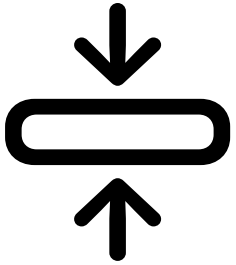
A performant Shamir Secret-Sharing Scheme implementation

by David Aimé Greven

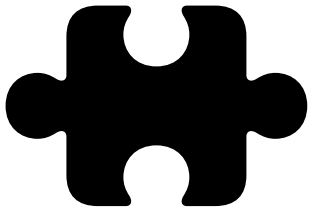
The **goal** is to create an optimized low-level Shamir Secret-Sharing Scheme implementation tuned for performance and space efficiency.

Principles & Properties

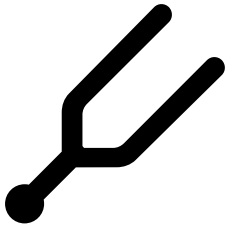
Secure



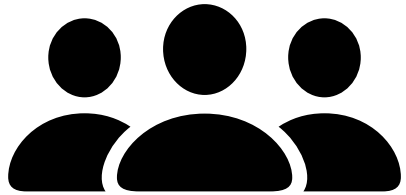
Minimal



Extensible



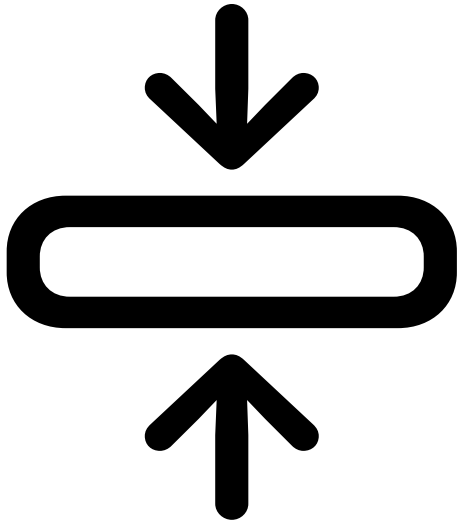
Dynamic



Flexible

Principles & Properties

Secure



Minimal

Encoding

$$b_0 \cdot x^7 + b_1 \cdot x^6 + b_2 \cdot x^5 + b_3 \cdot x^4 + b_4 \cdot x^3 + b_5 \cdot x^2 + b_6 \cdot x^1 + b_7$$

Implementation

```
extern inline char _mul(char l, char r) {
    char res = 0;
    for(char c = 0, hi = l & 0×80; c < 8;
        c++, r >>= 1, hi = l & 0×80) {
        if(r & 1 ≠ 0) res ^= l;
        l <<= 1;
        if(hi ≠ 0) l ^= 0×1b;
    }
    return res;
}
```

Baseline

```
extern inline char _mul(char l, char r) {
    if(l == 0 || r == 0) return 0;
    return _EXP[UINT(_LOG[UINT(l)])
                + UINT(_LOG[UINT(r)])];
}
```

Optimized

```c
char *shares(size_t secretc, char *secretv,
             rand_gen gen, size_t n, size_t k) {
    char *values = malloc(n * (secretc + 1));
    for(size_t i = 0; i < secretc; i++) {
        char *poly = _poly(gen, k - 1, secretv[i]);
        for(size_t x = 1; x ⩽ n; x++) {
            if(i == 0) values[(secretc + 1) * (x - 1)] = x;
            values[(secretc + 1) * (x - 1) + i + 1] =
                _eval(k, poly, x);
        }
        free(poly);
    }
    return values;
}
```

1. Allocate the shares array
2. Generate polynomial of degree k - 1
3. Evaluate the polynomial using x
4. Free the polynomial memory
5. Repeat steps 2-5 for all secret bytes

Sharing

```
char *shares(size_t secretc, char *secretv,
             rand_gen gen, size_t n, size_t k) {
    char *values = malloc(n * (secretc + 1));
    for(size_t i = 0; i < secretc; i++) {
        char *poly = _poly(gen, k - 1, secretv[i]);
        for(size_t x = 1; x ⩽ n; x++) {
            if(i == 0) values[(secretc + 1) * (x - 1)] = x;
            values[(secretc + 1) * (x - 1) + i + 1] =
                _eval(k, poly, x);
        }
        free(poly);
    }
    return values;
}
```

1. Allocate the shares array
2. Generate polynomial of degree k - 1
3. Evaluate the polynomial using x
4. Free the polynomial memory
5. Repeat steps 2-5 for all secret bytes

Sharing

```c
char *shares(size_t secretc, char *secretv,
             rand_gen gen, size_t n, size_t k) {
    char *values = malloc(n * (secretc + 1));
    for(size_t i = 0; i < secretc; i++) {
        char *poly = _poly(gen, k - 1, secretv[i]);
        for(size_t x = 1; x ⩽ n; x++) {
            if(i == 0) values[(secretc + 1) * (x - 1)] = x;
            values[(secretc + 1) * (x - 1) + i + 1] =
                _eval(k, poly, x);
        }
        free(poly);
    }
    return values;
}
```

1. Allocate the shares array
2. Generate polynomial of degree k - 1
3. Evaluate the polynomial using x
4. Free the polynomial memory
5. Repeat steps 2-5 for all secret bytes

Sharing

```
char *shares(size_t secretc, char *secretv,
             rand_gen gen, size_t n, size_t k) {
    char *values = malloc(n * (secretc + 1));
    for(size_t i = 0; i < secretc; i++) {
        char *poly = _poly(gen, k - 1, secretv[i]);
        for(size_t x = 1; x ⩽ n; x++) {
            if(i == 0) values[(secretc + 1) * (x - 1)] = x;
            values[(secretc + 1) * (x - 1) + i + 1] =
                _eval(k, poly, x);
        }
        free(poly);
    }
    return values;
}
```

1. Allocate the shares array
2. Generate polynomial of degree k - 1
3. Evaluate the polynomial using x
4. Free the polynomial memory
5. Repeat steps 2-5 for all secret bytes

Sharing

```c
char *shares(size_t secretc, char *secretv,
             rand_gen gen, size_t n, size_t k) {
    char *values = malloc(n * (secretc + 1));
    for(size_t i = 0; i < secretc; i++) {
        char *poly = _poly(gen, k - 1, secretv[i]);
        for(size_t x = 1; x ⩽ n; x++) {
            if(i == 0) values[(secretc + 1) * (x - 1)] = x;
            values[(secretc + 1) * (x - 1) + i + 1] =
                _eval(k, poly, x);
        }
        free(poly);
    }
    return values;
}
```

1. Allocate the shares array
2. Generate polynomial of degree k - 1
3. Evaluate the polynomial using x
4. Free the polynomial memory
5. Repeat steps 2-5 for all secret bytes

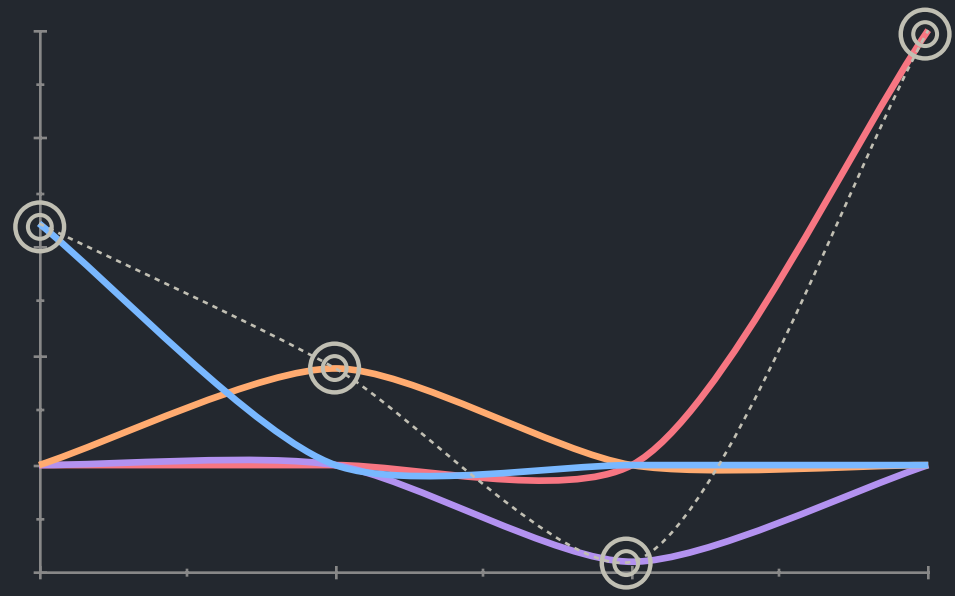
Sharing

```c
char *shares(size_t secretc, char *secretv,
             rand_gen gen, size_t n, size_t k) {
    char *values = malloc(n * (secretc + 1));
    for(size_t i = 0; i < secretc; i++) {
        char *poly = _poly(gen, k - 1, secretv[i]);
        for(size_t x = 1; x <= n; x++) {
            if(i == 0) values[(secretc + 1) * (x - 1)] = x;
            values[(secretc + 1) * (x - 1) + i + 1] =
                _eval(k, poly, x);
        }
        free(poly);
    }
    return values;
}
```

1. Allocate the shares array
2. Generate polynomial of degree k - 1
3. Evaluate the polynomial using x
4. Free the polynomial memory
5. Repeat steps 2-5 for all secret bytes

Sharing

```c
char _interpolate(size_t pointc, char *pointsv) {
    char y = 0, t = 1;
    for(size_t i = 0; i < pointc; i++, t = 1) {
        const char iX = pointsv[i * 2];
        const char iY = pointsv[(i * 2) + 1];
        for(size_t j = 0; j < pointc; j++) {
            const char jX = pointsv[j * 2];
            if(i ≠ j) t = _mul(t, _div(SUB(0, jX),
                                       SUB(iX, jX)));
        }
        y = ADD(y, _mul(t, iY));
    }
    return y;
}
```



Interpolation (Reconstruction)

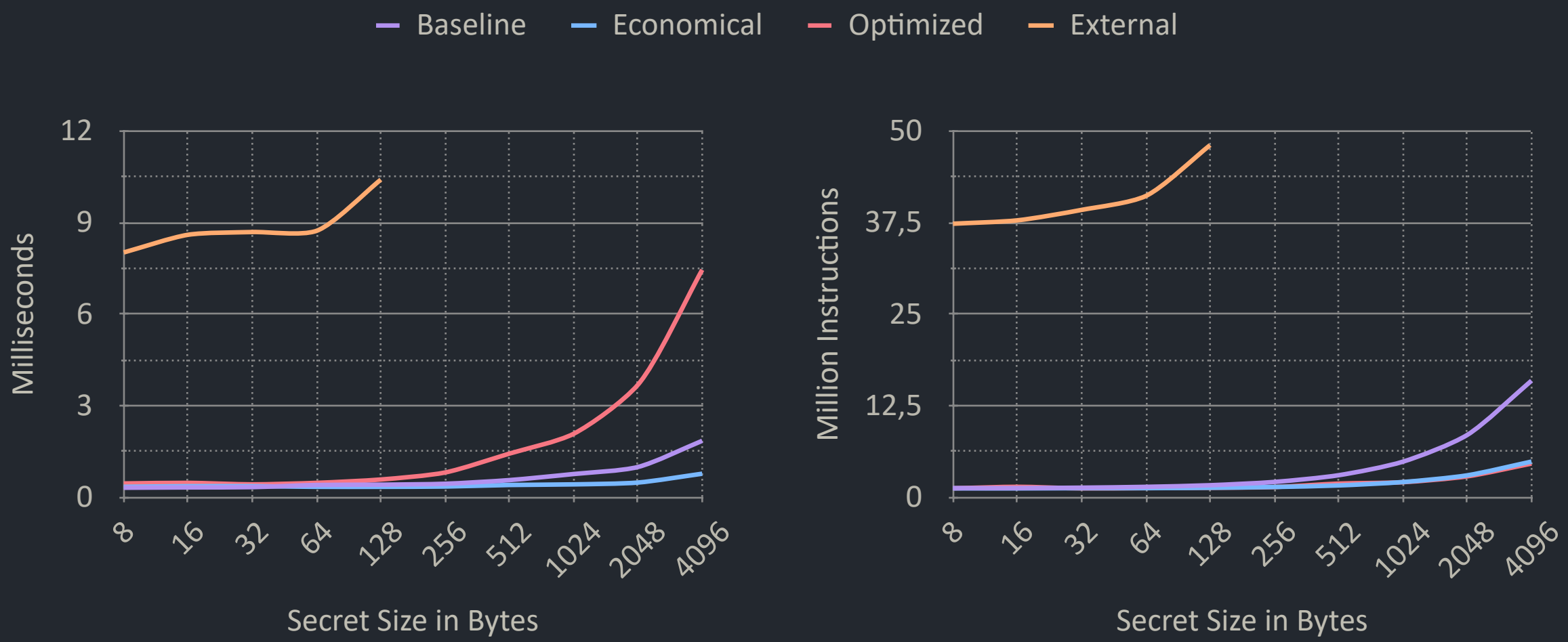
The graphic is for illustrative purposes only and shall not be interpreted as an accurate representation of polynomials over a finite field in a 2-dimensional plane.

```c
extern inline char *_rand_bytes_real_random(size_t len) {
    const size_t iterations = NEAREST(len, sizeof(uint64_t));
    uint64_t *arr = malloc(iterations);
    for (size_t i = 0; i < iterations / sizeof(uint64_t); i++) {
        unsigned int attempts = 10;
        do {
            unsigned char ok;
            uint64_t rand = 0;
            asm volatile ("rdrand %0; setc %1"
                : "=r" (*(&rand)), "=qm" (ok));
            if(ok) {
                arr[i] = rand;
                break;
            }
        } while(--attempts);
    }
    return (char*) arr;
}
```

```
    call    malloc@PLT
    movq    %rax, -16(%rbp)
    movq    $0, -32(%rbp)
    jmp .L10
.L14:
    movl    $10, -44(%rbp)
.L13:
    movq    $0, -40(%rbp)
#APP
# 36 "secret_sharing.c" 1
    rdrand %rax; setc %dl
# 0 "" 2
#NO_APP
    movq    %rax, -40(%rbp)
    movb    %dl, -45(%rbp)
    cmpb    $0, -45(%rbp)
```

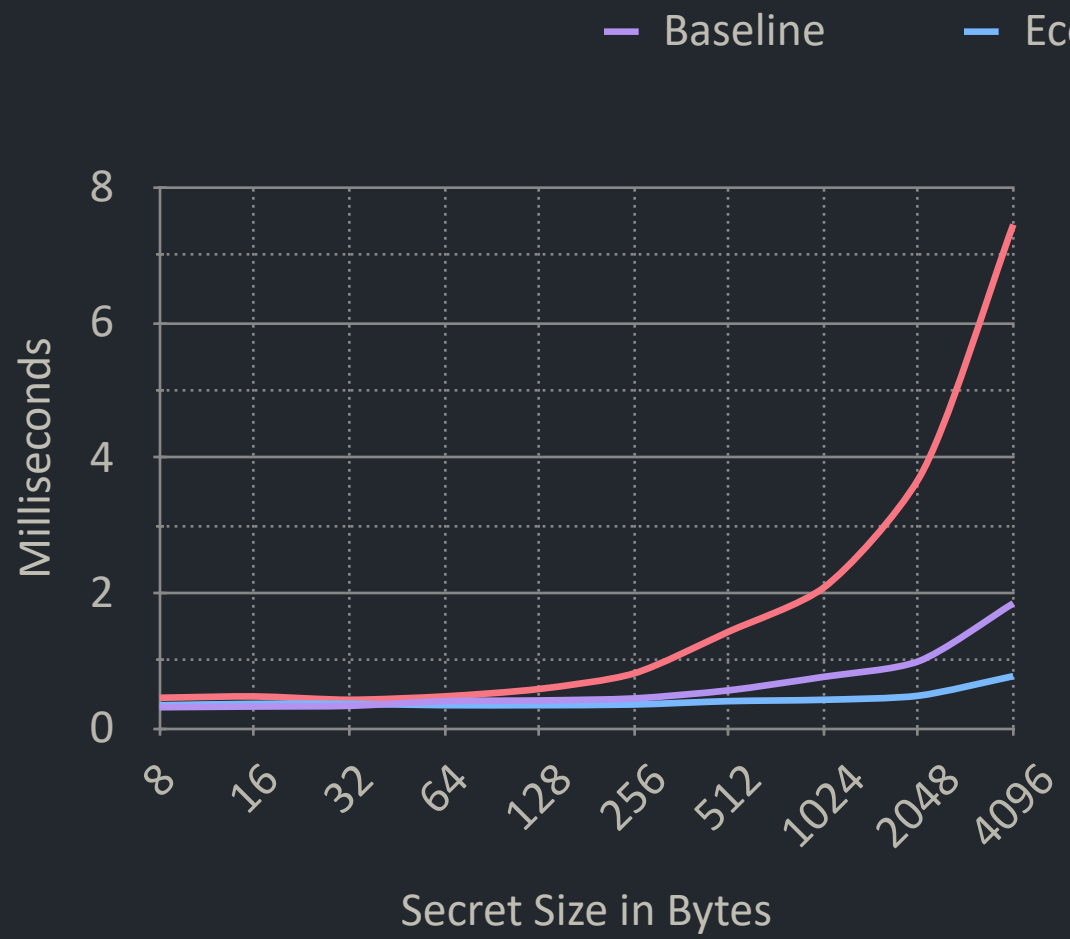
Random Number Generation

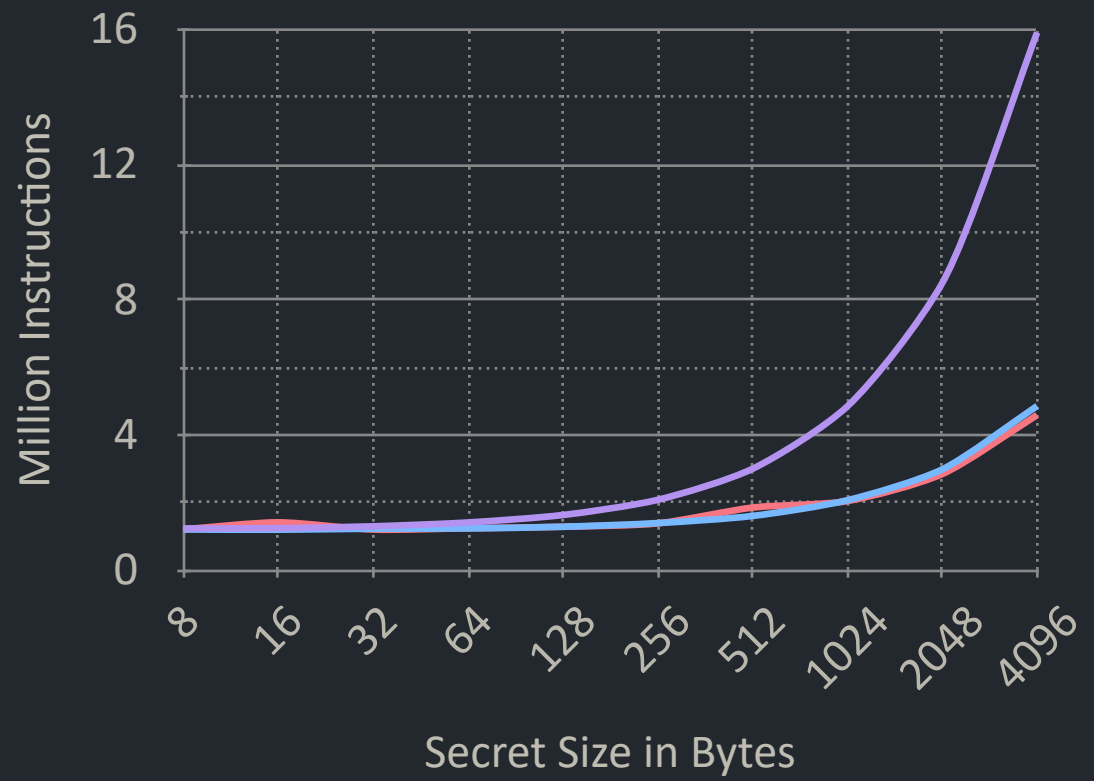
The presented slice of assembly code from the compilation output of the testing conducted in May 2022 may not be representative of tooling other than GCC 11.2.0-19ubuntu1, GNU Make 4.3, Python 2.7.18 and Ubuntu 22.04 LTS (Jammy Jellyfish) with Linux 5.15.0-27-generic amd64



Sharing Performance

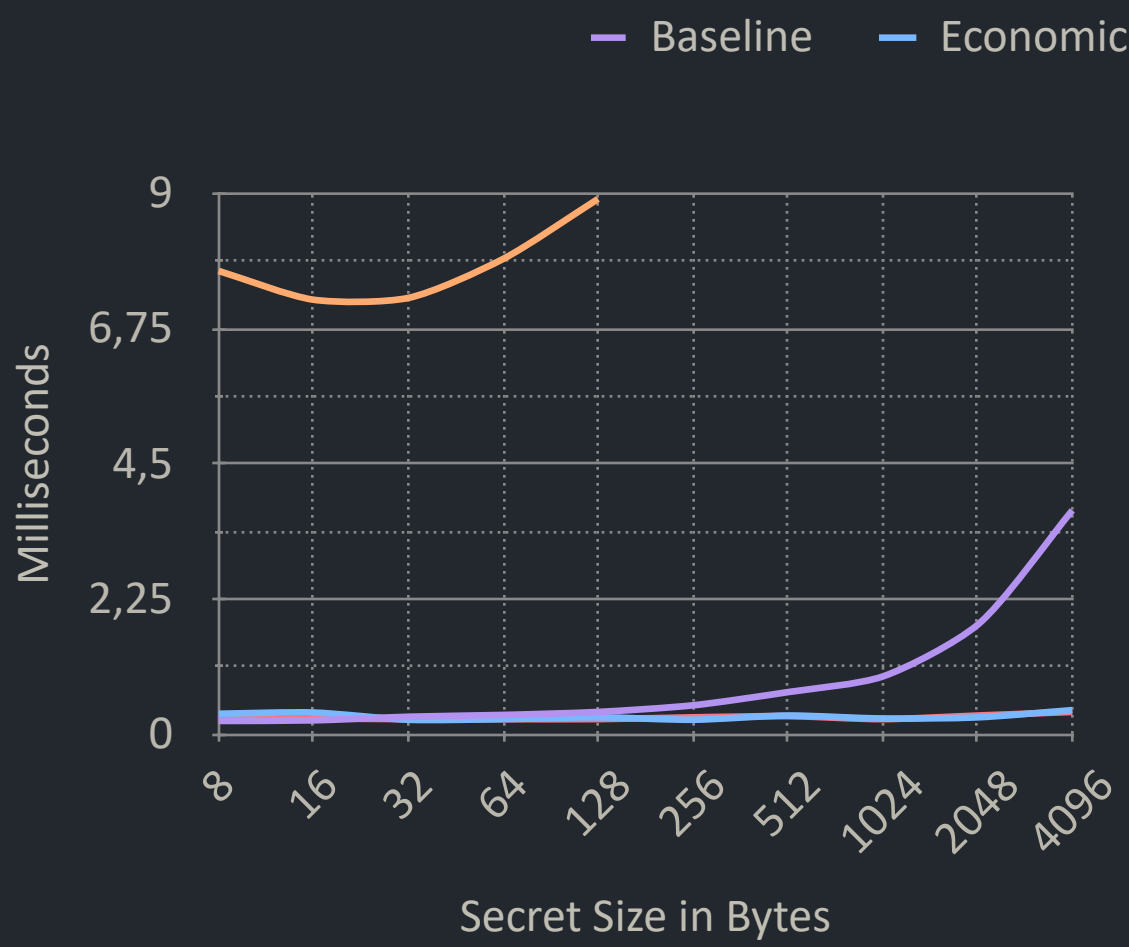
Testing conducted in May 2022 using GCC 11.2.0-19ubuntu1, GNU Make 4.3, Python 2.7.18 and Ubuntu 22.04 LTS (Jammy Jellyfish) with Linux 5.15.0-27-generic amd64 on Intel(R) Core(TM) i7-8565U CPU @ 4x 1.80 GHz + 8192 KB cache with 31GiB system memory

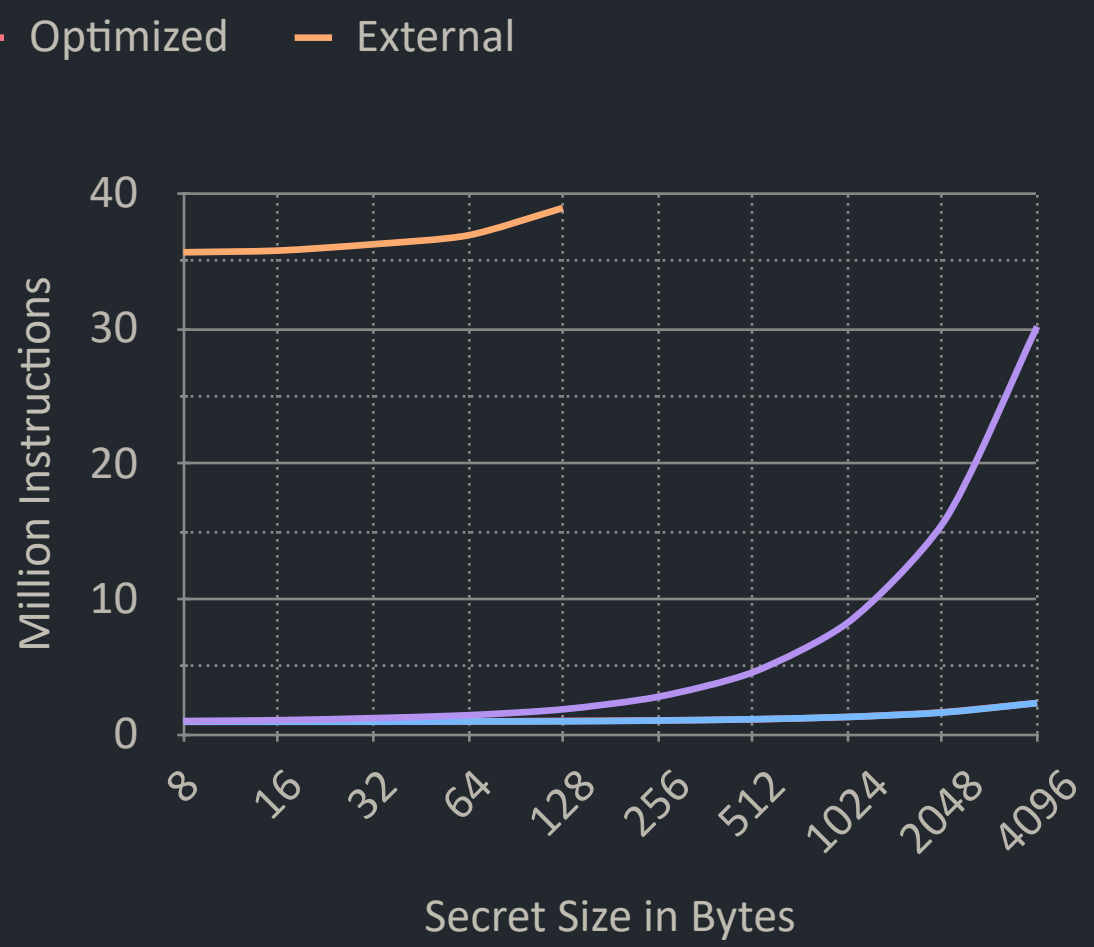




Sharing Performance

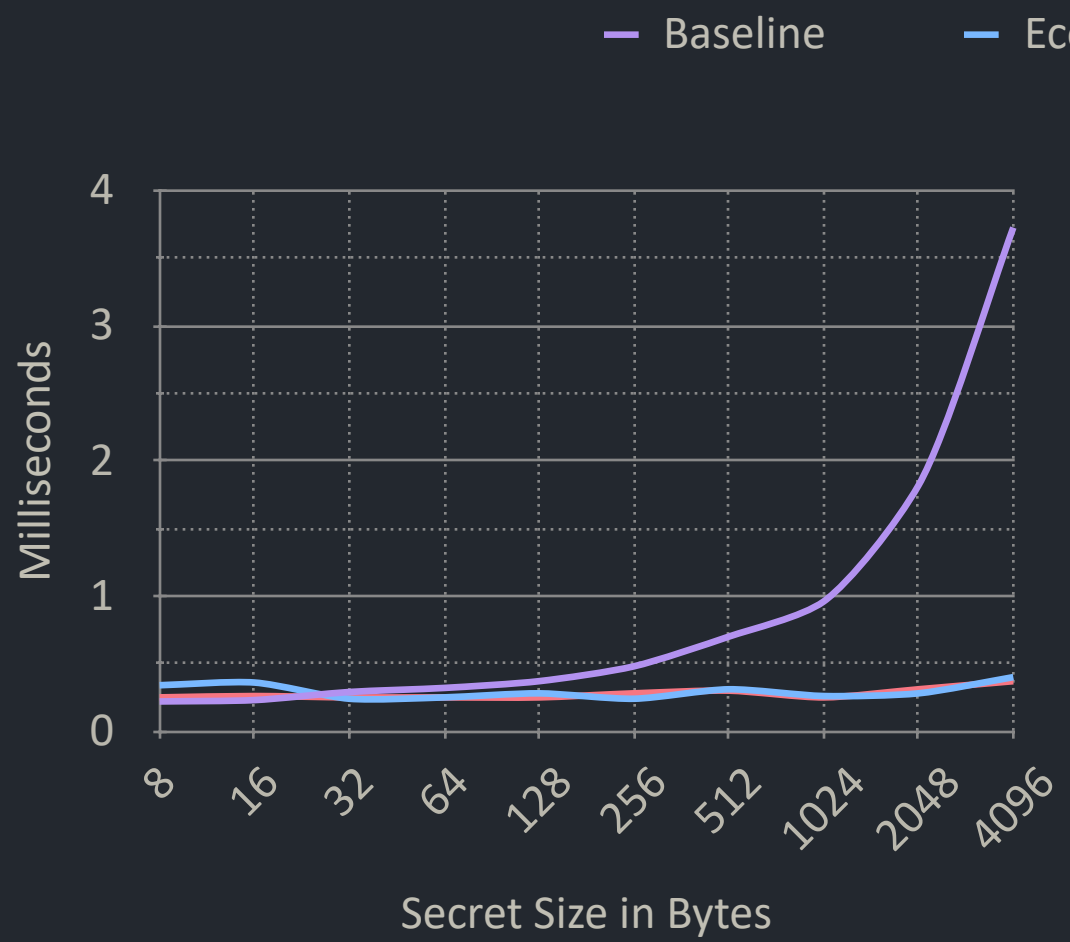
Testing conducted in May 2022 using GCC 11.2.0-19ubuntu1, GNU Make 4.3, Python 2.7.18 and Ubuntu 22.04 LTS (Jammy Jellyfish) with Linux 5.15.0-27-generic amd64 on Intel(R) Core(TM) i7-8565U CPU @ 4x 1.80 GHz + 8192 KB cache with 31GiB system memory

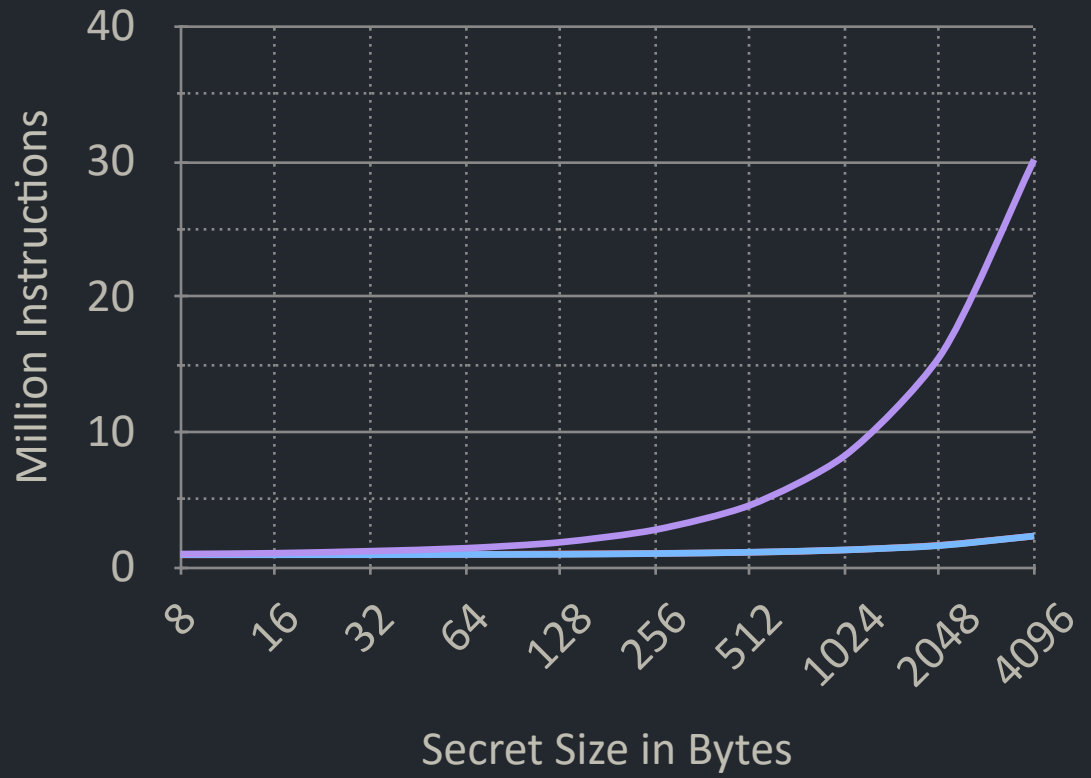




Recovering Performance

Testing conducted in May 2022 using GCC 11.2.0-19ubuntu1, GNU Make 4.3, Python 2.7.18 and Ubuntu 22.04 LTS (Jammy Jellyfish) with Linux 5.15.0-27-generic amd64 on Intel(R) Core(TM) i7-8565U CPU @ 4x 1.80 GHz + 8192 KB cache with 31GiB system memory





Recovering Performance

Testing conducted in May 2022 using GCC 11.2.0-19ubuntu1, GNU Make 4.3, Python 2.7.18 and Ubuntu 22.04 LTS (Jammy Jellyfish) with Linux 5.15.0-27-generic amd64 on Intel(R) Core(TM) i7-8565U CPU @ 4x 1.80 GHz + 8192 KB cache with 31GiB system memory

32x

Secret Throughput

Testing conducted in May 2022 using GCC 11.2.0-19ubuntu1, GNU Make 4.3, Python 2.7.18 and Ubuntu 22.04 LTS (Jammy Jellyfish) with Linux 5.15.0-27-generic amd64 on Intel(R) Core(TM) i7-8565U CPU @ 4x 1.80 GHz + 8192 KB cache with 31GiB system memory

Up to 8EB

Max Secret Size in Exabytes

Testing conducted in May 2022 using GCC 11.2.0-19ubuntu1, GNU Make 4.3, Python 2.7.18 and Ubuntu 22.04 LTS (Jammy Jellyfish) with Linux 5.15.0-27-generic amd64 on Intel(R) Core(TM) i7-8565U CPU @ 4x 1.80 GHz + 8192 KB cache with 31GiB system memory

Perfect Secrecy

Information-Theoretic Security

Minimality

Optimal Space Efficiency

Opportunities

Questions