CryptAttackTester: high-assurance attack analysis

https://cat.cr.yp.to/

Tung Chou [1] Daniel J. Bernstein [2]

[1] Academia Sinica, Taiwan

[2] University of Illinois at Chicago, USA and Ruhr University Bochum, Germany

CRYPTO, August 21, 2024

Motivation

We would like to compare efficiency of attacks. What is the problem?

- People are often talking about different cost models.
- Even worse, sometimes the cost model is not even well-defined.
- Also, claims for efficiency can be inaccurate due to errors and estimations.

CAT aims to solve the issues with a well-defined cost model + testing.

- two factors for efficiency: "cost" and success probability.

Cost model

- Each attack is considered as a **circuit**.
- Each circuit consists of a fixed sequence of **operations**.
- Each operation C_k is represented as $(\ell, F, i_0, \ldots, i_{\ell-1})$ where $\ell \in \{0, 1, 2\}$.
- F is any function that maps $\{0,1\}^\ell$ to $\{0,1\}$.
- C_k sets $x_k \leftarrow F(x_{i_0}, \ldots, x_{i_{\ell-1}})$.
- The cost is defined as the number of C_k's such that
  - $\ell = 2$ or
  - $\ell = 1$ and $F(x) = 1 - x$ (NOT).

The three main components

- $\mathcal{A}$: a piece of code that defines a family of attack circuits.
  - $\mathcal{A}$ generates the **exact** cost. Not overestimating. Not underestimating.
  - Success probability can be measured by running $\mathcal{A}$ many times.
- $\mathcal{C}$: a piece of code that predicts cost taken by $\mathcal{A}$.
  - Quickly generates output even when $\mathcal{A}$ takes, say, 2^{256} bit operations.
  - We gain confidence in $\mathcal{C}$ by seeing $\mathcal{C}$'s output matches $\mathcal{A}$'s output.
- $\mathcal{P}$: a piece of code that predicts the success probability of $\mathcal{A}$.
  - Quickly generates output even when $\mathcal{A}$ takes, say, 2^{256} bit operations.
  - We gain confidence in $\mathcal{P}$ by seeing $\mathcal{P}$'s output is close to measurement result.

Automatically counting bit operations in $\mathcal{A}$

```
class bit {
  int b; ⟵ private!

public:

  bit operator~() const { cost += bit_not_cost; ++numnot; return bit(b ^ 1); }
  bit operator^(const bit &c) const { cost += bit_xor_cost; ++numxor; return bit(b ^ c.b); }
  bit operator&(const bit &c) const { cost += bit_and_cost; ++numand; return bit(b & c.b); }
  bit operator|(const bit &c) const { cost += bit_or_cost; ++numor; return bit(b | c.b); }

  bit xnor(const bit &c) const { cost += bit_xnor_cost; ++numxnor; return bit(~(b ^ c.b)); }
  bit andn(const bit &c) const { cost += bit_andn_cost; ++numandn; return bit(b & ~c.b); }
  bit nand(const bit &c) const { cost += bit_nand_cost; ++numnand; return bit(~(b & c.b)); }
  bit orn(const bit &c) const { cost += bit_orn_cost; ++numorn; return bit(b | ~c.b); }
  bit nor(const bit &c) const { cost += bit_nor_cost; ++numnor; return bit(~(b | c.b)); }
};
```

Example code in $\mathcal{A}$: `half_adder`, `full_adder`

```
static inline void half_adder(bit &s, bit &c, bit a, bit b)
{
        s = a ^ b;
        c = a & b;
}

static inline void full_adder(bit &s, bit &c, bit a, bit b)
{
        bit t = (a ^ b);

        s = t ^ c;
        c = (a & b) | (c & t);
}
```

Example code in $\mathcal{A}$: bit_vector_add

```
static inline void bit_vector_add(vector<bit> &ret,
                                  vector<bit> a,
                                  vector<bit> b,
                                  bit c_in)
{
        assert(ret.size() >= a.size());
        assert(a.size() >= b.size());

        bit c = c_in;
        for (long long i = 0; i < b.size(); i++)
                full_adder(ret.at(i), c, a.at(i), b.at(i));

        for (long long i = b.size(); i < a.size(); i++)
                half_adder(ret.at(i), c, a.at(i), c);

        if (a.size() < ret.size())
                ret.at(a.size()) = c;
}
```

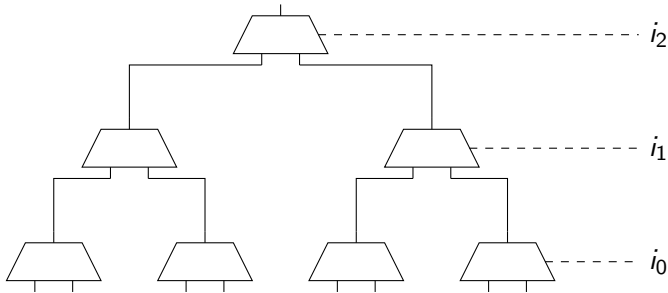
Example code in $\mathcal{A}$: bit_vector_add

```
static inline void bit_vector_add(vector<bit> &ret,
                                  vector<bit> a,
                                  vector<bit> b,
                                  bit c_in)
{
        assert(ret.size() >= a.size());
        assert(a.size() >= b.size());

        bit c = c_in;
        for (long long i = 0; i < b.size(); i++)
                full_adder(ret.at(i), c, a.at(i), b.at(i));

        for (long long i = b.size(); i < a.size(); i++)
                half_adder(ret.at(i), c, a.at(i), c);

        if (a.size() < ret.size())
                ret.at(a.size()) = c;
}
```

- $\mathcal{A}$ can be very complex: one can try to specify circuits in a CPU.

"RAM" read/write operations

- Given 8 bits $R[0], \ldots, R[7]$ and index $i = (i_2\ i_1\ i_0)_2$, obtain $R[i]$.



- The cost is linear in the number of bits N for the random access (N can change for each access).

Attacks included in CAT so far

Brute-force attack against AES-128

- Setting: given 2 plaintext-ciphertext pairs, find the key.
- CAT prediction: cost $2^{141.88}$, success probability 0.5.
- Appears to match NIST category I (2^{143} "classical gates").
- but NIST is considering the "gate count model".

Attacks included in CAT so far (cont.)

Information set decoding (ISD) algorithms against the syndrome decoding problem

- Including many variants: Leon, Stern, MMT, BJMM, ..., etc..
- Variants with the same number of levels of collision search are merged: we have ISD-0, ISD-1, and ISD-2.
- Many parameters and low-level optimizations.
- See `ia.cr/2023/940` for details.

Collision search

- Previous papers often use RAM operations to find collisions.
  - Extremely expensive for a circuit.

- To find collisions between lists L_1, L_2, $\texttt{CAT}$
  - sorts $L = L_1 + L_2$ using a **sorting network** and
  - checks all pairs $(L[i], L[i+d])$ for all $1 \leq d \leq \texttt{WI}$.
  - Note that we might miss collisions.
    ($\mathcal{P}$ needs to take this into account)

- We use Knuth's "merge exchange" sorting network.
  - Taking $\Theta(n \log^2 n)$ compare-and-swap operations.

Usage of queues

- Pairs that result in collisions are pushed into a queue of `QU` elements.
- Elements in the queue are processed and the queue is cleared periodically (specifically, after `PE` pairs are checked)
- A pair of the form $(L[i], L[i+d])$ will always be pushed into the queue, but it can be kicked out from the queue.
  (again, $\mathcal{P}$ needs to take this into account)
- To avoid bursts of collisions, pairs are checked in random order.

ISD results, in cost-probability ratio

isd	RE	ℓ	p	p'	p''	C	1284	3488	4608	6688	8192
2	1			2	1	1	72.59	158.59	201.70	278.31	315.21
2				2	1	1	70.90	156.26	198.21	275.14	312.21
2	1			4	2	1	70.99	158.62	199.22	278.45	309.19
2				4	2	1	70.95	158.46	198.90	278.12	309.06
2				6	3	1	71.07	154.21	200.67	272.72	307.34
2				8	4	1	72.45	154.17	195.37	270.42	305.78
2				10	5	1	75.35	152.45	193.88	267.79	303.99
2				12	6	1	82.39	151.78	192.78	266.34	301.82
2				14	7	1		150.84	191.95	264.40	299.18
2				16	8	1		150.91	191.56	263.57	296.93
2				18	9	1		150.59	190.62	260.44	296.45
2				20	10	1		151.46	191.41	261.13	294.64
2				22	11	1		151.77	190.50	260.40	292.46
2				24	12	1		152.91	191.18	259.02	291.14
2				26	13	1		154.04	190.55	259.44	290.83
2				28	14	1		156.08	192.21	258.54	289.99
2				30	15	1		159.08	193.16	258.69	289.31
2				32	16	1		165.51	194.12	258.29	287.21
2				34	17	1			195.99	257.36	288.00
2				36	18	1			197.89	258.34	287.50

https://cat.cr.yp.to/

NIST's comments

In the context of the NIST PQC Standardization Process, the version of the RAM model, where the operations being counted are "bit operations" that act on no more than 2 bits at a time and where each one-bit memory read or write is counted as one bit-operation, is sometimes referred to as the gate count model.

Additionally, while some submitters have rightly observed that many widely used cost models, such as the RAM model, underestimate the difficulty of certain memory intensive attacks, the comparative lack of published cryptanalysis using more realistic models may bring into question whether sufficient effort has been made to optimize the best-known attacks to perform well in these models.