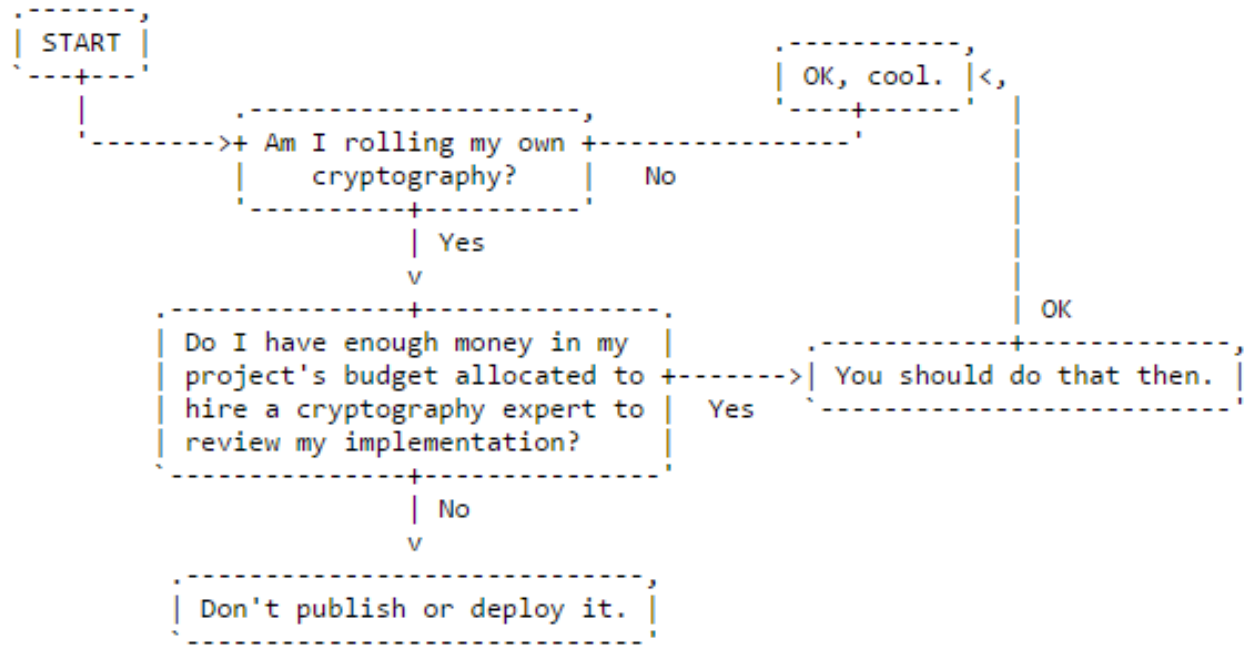



ВОТ ТАК, С ПОМОЩЬЮ НЕХИТРЫХ ПРИСПОСОБЛЕНИЙ
БУХАНКУ БЕЛОГО (ИЛИ ЧЕРНОГО) ХЛЕБА
МОЖНО ПРЕВРАТИТЬ В ТРОЛЛЕЙБУС...
НО ЗАЧЕМ?!
УМЕЛЫЕ РУКИ

…но тогда я хотя бы напишу реализацию!

Никогда не реализовывать криптографию!

- Необходимо использовать существующие распространённые реализации
- Даже если алгоритм прост и понятен – нужно использовать существующую реализацию
- … но даже в существующих распространённых реализациях могут быть критические ошибки



Уязвимости в коде (Apple “goto fail”, 2014)

Код проверки сертификатов при установлении SSL соединения.

```
. . .
hashOut.data = hashes + SSL_MD5_DIGEST_LEN;
hashOut.length = SSL_SHA1_DIGEST_LEN;
if ((err = SSLFreeBuffer(&hashCtx)) != 0)
    goto fail;
if ((err = ReadyHash(&SSLHashSHA1, &hashCtx)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &clientRandom)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
    goto fail;
    goto fail;  /* MISTAKE! THIS LINE SHOULD NOT BE HERE */
if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
    goto fail;

err = sslRawVerify(...);
. . .
```

Уязвимости в коде (Apple “goto fail”, 2014)

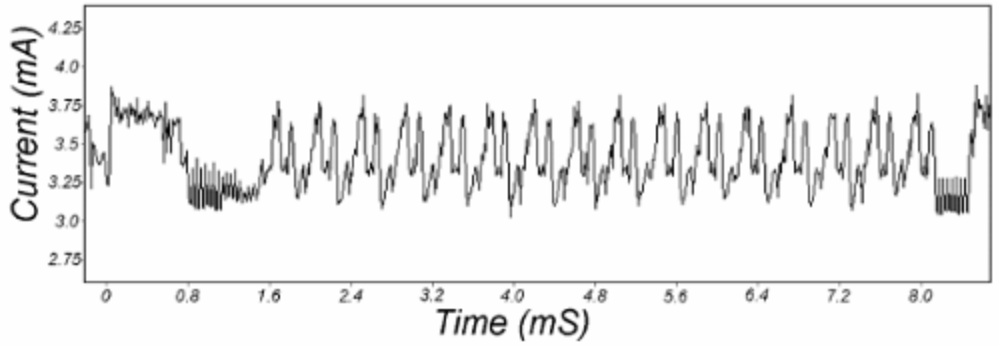
Лишний GoTo выполнялся безусловно, так как в if не стояли скобки.

Т.е. если функция SSLHashSHA1.update выполнялась без ошибки, остальные проверки не выполнялись, и проверка возвращала true.

```
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
  goto fail;
  goto fail;
... other checks ...
fail:
  ... buffer frees (cleanups) ...
  return err;
```

Уязвимости в других местах

- Уязвимости в стандартах: Пример WiFi: WEP, WPA-1, WPA-2, WPS, TLS
- Уязвимости в криптографических библиотеках: openssl
- Уязвимости в примитивах: SHA-1, DES, RC4, CSS
- Уязвимости при атаке по побочным каналам:



[Kocher, Jaffe, Jun, 1998]

О важности обновлений

- Если что то безопасно сегодня, не факт что будет завтра!
- Могут появляться новые атаки и могут обнаружится новые атаки, переводя криптосистему в класс нестойких.
- Регулярное обновление версий криптографических библиотек и оборудования.

mephi.ru Go!

mephi.ru IS VULNERABLE.

You can specify a port like this `example.com:4433`. 443 by default.

Тогда зачем этот курс? (в практическом смысле)

Основная цели:

- Знать достаточно чтоб сознательно избегать плохих конструкций и реализаций
  - Ничего не мешает вам посмотреть как работает чужая библиотека, как она генерирует ключи, какие режимы шифрования использует итд.
- Иметь возможность задать адекватные вопросы о криптосистемах (в том числе и гуглу)
- Понимать что нужно делать для решения практических задач, понимать как именно они были решены и с какими ограничениями.

Безопасное программирование

- Проверка входных значений (длины входов, типы, корректность)
  - Должна производиться по возможности в начале функции
  - Параноидальные проверки
- Криптографические методы должны быть вынесены в отдельные функции и модули
- Все внешние криптографические интерфейсы работают только с массивом байт
- Не использование «магических чисел» – все константы должны быть определены
- Не использовать «алгоритмы по умолчанию», т.е. явно задавать алгоритмы шифрования через параметры

Проверка входных данных

```
def decrypt(key, data, cipher_suite):
        if len(key) != AES_KEY_SIZE:
                raise Exception($‘invalid key size, expecting {AES_KEY_SIZE}’)
        if len(data) < NONCE_SIZE:
                raise Exception(‘invalid ciphertext length’)
        if cipher_suite = AES_CBC_WITH_CBC_MAC:
                return aes.cbc.decrypt(key, data)
        else if cipher_suite = AES_CTR_WITH_CBC_MAC :
                return aes.ctr.decrypt(key, data)
        else:
                raise Exception($‘cipher_suite {cipher_suite} is not supported’)
```

Отдельные функции, работа только с массивом байт

```
def generate_aes_key():
        return Crypto.random.getBytes(AES_KEY_SIZE)
--------------------
def encrypt_user_input(user_string):
        user_bytes = Encode.utf8.getBytes(user_string)
        key = generateAesKey()
        encrypted = encrypt(key, user_bytes , AES_CBC_WITH_CBC_MAC)
        encrypted_hex = Converter.toHex(encrypted)
        key_string = Converter.toHex(key)
        return key_string, encrypted_hex
```