Universität des Saarlandes
Lehrstuhl für Softwaretechnik
Prof. Andreas Zeller, Stephan Neuhaus
Design of Secure Software Systems
Solution 3

1 Mixing Functions

1.1 Statistical Distribution of 0 and 1 Bits

What is the expected number of 0 and 1 bits? If the probabilities p_0 and p_1 of 0 and 1 bits are equal, we must have $p_0 = p_1 = 0.5$, since $p_0 + p_1 = 1$. The expected number of 0 bits in a sample of N bits is therefore the same as the expected number of 1 bits and is $0.5N$.

What is the standard deviation? The number of one bits is k with probability $\binom{N}{k}(1/2)^k(1/2)^{N-k}$ for $0 \leq k \leq N$, so we have a binomial distribution with $p = q = 1/2$. Therefore, $\sigma = \sqrt{Npq} = \sqrt{0.25N} = 0.5\sqrt{N}$.

1.2 Experiments With Various Mixing Functions

The results of the tests are found in Table 1.

What can we see from this table? The two functions create numbers that are indistinguishable from random numbers when it comes to the average numbers and standard deviations of 0 and 1 bits. Still you know that the f function, being a LCPRNG, will not yield "good" random numbers.

The conclusion is therefore threefold:

- Both functions produce digests where the test yield excellent results.
- MD5 is still better than f as a mixing function, because f will produce numbers that are biased.
- If a test produces no failure, this doesn't mean that the results are good.

The program to compute the statistics is given in Appendix A.

2 Linear-Congruential Pseudo-Random Number Generators

2.1 Area of the Parallelepid

The area of a parallelogram defined by two vectors (x_1, y_1) and (x_2, y_2) is $|(x_1 - y_2)(x_2 - y_1)|$. (See, for example, `http://www-math.mit.edu/18.013A/chapter04/section01.html` for why this is true.) So the area of the parallelogram defined by the vectors $(X_i - X_0, X_{i+1} - X_1)$ and $(X_j - X_0, X_{j+1} - X_1)$ is $|(X_i - X_0)(X_{j+1} - X_1) - (X_{i+1} - X_1)(X_j - X_0)|$. But this is simply the parallelepid defined by (X_0, X_1), (X_i, X_{i+1}), (X_j, X_{j+1}), translated so that (X_0, X_1) lies in the origin.

Digest	Stats	μ	Pct.	σ
MD5	Measured 1	64.00	50.00%	5.66
	Measured 0	64.00	50.00%	5.66
	Expected	64.00	50.00%	5.66
f	Measured 1	16.00	50.00%	2.83
	Measured 0	16.00	50.00%	2.83
	Expected	16.00	50.00%	2.83

Table 1: Distribution of 0 and 1 bits for various mixing functions.

i	$X_i - X_0$	$X_{i+2} - X_1$	$X_{i+1} - X_0$	$X_{i+1} - X_1$	$d(i,j)$	gcd
1	253	-113	129	-124	12593	12593
2	129	-29	140	-113	12079	257
3	140	-252	224	-29	28784	257
4	224	-226	1	-252	50372	257
5	1	-191	27	-226	5911	257

Table 2: Computing the modulus

2.2 Cracking a LCPRNG

The programs used to compute the modulus and the extended Euclidean algorithm are given in Appendices B and C.

Finding the Modulus. If we run the program given in Appendix B, we get the results in Table 2. From these it looks as if $m = 257$.

Finding the Multiplier. To do that, we must first verifying that $\gcd(X_1 - X_0, m) = 1$, which is easy, since 257 is a prime. Now we find x and y with $x(X_1 - X_0) + ym = 1$. Putting the numbers into the extended Euclidean Algorithm gives $x = 64$ and $y = -63$, so $a \equiv x(X_2 - X_1) \equiv 64 \cdot (130 - 254) \equiv 31 \pmod{257}$.

Finding the Increment. We find c as $X_1 - aX_0 \equiv 223 \pmod{257}$.

Test and Conclusion. Computing the first few numbers in the sequence, we find complete agreement.

3 Modulo Operations In Your Head

3.1 Modulo Base-Minus-One

Prove the Formula. The proof is quite simple: Let $a = (a_{n-1}, \ldots, a_0)_{10}$ be a number written in base 10. In other words, $a = \sum_{0 \le k < n} a_k 10^k$, where each a_k is an integer between 0 and 9. Then

$$\begin{aligned}
a &\equiv \sum_{0 \le k < n} a_k 10^k \\
&\equiv \sum_{0 \le k < n} (a_k 10^k \bmod 9) \\
&\equiv \sum_{0 \le k < n} \big(a_k (9+1)^k \bmod 9\big) \\
&\equiv \sum_{0 \le k < n} a_k \left(\sum_{0 \le j \le k} \binom{k}{j} 9^j \bmod 9 \right) \\
&\equiv \sum_{0 \le k < n} a_k \left(1 + 9 \sum_{1 \le j \le k} \binom{k}{j} 9^{j-1} \bmod 9 \right) \\
&\equiv \sum_{0 \le k < n} a_k (1 + 0) \\
&\equiv \sum_{0 \le k < n} a_k
\end{aligned}$$

All equivalences above are modulo 9.

A somewhat simpler proof goes as follows: first, we prove that $10^k \bmod 9 = 1$ for all integers $k \geq 0$ by induction. For $k = 0$, we have $10^0 = 1$ and that is also 1 modulo 9. Now assume that $10^j \bmod 9 = 1$ for $0 \leq j < k$. Now $10^k \equiv 10 \cdot 10^{k-1} \equiv 10 \cdot 1 \equiv 10 \equiv 1 \pmod 9$. Then we write $a = \sum_{0 \leq k < n} a_k 10^k$ and see immediately that the 10^k are all equivalent to 1 modulo 9, q.e.d.

Extend to Arbitrary Bases. The proof works just as well for arbitrary bases b if you substitute $b - 1$ for every 9.

Compute Residues Modulo 31. If you have the number written out in binary, mark off every five bits, starting from the right. Treat these as digits to the base $b = 32$ and apply the $b - 1$ rule.

For example, $(35264)_{10} = (89\text{c}0)_{16} = (1000100111000000)_2$. Marking off every five bits gives $(1\ 00010\ 01110\ 00000)_{32} = (1\ 2\ 14\ 0)_{32}$, so $35264 \bmod 31 \equiv 0 + 14 + 2 + 1 \equiv 17 \equiv 18$, and indeed, $35264 = 1137 \cdot 31 + 17$.

3.2 Modulo Base-Plus-One

Prove the Rule. The proof is again quite simple: Let $a = (a_{n-1}, \ldots, a_0)_{10}$ be a number written in base 10, i.e., $a = \sum_{0 \leq k < n} a_k 10^k$, where each a_k is an integer between 0 and 9. Then

$$
\begin{aligned}
a &\equiv \sum_{0 \leq k < n} a_k 10^k \\
&\equiv \sum_{0 \leq k < n} (a_k 10^k \bmod 11) \\
&\equiv \sum_{0 \leq k < n} \bigl(a_k (11-1)^k \bmod 11\bigr) \\
&\equiv \sum_{0 \leq k < n} a_k \left(\sum_{0 \leq j \leq k} \binom{k}{j} 11^j (-1)^{k-j} \bmod 11 \right) \\
&\equiv \sum_{0 \leq k < n} a_k \left((-1)^k + 11 \sum_{1 \leq j \leq k} \binom{k}{j} 11^{j-1} (-1)^{k-j} \bmod 11 \right) \\
&\equiv \sum_{0 \leq k < n} a_k \bigl((-1)^k + 0\bigr) \\
&\equiv \sum_{0 \leq k < n} (-1)^k a_k
\end{aligned}
$$

All equivalences above are modulo 11.

There is a somewhat simpler proof here, too. First, we prove that $10^k \bmod 11 = (-1)^k$ for all integers $k \geq 0$ by induction. For $k = 0$, we have $10^0 = 1$ and that is also 1 modulo 11. Now assume that $10^j \bmod 11 = (-1)^k$ for $0 \leq j < k$. Now $10^k \equiv 10 \cdot 10^{k-1} \equiv 10 \cdot (-1)^{k-1} \equiv (-1) \cdot (-1)^{k-1} \equiv (-1)^k \pmod{11}$. (This works because $10 \equiv -1 \pmod{11}$) Then we write $a = \sum_{0 \leq k < n} a_k 10^k$ and see immediately that the 10^k are all equivalent to $(-1)^k$ modulo 11, q.e.d.

Extend to Arbitrary Bases. The proof works just as well for arbitrary bases b if you substitute $b + 1$ for every 11.

Compute Residues Modulo 65537. If you have the number written out in hex, mark off every sixteen bits, starting from the right. Treat these as digits to the base $b = 65536$ and apply the $b + 1$ rule.

For example, $(1892678289)_{10} = (70\text{cffa}91)_{16}$. Therefore, $1892678289 \bmod 65537 = (\text{fa}91)_{16} - (70\text{cf})_{16} = (35266)_{10}$, and indeed, $1892678289 = 28879 \cdot 65537 + 35266$..

A Program to Compute Mixing Function Statistics

```
#include <openssl/md5.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>

static int count_one_bits(const unsigned char* buf, int size) {
  int ret = 0;
  const unsigned char* p;

  for (p = buf; p < buf + size; p++) {
    unsigned char byte = *p;
    while (byte != 0) {
      if (byte & 1)
        ret++;
      byte >>= 1;
    }
  }

  return ret;
}

static int one_bits_in_md5_digest(unsigned int n) {
  unsigned char digest[MD5_DIGEST_LENGTH];
  MD5_CTX md5;
  unsigned char buf[sizeof(unsigned int)];
  int i;

  MD5_Init(&md5);
  for (i = 0; i < sizeof(unsigned int); i++) {
    buf[i] = n & 0xff;
    n >>= 8;
  }
  MD5_Update(&md5, buf, sizeof(buf));
  MD5_Final(digest, &md5);

  return count_one_bits(digest, MD5_DIGEST_LENGTH);
}

static int one_bits_in_pseudorandom_digest(unsigned int n) {
  unsigned int d = 314159269*n;

  return count_one_bits((const unsigned char*) &d, sizeof(unsigned int));
}

struct digest_stats_t {
  double r_average_ones;       /* Running average for one bits */
  double average_ones;         /* Final average for one bits */
  double r_average_zeros;      /* Running average for zero bits */
  double average_zeros;        /* Final average for zero bits */
  double r_variance_ones;      /* Running variance for one bits */
  double variance_ones;        /* Final variance for one bits */
  double r_variance_zeros;     /* Running variance for zero bits */
  double variance_zeros;       /* Final variance for zero bits */
  unsigned int n;              /* Number of samples processed */
};

static void stats_init(struct digest_stats_t *stats) {
  stats->r_average_ones = 0.0;
  stats->r_average_zeros = 0.0;
  stats->r_variance_zeros = 0.0;
  stats->r_variance_ones = 0.0;
  stats->n = 0;
}
```

```c
static void stats_update(struct digest_stats_t *stats, unsigned int one_bits,
             unsigned int zero_bits) {
  stats->n++;
  if (stats->n == 1) {
    stats->r_average_ones = one_bits;
    stats->r_variance_ones = 0.0;
    stats->r_average_zeros = zero_bits;
    stats->r_variance_zeros = 0;
  } else {
    double old_r_average_ones = stats->r_average_ones;
    double old_r_average_zeros = stats->r_average_zeros;

    stats->r_average_ones += (one_bits - old_r_average_ones)/stats->n;
    stats->r_variance_ones += (one_bits - old_r_average_ones)
      *(one_bits - stats->r_average_ones);

    stats->r_average_zeros += (zero_bits - old_r_average_zeros)/stats->n;
    stats->r_variance_zeros += (zero_bits - old_r_average_zeros)
      *(zero_bits - stats->r_average_zeros);
  }
}

static void stats_final(struct digest_stats_t *stats) {
  stats->average_ones = stats->r_average_ones;
  stats->average_zeros = stats->r_average_zeros;
  stats->variance_ones = stats->r_variance_ones/(stats->n - 1.0);
  stats->variance_zeros = stats->r_variance_zeros/(stats->n - 1.0);
}

int main() {
  struct digest_stats_t md5_stats;
  struct digest_stats_t pr_stats; /* pr = pseudorandom */
  unsigned int n;

  stats_init(&md5_stats);
  stats_init(&pr_stats);

  for (n = 0; n < 9999999; n++) {
    unsigned int ones_md5 = one_bits_in_md5_digest(n);
    unsigned int ones_pr = one_bits_in_pseudorandom_digest(n);

    stats_update(&md5_stats, ones_md5, 8*MD5_DIGEST_LENGTH - ones_md5);
    stats_update(&pr_stats, ones_pr, 8*sizeof(unsigned int) - ones_pr);
  }

  stats_final(&md5_stats);
  stats_final(&pr_stats);

  printf("MD5\n");
  printf("\t1 bits: Measured Average %5.2f (%5.2f%%), Std. Dev: %5.2f\n",
         md5_stats.average_ones,
         100.0*md5_stats.average_ones/(8*MD5_DIGEST_LENGTH),
         sqrt(md5_stats.variance_ones));
  printf("\t0 bits: Measured Average %5.2f (%5.2f%%), Std. Dev: %5.2f\n",
         md5_stats.average_zeros,
         100.0*md5_stats.average_zeros/(8*MD5_DIGEST_LENGTH),
         sqrt(md5_stats.variance_zeros));
  printf("\t        Expected Average %5.2f (%5.2f%%), Std. Dev: %5.2f\n",
         8*MD5_DIGEST_LENGTH*0.5,
         100.0*(8*MD5_DIGEST_LENGTH*0.5)/(8*MD5_DIGEST_LENGTH),
         sqrt(8*MD5_DIGEST_LENGTH*0.5*0.5));

  printf("PR\n");
  printf("\t1 bits: Measured Average %5.2f (%5.2f%%), Std. Dev: %5.2f\n",
         pr_stats.average_ones,
         100.0*pr_stats.average_ones/(8*sizeof(unsigned int)),
         sqrt(pr_stats.variance_ones));
```

```
  printf("\t0 bits: Measured Average %5.2f (%5.2f%%), Std. Dev: %5.2f\n",
         pr_stats.average_zeros,
         100.0*pr_stats.average_zeros/(8*sizeof(unsigned int)),
         sqrt(pr_stats.variance_zeros));
  printf("\t        Expected Average %5.2f (%5.2f%%), Std. Dev: %5.2f\n",
         8*sizeof(unsigned int)*0.5,
         100.0*(8*sizeof(unsigned int)*0.5)/(8*sizeof(unsigned int)),
         sqrt(8*sizeof(unsigned int)*0.5*0.5));

  return 0;
}
```

B Program Used to Compute the Modulus

```
#include <stdio.h>
#include <stdlib.h>

static unsigned int X[] = {1, 254, 130, 141, 225, 2, 28, 63, 120};

/* Given two positive integers m and n, return gcd(m,n) */
int gcd(int m, int n) {
  int r = m % n;
  while (r != 0) {
    m = n;
    n = r;
    r = m % n;
  }
  return n;
}

int main() {
  int i;
  int last_gcd = -1;

  for (i = 1; i < 6; i++) {
    int x1 = X[i] - X[0];
    int x2 = X[i+2] - X[1];
    int x3 = X[i+1] - X[0];
    int x4 = X[i+1] - X[1];
    int d = abs(x1*x2 - x3*x4);

    if (last_gcd == -1)
      last_gcd = d;
    else
      last_gcd = gcd(last_gcd, d);

    printf ("%d & %d & %d & %d & %d & %d & %d\n",
            i, x1, x2, x3, x4, d, last_gcd);
  }
  return 0;
}
```

C Program Used to Compute Extended Euclidean Algorithm

```
#include <stdio.h>
#include <stdlib.h>

int egcd(int* a, int* b, int m, int n) {
  int a_prime = 1, b_prime = 0;
```

```
  int c = m, d = n;
  int q = c / d, r = c % d;

  *b = 1; *a = 0;

  /* am + bn = d; a_prime m + b_prime n = c = qd + r */
  while (r != 0) {
    int t;

    c = d; d = r;
    t = a_prime; a_prime = *a; *a = t - q*(*a);
    t = b_prime; b_prime = *b; *b = t - q*(*b);
    q = c / d; r = c % d;
    /* am + bn = d; a_prime m + b_prime n = c = qd + r */
  }

  /* am + bn = d = gcd(m, n) */
  return d;
}

int main(int argc, const char* argv[]) {
  int m = atoi(argv[1]);
  int n = atoi(argv[2]);
  int x, y, gcd;

  gcd = egcd(&x, &y, m, n);

  printf("%d*%d + %d*%d = %d (%d)\n", x, m, y, n, gcd,
         x*m+y*n);

  return 0;
}
```