Rolling up sleeves when subversion's in a field?

Daniel R. L. Brown

danibrown@blackberry.com

December 15, 2020

Abstract

A nothing-up-my-sleeve number is a cryptographic constant, such as a field size in elliptic curve cryptography, with qualities to assure users against subversion of the number by the system designer. A number with low Kolmogorov descriptional complexity resists being subverted to the extent that the speculated subversion would leave a trace that cannot be hidden within the short description.

The roll programming language, a version of Godel's 1930s definition of computability, can somewhat objectively quantify low descriptional complexity, a nothing-up-my-sleeve quality, of a number. For example, curves NIST-P-256, Curve25519, and NIST-P-521 have fields sizes with roll programs of 112, 84, and 63 words (respectively).

Field size	Program	Words	Hours
$(2^{127}-1)^2$	Table 7	58	1
$2^{521}-1$	Table 2	63	2
2^{283}	Table 5	68	2
$8^{91}+5$	Table 6	68	10
$2^{336}-3$	Table 9	79	1
$2^{255}-19$	Table 4	84	1
$2^{256}-2^{224}+2^{192}+2^{96}-1$	Table 3	112	1
$2^{256}-2^{32}-977$	Table 8	127	1

Table 1: Shortest roll programs found (in roundly estimated time spent code golfing) for some especially efficient, previously proposed ECC field sizes

1 Early estimates in roll complexity

The roll programming language is defined in §2, but the listed roll programs aim to be almost self-explanatory, to serve as a primer on the roll programming language.

The purpose of the listed programs is to provide preliminary code golf scores of proposed field sizes in elliptic curve cryptography, towards eventually quantifying their nothing-up-my-sleeve quality.

1.1 Mersenne prime $2^{521} - 1$

The NIST-recommended elliptic curve P-521 uses $2^{521} - 1$ as its field size. The curve P-521, or at least its field size, is popular in some circles, but is not actually widely used.

Robinson discovered that $2^{521} - 1$ is prime in 1950, during basic number theoretic research on Mersenne primes, using the SWAC, a vacuum tube computer at an NBS (now NIST), making the historical first machine-aided prime number record. In dramatic terms, $2^{521} - 1$ symbolizes the dawn of the digital computer age in mathematics. Its importance as a number pre-dates elliptic curve cryptography.

```
2^521-1 subs 521 in 2^-1
521 subs 65 in *8+1
65 subs 8 in *8+1
8 subs 1 in *8
2^-1 roll *2+1 up 0
*2+1 subs *2 in +1
*8+1 subs *8 in +1
*8 subs *4 in *2
*4 subs *2 in *2
*2 roll +2 up 0
1 subs in +2
0 subs in +1
+2 subs +1 in +1
```

Table 2: A 63-word roll program for $2^{521} - 1$

Table 2 lists a roll program implementing a constant function that returns $2^{521} - 1$ on any input.

A previous version of this report listed a 5 word longer roll program (of 68 words). The five words were eliminated by a code golfing effort re-used from the roll programming for $8^{91} + 5$.

1.2 NIST prime P-256

```
2^224(2^32-1)+2^192+2^96-1 subs
2^224(2^32-1)+2^192 2^96-1 in +
2^224(2^32-1)+2^192 subs
192 2^32(2^32-1)+1 in 2^a*b
2^32(2^32-1)+1 subs 2^32(2^32-1) in +1
2^32(2^32-1) subs 32 2^32-1 in 2^a*b
2^96-1 subs 96 in 2^-1
2^32-1 subs 32 in 2^-1
192 subs 96 in *2
96 subs 64 32 in +
64 subs 32 in *2
32 subs 31 in +1
31 subs 5  in 2^-1
5 subs  3  in +2
3 subs  1  in +2
2^a*b roll *2 up a
2^-1 roll *2+1 up 0
*2+1 roll +2 up 1
1    subs    in +2
*2   roll +2 up 0
+    roll +1  up a
a    roll +1  up 0
+2   subs +1 in +1
0    subs    in +1
```

Table 3: A 112-word roll program for P-256

Table 3 lists a roll program that computes the prime field size used for NIST curve P-256. Many web sites now connect to web browsers using elliptic curve Diffie–Hellman with curve P-256.

The prime from P-256 is often called a Solinas prime, meaning a sum of signed version of powers of 2^{32} aiming to have efficent modular reduction on

32-bit machines.

1.3 Prime $2^{255} - 19$

```
2^255-19 subs 2^254-10 in *2+1
2^254-10 subs 2^253-5  in *2
2^253-5  subs 2^252-3  in *2+1
2^252-3  subs 2^251-2  in *2+1
2^251-2  subs 2^250-1  in *2
2^250-1  subs 250      in 2^-1
250 subs 125 in *2
125 subs  62 in *2+1
 62 subs  31 in *2
 31 subs   5 in 2^-1
  5 subs   2 in *2+1
  2 subs   0 in +2
2^-1 roll *2+1 up 0
*2+1 subs *2 in +1
*2   roll +2 up 0
+2   subs +1 in +1
0    subs    in +1
```

Table 4: An 84-word roll program for $2^{255} - 19$

Bernstein's public domain elliptic curve Curve25519 is incorporated into TLS 1.3 and various other systems. Its field size is the prime $2^{255} - 19$. Table 4 lists a roll program implementing $2^{255} - 19$.

Further code golfing may find a shorter program than the one in Table 4, as Table 4 uses the word `roll` only twice, but the shorter program Table 7 uses it four times.

1.4 Composite 2^{283}

The field size 2^{283} is a composite number used by the (formerly?) NIST-recommended curves B-283 and K-283. The curve K-283 is used in a few real-world applications, but recent advances in the elliptic curve discrete logarithm problem have raised concern about its security. The number 2^{283} can be computed by the roll program in Table 5.

```
2^283 subs 283 in 2^
283 subs 275 in +8
275 subs 273 in +2
273 subs 17 in *16+1
17  subs  1 in *16+1
*16+1 roll +16 up 1
+16 subs +8 in +8
+8  subs +4 in +4
+4  subs +2 in +2
2^ roll *2 up  1
1 subs  in +2
*2 roll +2 up  0
+2 subs +1 in +1
0 subs   in +1
```

Table 5: A 68-word program to compute 2^{283}

A previous version of this report listed a 10 word longer program (78 words) for 2^{283}. The newer shorter version leverages the better code golfing used for $8^{91}+5$ discussed in the next section.

1.5 Prime $8^{91}+5$

Field size $8^{91}+5$ was proposed in a previous IACR eprint of the author, for of its intuitively low Kolmogorov descriptional complexity (via the six character expression `8^91+5`) and because of its efficiencies (simple and efficient field arithmetic for its size: addition, multiplication, inversion and square roots). The field size $8^{91}+5$ has subsequently been proposed to the Internet Research Task Force. Table 6 lists a roll program for $8^{91}+5$.

The six-character decimal-exponential-complexity in the expression

```
8^91+5
```

may somehow underrate its Kolmogorov descriptional complexity. If standard notations, such as decimal and exponential operators, happen to favor a subverted number (leading to weak ECC), then the subverter would present decimal-exponential-complexity to bolster the number.

The roll programming language aims to partially alleviate this concern (which is a legitimate concern within the narrow confines of a subverted

```
8^91+5 subs 8^91+1 in +4
8^91+1 subs 2^273 in +1
2^273 subs 273 in 2^
273 subs 17 in *16+1
17  subs  1 in *16+1
*16+1 roll +16 up 1
+16 subs +8 in +8
+8  subs +4 in +4
+4  subs +2 in +2
2^ roll *2 up  1
1 subs  in +2
*2 roll +2 up  0
+2 subs +1 in +1
0 subs   in +1
```

Table 6: A 68-word roll program for $8^{91} + 5$

number hypothesis). It aims for greater objectivity by breaking down numbers to the bare basics, Godel's computability, Peano's axioms, which reduce arithmetic to counting.

The number $8^{91} + 5$ was used used in designing the roll language. Precursors to the roll programming language were tested for clarity mainly by implementing $8^{91} + 5$, and various ad hoc smaller numbers. Clarity corrections were incorporated into the roll programming language. The roll programming language could easily favor $8^{91} + 5$, merely due to its central role in the design process, and perhaps also due to bias arising from inevitable motivational preferences.

In any event, experience accumulated from repeatedly programming $8^{91} + 5$ (with Godel's computational constructions but with various alternative syntaxes) likely contributed considerably to the shortness of Table 6. (Some of this experience was recently tranferred to a roll programs for 2^{283} and $2^{521} - 1$.)

1.6 Composite $(2^{127} - 1)^2$

An interesting field size is $(2^{127} - 1)^2$, a composite number, the square of a famous prime $2^{127} - 1$. This prime has been proposed for Galbraith–Lin–Scott elliptic curves and variants, both for its especially efficient field arithmetic, and for provided an extra efficiency to an elliptic curve.

```
(2^127-1)^2 subs 2^127-1 in ^2
2^127-1 subs 4 in catalan-mersenne
catalan-mersenne roll 2^-1 up 2
2^-1 roll *2+1 up 0
^2 roll +*2+1 up 0
+*2+1 roll +1 up *2+1
*2+1 roll +2 up 1
4 subs 2 in +2
2 subs 0 in +2
1 subs   in +2
+2 subs +1 in +1
0 subs in +1
```

Table 7: A 58-word program to compute $(2^{127}-1)^2$

The prime $2^{127}-1$ is special in many senses. Mersenne conjectured in 1644, with little justification, that $2^{127}-1$ is prime. (Several of Mersenne's similar guesses were wrong, so his guess for $2^{127}-1$ seems a fluke.) Lucas proved in 1876 that $2^{127}-1$ is prime. It remained the largest known prime until 1950, when it was beat by $2^{521}-1$. So, $2^{127}-1$ seems to be the largest prime ever proved by hand.

Learning from Lucas that $2^{127}-1$ is prime, Catalan defined Catalan–Mersenne numbers and conjectured them to be prime, based on

$$2^{127}-1 = 2^{2^{2^{2^2-1}-1}-1}-1.$$

The next Catalan–Mersenne number has $2^{127}-1$ binary digits, and all known tests for primality are infeasible, with known theory (the prime number theorem plus some heuristics) only predicting a negligible probability ($\approx 2^{-126.47}$) of primality.

Table 7 uses the Catalan–Mersenne sequence to implement $(2^{127}-1)^2$ with a short roll program.

The concern that elliptic curve cryptography might be weaker for composite field sizes is considered milder for $(2^{127}-1)^2$ than for 2^{283}, because the field extension degree 2 is smaller than 283, offering less structure to attack.

1.7 Bitcoin's secp256k1 prime field size

The prime $2^{256} - 2^{32} - 977$ is less than 2^{256}. It is close to a Solinas number yet not Crandall number (which is a number close to a power of two). It is specified as a field size of a prime-order elliptic curve with complex multiplication by $\sqrt[3]{1}$, permitting Gallant–Lambert–Vanstone scalar multiplication. The resulting curve is known as `secp256k1` because of its ASN.1 object identifier in the standard SEC2.

Table 8 lists a roll program to compute $2^{256} - 2^{32} - 977$, but it uses somewhat rote code golf steps compared to the custom methods of the other listed programs.

```
2^256-2^32-977 subs 2^32+977 256  in 2^b-a
2^32+977 subs 2^32 977 in +
2^32  subs 32 in 2^
977 subs 47 10 in 2^b-a
256 subs 8 in 2^
47 subs 17 6 in 2^b-a
2^b-a roll -1 up 2^
32 subs 5 in 2^
17 subs 16 in +1
16 subs 4 in 2^
10 subs 8 in +2
8 subs 6 in +2
6 subs 5 in +1
5 subs 3 in +2
4 subs 3 in +1
3 subs 1 in +2
-1 roll b up 0
2^ roll *2 up 1
1 subs  in +2
*2 roll +2 up 0
+2 subs +1 in +1
+ roll +1 up a
b roll  a up a
a roll +1 up 0
0 subs in +1
```

Table 8: A 127-word program to compute $2^{256} - 2^{32} - 977$

The curve `secp256k1` is used in Bitcoin, and some other cryptocurrencies, for transaction signatures. So, presumably, its discrete logarithm is secure. Some might wish to argue that this curve is a less likely to be subverted than NIST curve P-256. In particular, the curve coefficients are clearly simpler. Reasons to consider the field size less subvertible than are P-256 are unclear. So far, code golf efforts with roll favor the field size of P-256 over that of `secp256k1`.

1.8 Prime $2^{336}-3$

Recently, M. Scott proposed an elliptic curve with the prime field size $2^{336}-3$. The roll program in Table 9 computes this field size.

```
2^336-3 subs 2^335-2 in *2+1
2^335-2 subs 2^334-1 in *2
2^334-1 subs 334     in 2^-1
334 subs 167 in *2
167 subs 83  in *2+1
83  subs 41  in *2+1
41  subs 20  in *2+1
20  subs 10  in *2
10  subs  5  in *2
 5  subs  2  in *2+1
 2  subs  0  in +2
2^-1 roll *2+1 up 0
*2+1 subs *2 in +1
*2 roll +2 up 0
+2 subs +1 in +1
0  subs in +1
```

Table 9: A 79-word program to compute $2^{336}-3$

2 Defining roll

This section defines the roll programming language. Table 10, together with the system of forward (look-ahead) references summarizes the roll programming language pretty well.

Definition	Input	Output
(none)	$()$	0
(none)	$(a, \dots)$	$a + 1$
`f subs g ...  h in k`	$(\dots)$	$k(g(\dots), \dots, h(\dots))$
`f roll g up h`	$()$	0
`f roll g up h`	$(0, \dots)$	$h(\dots)$
`f roll g up h`	$(a + 1, \dots)$	$g(f(a, \dots), a, \dots)$
`f when g`	$(\dots)$	$\min\{a : 0 = g(a, \dots)\}$

Table 10: Functions in a roll program

2.1 Words

A roll program consists of zero or more space-separated **words**. A word is any space-free sequence of characters. The **length** of a roll program is its number of words. (In Linux, `wc -w` can compute the length.)

The five words

```
subs        in        roll        up        when
```

are **verbs**. Verbs define the meanings of remaining words in a program, which are **nouns**.

If a noun is followed by one of the three verbs `subs`, `roll`, or `when`, then the word is a **name** of a definition. Any other noun refers, by **forward reference**, to the next occurence of that noun as a name (if there is any such occurrence).

2.2 Roll programs as functions

Each roll program describes a mathematical function.

Many different roll programs may describe the same mathematical function. One of the tasks of this report is to seek the shortest roll program for a given mathematical function, thereby providing one measurement of its Kolmogorov descritpional complexity.

The focus of this report are constant functions, that always return the size of the field. The roll programs for these constant functions internally use intermediate non-constant functions.

2.3 Numbers

The range of any function described by a roll program is the set of numbers

$$N = \{0, 1, 2, \dots\}.$$

The image of a function is a subset of its range. For constant functions, the image set contains just one number.

2.4 Strings

The set $N^* = \bigcup_{e \in N} N^e$, using Kleene's notation, is the set **strings** of numbers. These are finite strings of the form

$$(a, b, \dots, c).$$

The string's entries are $a, b, \dots, c \in N$. The length of the string is its how many entries it has. For example, the empty string $()$ has length 0, the string $(3, 97)$ has length 2.

If f is the function, and $(a, b, \dots, c)$ is an input to the function, then we generally describe the output of the function is written as $f(a, b, \dots, c)$, which is the usual standard mathematical notation. In particular, if the input is the empty string, then the output is $f()$.

The standard notation $f(a, b, \dots, c)$ is not part of the definition of the roll programming language (but programmers are free to use nouns with this notation embedded in them). The standard notation $f(a, b, \dots, c)$ is used in this export to explain how roll programs work.

Every function defined by a roll program takes inputs which are strings of numbers.

2.5 Domains or partial functions

A roll program describes a function $f : M \to N$, where $M \subseteq N^*$ is the **domain** of the roll program. In other words, a roll program describes a **partial** function $f : N^* \to N$. A function is **total** if $M = N^*$.

The main aim of this report is roll programs that are both total and constant. In particular, all the functions defined within the example programs, including the intermediate function internal to the programs, are total. No partial functions appear in the examples.

2.6 Default function

The **default** function is a function f that returns 0 on the empty string and otherwise the successor of the first entry of the input string. In other words,

$$f() = 0,$$
$$f(a, \dots) = a + 1.$$

A common example is to use `+1` to refer to the default function, and to never use `+1` as the name of a definition. If the noun `+1` never appears as a name of defintion, all appearances as forward references find no name to refer, which implies that the noun refers to the default function above, not a program-defined function.

The domain of the default function is N^*, so the default function is a total function $N^* \to N$.

2.7 Substitution

A function f can be described in a roll program with a definition of the form

```
f subs g ...  h in k
```

Such a definition in a roll program defines function f using substitutions of the form

$$f(a, b, \dots, c) = k(g(a, b, \dots, c), \dots, h(a, b, \dots, c)). \qquad (1)$$

A common example is

```
0 subs in +1
```

where `+1` refers to the default function. The described function is constant, with $0(a, b, \dots, c) = +1() = 0$.

The domain of f depends on the domain of $g, \dots, h, k$. If the latter all have domain N^*, then f has domain N^*. In general, an input is in the domain of f only if it is the domain of $g, \dots, h$, and the resulting application of these functions is in the domain of k.

2.8 Primitive recursion

A function f can be defined in a roll program as

```
f roll g up h
```

which defines function f using primitive recursion:

$$f() = 0 \tag{2}$$

$$f(0, b, \ldots, c) = h(b, \ldots, c), \tag{3}$$

$$f(a + 1, b, \ldots, c) = g(f(a, b, \ldots, c), a, b, \ldots, c). \tag{4}$$

A typical example is

```
a roll +1 up 0
```

which defines a function a such that $a(a, b, \ldots, c) = a$, provided that `+1` refers to the default function, and `0` refers to some definition of the zero constant function.

The domain of f depends on the domains of g and h. If the latter have domain N^*, then so does f. Otherwise, an input is in the domain of f only if, in every intermediate step, the input to each function is in its domain.

2.9 Unbounded recursion

A function f can be defined in a roll program as

```
f when g
```

which defines a function f whose defined outputs essentially take the form

$$f(b, \ldots, c) = \min\{a : g(a, b, \ldots, c) = 0\}. \tag{5}$$

More precisely, let $f(b, \ldots, c) = a$ if

$$g(a, b, \ldots, c) = 0$$
$$g(a', b, \ldots, c) > 0$$

for all $a' < a$. (This condition implicitly requires that $g(a', b, \ldots, c)$ is defined for all $a' \leq a$.)

Otherwise, $f(b, \ldots, c)$ is not defined. Roll programs that describe partial functions $N^* \to N$, whose domain is not the full set N^* of strings, must contain the word `when`. In other words, if a roll program lacks the word `when`, then it defines a total function.

3 Running roll programs by machine

It can helpful to have a machine run a roll program, even though a nicely written roll program should be verifiable with basic mathematical skills.

3.1 Unoptimized

The following C++ code is unoptimized, except for inclusion of optimized code described in the next section of this report. The unoptimized code takes a time at least a constant times the difference of the output and largest input.

By changing the macro `USING_NTL` to `0`, the code becomes valid C99 code, but then the maximum numbers that can be processed is much smaller.

```
// roll.c++

#define USING_NTL 1

// Parsers:
typedef char*P;
P end(P p){return *p? 0: p;}
P blank(P p){return ' '==*p|| '\n'==*p|| '\t'==*p? p+1: 0;}
P letter(P p){return  '!' <= *p&&*p <= '~'? p+1: 0;}
P space(P p){P t; while(t=blank(p))p=t; return p;}
P word(P p){P t;
  if (!(p=letter(p))) return 0;
  while(t=letter(p)) p=t;
  return space(p);}
P hear(P p,P q){P t;
  return (t=letter(p))?
    *p==*q?  hear(t,letter(q)): 0:
    letter(q)? 0: space(p);}
#define hear(p,q) hear(p,(P)q)  // needed for C++
P subs (P p){return hear(p, "subs ");}
P in   (P p){return hear(p, (P)"in   ");}
P roll (P p){return hear(p, (P)"roll ");}
P up   (P p){return hear(p, (P)"up   ");}
P when (P p){return hear(p, (P)"when ");}
P verb (P p){P t; return (t=subs(p)) || (t=roll(p)) || (t=when(p)), t;}
P term(P p){return verb(p)? 0: word(p);}
P noun(P p){return   in(p)? 0: term(p);}
P sentence (P p){P t;
  return (t=in(p)) || (t=noun(p)) && (t=sentence(t)), t;}
P plan (P p){P t;
  return
    (t=subs(p)) && (t=sentence(t)) && (t=noun(t)) ||
    (t=roll(p)) && (t=noun(t))     && (t=up(t)) && (t=noun(t)) ||
    (t=when(p)) && (t=noun(t)) , t;}
P strategy (P p) {P t;
  return
    (t=end(p)) ||
    (t=noun(p)) && (t=plan(t)) && (t=strategy(t)), t;}
P program (P p){P t; return (t=space(p)) && (t=strategy(t)), t ;}

// Analyzers
int num_subs(P p){int n=0;
  while(!in(p)) n+=1,p=word(p);
  return n;}

// Mover
P call (P p){P t=p;
```

```
  while(t && !strategy(t)) t=word(t);
  while(t && !end(t))
    if(hear(t,p)) return t;
    else (t=noun(t)) && (t=plan(t));
  return t;}

#if USING_NTL
#include <NTL/ZZ.h>
using namespace std;
using namespace NTL;
typedef ZZ I;
#else
typedef long long I;
#endif

// internal input managers
void let(I*j,I*i){
  while(*i>-1)*j++=*i++;
  *j=-1;}
int len(I*i){int len=0;
  while(*i++>-1)len++;
  return len;}

// general program runners
I run_strategy(P p,I*i);

// #include "opt_subs.c++"
I run_subs(P p,I*i){
  int k,n = num_subs(p);
  I j[n+1];
  for(k=0;k<n;k++){
    j[k]=run_strategy(call(p),i);
    p=word(p);}
  p=in(p);
  j[n]=-1;
  return run_strategy(call(p),j);}

#include "opt_roll.c++"
I run_roll(P p,I*i){
  if(*i<=0) return 0!=*i?(I)0: run_strategy(call(up(noun(p))),i+1);
  else {I o=run_roll_opt(p,i);//=(I)-1;
    if (o>=0) return o;
    else {
      I j[len(i)+2];let(j+1,i);j[1]=0;
      j[0]=run_roll(p,j+1);
      for(;j[1]<i[0];j[1]+=1)
        j[0]=run_strategy(call(p),j);
      return j[0];}}}

I run_when(P p,I*i){
  I j[len(i)+1];let(j+1,i);
  for(p=call(p),j[0]=0; 0!=run_strategy(p,j); j[0]+=1);
  return j[0];}

I run_plan(P p,I*i){P t; return
    (t=subs(p))? run_subs(t,i):
    (t=roll(p))? run_roll(t,i):
```

```
    (t=when(p))? run_when(t,i): (I)-1;}
I run_strategy(P p,I*i){return end(p)?1+*i: run_plan(noun(p),i);}
I run_program (P p,I*i){return program(p)?run_strategy(space(p),i):(I)-7;}

// input from character strings
void decimal(I*i,int c,char**s){int b;
  for(b=0;b<c;b++,i++)
#if USING_NTL
    conv(*i,s[b]);
#else
  {char *t; for(*i=0,t=s[b];*t;t++)*i*=10,*i+=*t-'0';}
#endif
  *i=-1;}

#include <stdio.h>
#define MAX_FILE (1000*1000)

I run_file(P a,I*i){
  char p[MAX_FILE+1]={};
  if (fopen(a,"r")) {
    p[fread(p,1,MAX_FILE,fopen(a,"r"))]=0;
    run_program(p,i);}}
I run_arg1(P p,I*i){return (noun(p) && end(noun(p)))?
    run_file(p,i): run_program(p,i);}


void print(I a){
#if USING_NTL
    cout << a << "\n";
#else
    printf("%lld\n",a);
#endif
}

int main (int c, char**a) {
  if(2<=c){
    I i[c-1];
    decimal(i,c-2,a+2);
    print(run_arg1(a[1],i));}}
```

3.2 Optimizations

The naive roll interpreter code of the previous section only ever modifies integers by adding (or subtracting) one. So, running roll programs for a secure ECC field size n with the naive interpreter would take at least n steps, which would be too long.

For simple roll programs, a few optimization suffice to make the interpreter run fast enough. For example, the code

```
f roll +2 up g
```

can implemented with the shortcut,

$$f(a, b, \ldots, c) = 2a + g(b, \ldots, c).$$

For large a and efficient addition arithmetic, this is exponentially faster than incrementing $2a$ times.

```
// opt_roll.c++

// parsers for optimizable roll steps
P opt_plus_1(P),opt_a(P),opt_b(P),opt_plus_2(P);

// optimized code
I run_roll_opt(P p,I*i){
  return
    opt_plus_1 (p)? run_strategy(call(up(noun(p))),i+1) + i[0]:
    opt_a      (p)? run_strategy(call(up(noun(p))),i+1):
    opt_b      (p)? i[0]-1:
    opt_plus_2 (p)? run_strategy(call(up(noun(p))),i+1) + 2*i[0]:
    (I)-1107;} // not optimized

// "+1" -> ""
P opt_plus_1(P p){
  return hear(p,(P)"+1 ") &&
    (end(call(p))) ?
    noun(p): 0;}

// "+2" -> "+2 subs +1 in +1" -> ...
P opt_plus_2(P p){P t;
  return hear(p,(P)"+2 ") &&
    (t=opt_plus_1(subs(noun(call(p))))) &&
    opt_plus_1(in(t)) ?
    noun(p) : 0;}

// "0" -> "0 subs in +1" -> ...
P opt_0(P p){
  return hear(p,(P)"0 ") &&
    opt_plus_1(in(subs(noun(call(p))))) ?
    noun(p): 0;}

// "a" -> "a roll +1 up 0" --> ...
P opt_a(P p){P t;
  return hear(p,(P)"a ") &&
    (t=opt_plus_1(roll(noun(call(p))))) &&
    opt_0(up(t)) ?
    noun(p): 0;}

// "b" --> "b roll a up a" --> ...
P opt_b(P p){P t;
  return hear(p,(P)"b ") &&
    (t=opt_a(roll(noun(call(p))))) &&
    opt_a(up(t)) ?
    noun(p): 0;}
```

4 To do

Many elaborations of this work may be doable.

- Shorter versions of the listed programs.
- Roll programs for other cryptographic constants, such as
  - numbers far from a power of two (being inefficient field sizes for ECC), like
    - the 314-bit prime $99^9 + 4$ with a 63-word roll program,
    - a 43-word roll program implementing a composite $\approx 2^{2031.4}$,
  - numbers derived from irrationals like $\sqrt{2}$, π and e,
  - numbers derived from cryptographic hash functions.
- Bit complexity of roll programs, a normalization of word length.
- A more thoroughly optimized interpreter or compiler.
- Typical lengths (par scores) for numbers of a given bit length.
- Steamrolling: searching through all roll programs to find the shortest implementing a constant with a given property.
- Ways to find, to verify, or to estimate, the shortest program for a given function, especially a constant.
- Models that imply information must be embedded into subverted numbers.
- Basic language analysis for roll, such as
  - illustrative tutorial and guide,
  - advance programming tips and tools,
  - limitation such as finite arity (roll can only describe functions depending on the first $R + 1$ entries of the input string, where R is the number of words `roll` in the program),
  - subtleties and common pitfalls (e.g., forgetting no number is pre-defined),
  - design motivation and justification,
  - length bounds for restricted language subsets like
    - primitive recursive programs (no `when`),
    - non-recursive programs (no `roll` or `when`),
    - trickle programs (each noun appears at most twice),
  - Kolmogorov's algorithmically random numbers (whose shortest program does not use `roll`?),
  - systematic comparison to other measures of descriptional complexity like
    - decimal exponential complexity (e.g., `8^91+5` is six characters and standard notation as understood by `bc`, etc.)

    - straight line programs (with addition and multiplication, but no loops),
    - code size in terse languages, J or code golfing languages,
    - length in various Turing tar-pits (e.g., automata),
  - Turing completeness and relative efficiency,
  - Kleene normal form, halting problem, and undecidability of length.

References

Wikipedia was the reference for Godel's definition of computability.

People and organizations who proposed the cryptographic constants studied in this report are each named in the appropriate section.