Hiding Data in Plain Sight: Towards Provably Unbreakable Encryption with Short Secret Keys and One-Way Functions

Mircea-Adrian Digulescu[1,2]

December 2019

[1]Individual Researcher, Russian Federation
[2]Formerly: Department of Computer Science, Faculty of Mathematics and Computer Science, University of Bucharest, Bucharest, Romania, Romania
mircea.digulescu@mail.ru, mircea.digulescu@my.fmi.unibuc.ro

Abstract

It has long been known that cryptographic schemes offering provably unbreakable security exist - namely the One Time Pad (OTP). The OTP, however, comes at the cost of a very long secret key - as long as the plain-text itself. In this paper we propose an encryption scheme which we (boldly) claim offers the same level of security as the OTP, while allowing for much shorter keys, of size polylogarithmic in the computing power available to the adversary. The Scheme requires a large sequence of *truly random* words, of length polynomial in the both plain-text size and the logarithm of the computing power the adversary has. We claim that it ensures such an attacker cannot discern the cipher output from random data, except with small probability. We also show how it can be adapted to allow for several plain-texts to be encrypted in the same cipher output, with almost independent keys. Also, we describe how it can be used in lieu of a One Way Function.

Keywords: Encryption, Provable Security, Chaos Machine, Truly Random, One Time Pad, One Way Function.

1 Introduction

Most existing encryption schemes work by identifying some suitable family of bijective functions, one for each potential secret key, from the universe $\{0,1\}^n$ of plain-texts to the universe $\{0,1\}^m$ of cipher outputs. In this paper, we, the author, take a different approach by supplementing the input with a large, *truly random* sequence. This sequence, with only a few minor, hopefully undetectable, changes is then outputted as the result of encryption. The core idea is the following: we will partially permute the words in the truly random sequence based on the plain-text and the short secret key. The main ingredient to security lies in using the elements of the random sequence itself to determine which positions to exchange. The assumption on which our security claims are based is that the partial permutation determined by our method will appear random to any adversary having less computing power than allowed. More precisely, no algorithm of running time less than exponential in some arbitrary chosen security parameter should be able to distinguish the cipher output from random, even when the plain text is known or can be chosen.

From the ancient polyalphabetic substitution cyphers, the idea of employing random decision making in the encryption method is not new. The security of a scheme depends, however, on how randomness is employed precisely. The scheme we propose paves the way to a plethora of similarly-built ciphers, all relying on the same core ideas we present in this paper. As such, it can be regarded as being the first specimen from a newly introduced class of encryption methods.

The proposed scheme is intended to be cryptographically secure. All of the author's knowledge of fields such as computability (e.g. Kolmogorov complexity), complexity (e.g. NP-Completeness), [CS]PRNGS ([Cryptographically Secure] Pseudorandom Number Generators) as well as knowledge of existent schemes such as AES [2] and RSA [3], together with the assumptions on which their security claims are based, played a role in developing the current proposal. The claim that the proposed scheme is cryptographically secure is thus intended to be taken as an educated statement, not a mere shot in the dark. Furthermore, the scheme can be adapted for use in lieu of a One Way Function (which is something actively sought by researchers, consisting of an easy to calculate function, whose inverse is computatinally hard to determine - see [1]), for most scenarios.

1.1 Prior work

To the best of our knowledge there is no public body of literature pertaining to encryption schemes which involve making small changes to a large volume of random data, based on the secret key and the plain text as well as on said random data. Nevertheless, we, the author, strongly suspect that non-public research, by people such as Marius Zimand (see [4] and [5]), Leonid Levin (see [1]) and others who have a competent scientific interest in randomness, Kolmogorov complexity and the like, exists which includes ideas similar to those in this paper. Nevertheless, to the best of our knowledge, such research, if it exists, is not public. Chaos Theory has played a role in the development of some ideas in this paper. Its applications in encryption, in the form of Chaos Machines, are best described by Armour in [6]. Such machines can easily be employed to augment the security of the proposed encryption scheme.

1.2 Overview of this paper

The rest of the paper is organized as follows. In Section 2 we present the proposed Encryption Scheme, including an abridged version of its pseudocode. We include a brief natural language description of its steps and also very briefly discuss the theory behind it, before making bold claims regarding its cryptographic security. We provide no formal proofs, but, instead, claim that such exist. We conclude by analyzing the performance of the proposed algorithm. In Section 3 we present some important ideas on further increasing security, while in Section 4, we tackle the opposite tradeoff, by presenting a practically feasible, simplified version of the Scheme. In Section 5, we present an idea on how such ciphers can be modified to allow multiple, independent plain-texts to be encrypted within the same cipher output, using almost independent keys. In Section 6, we show how the scheme can be used in lieu of a one way function. Sections 7 and 8 are dedicated to conclusions and an abridged version of The Acknowledgments, respectively.

Appendix A contains the full pseudocode of the proposed Scheme, while Appendix B includes a detailed discussion of the theory and rationales behind decisions taken in its design. Proofs of the theorems in this paper are found in Appendix C. Appendix D includes a more detailed discussion related to the practical feasibility of the simplified version, while Appendix E discusses in further detail the way in which such schemes can be used in lieu of a one way functions. In Appendix F, we briefly tackle a few other practical considerations. The Appendixes are included as an additional separate Annex to this paper, which is not part of it.

2 The Encryption Scheme

In this section we proceed to describe the proposed encryption scheme. For lack of a better name, we shall call it Short Key Random Encryption Machine or SKREM in short.

2.1 The Encryption Scheme SKREM

The scheme takes the following inputs for encryption:

1. Three sequences $M1$,$M2$ and $M3$, of sufficiently large size, consisting of m independent, uniformly distributed, truly random w-bit words each. We call these **grand master tables**. When it is obvious to which we refer, we use the notation M. These are unique to a particular encryption and are never reused.
2. The **plain text** P consisting of n bits which need to be hidden. It can be arbitrary.
3. A sequence T of sufficiently large size, consisting of independent, uniformly distributed, truly random w-bit words, to be used for random decision making. This sequence is discarded after use (and never reused). We call it the **randomness well**.
4. A relatively short list of secret key elements, $K\_small[]$, consisting of uniformly distributed, independent, truly random bits. This is the **secret key** which needs to be provided at decryption.
5. Two equally sized lists of secret key elements $K1\_large[]$ and $K2\_large[]$, consisting of uniformly distributed, independent, truly random bits. We call these, the **large secret keys**. These are to be discarded after use (and never reused).

SKREM also incorporates a number of security parameters, grouped in the pseudocode in the $parameters\_*$ structures which are an integral part of the scheme. Their values can be adjusted to obtain other SKREM-like schemes, however they are required to be identical at both encryption and decryption. The secret key and large secret keys can just as well be generated using the randomness well and provided as output, when no specific secret key generation method is required.

The first half of each grand master table is conceptually split into a number of smaller master tables, of equal size, to allow locations from it to be sampled using a lesser number of bits. The second half is used for replenishing values consumed from the first half. Each secret key element incorporates a few small numbers (called **key atoms**) which are used to generate a single location within a small master table.

The cipher output produced at encryption by the scheme, consists of the three, slightly modified, grand master tables $M1$,$M2$ and $M3$. The Decrypt routine, as expected, takes as input the output from the Encrypt and produces the original plain text.

Consider the following abridged version of the pseudocode for the SKREM encryption scheme. The full, detailed pseudocode is available in Appendix A.

Algorithm 1. *Abridged version of the pseudocode of the Encryption Scheme SKREM.*

```
1: STRUCT params_normal
2: reqsec ← 256{Security strength parameter: Logarithm of computing power available to an adversary}
3: dmod ← 0{Used to specify direct mode of key extension, using only one round}
4: vrfy ← 1{Specifies whether to return an error in case the security parameters do not offer the specified security strength}
5: mtsize ← 85{Used to determine the size of one small master table}
```

6: $secrbase \leftarrow 4$\{Number of additional secret bits hidden in each key element to be used as base during key extension\}
7: $secrtwo \leftarrow 1$\{Used to specify that an additional random exponent is to be sampled from the grand master table, during key extension\}
8: $secrbpp \leftarrow 256$\{Length in bits of the offset used to select a random prime. Must be $\leq reqsec$.\}
9: $secrbpb \leftarrow 9$\{Number of bits, whose XOR is used to represent 1 emitted bit, during key extension\}
10: $secrbpn \leftarrow 9$\{Number of bits, whose XOR is used to represent 1 bit of plain text\}
11: $ppx \leftarrow 9$\{Number of key atoms per secret key element\}
12: $w \leftarrow 8$\{Number of bits in a word in M and T\}
13: $bopf \leftarrow 2$\{Multiplication factor for the number of bits in a number l, necessary to sample it with (almost) uniform distribution from uniformly distributed individual bits\}
14: **END**
15: **METHOD Encrypt**$(P, K1\_large[], K2\_large[], K\_small[], M1, M2, M3, T)$
16: Use BasicEncryptDecrypt() to encrypt the two large keys, $K1\_large[]$ and $K2\_large[]$ into $M3$, with secret key $K\_small[]$.
17: Use the randomness well T to generate an OTP for the plain text P.
18: Use BasicEncryptDecrypt() to encrypt the OTP into $M1$, with secret key $K1\_large[]$, and its XOR with the plain text in $M2$, using secret key $K2\_large[]$.
19: **return** The modified grand master tables $M1$,$M2$ and $M3$ as the cipher output.
20: **END**
21: **METHOD Decrypt**$(M1, M2, M3, n, szLKey, K_small)$
22: Use BasicEncryptDecrypt() to retrieve the two large keys, $K1\_large[]$ and $K2\_large[]$ from $M3$, using key $K\_small[]$.
23: Use BasicEncryptDecrypt() to retrieve the OTP from $M1$, using secret key $K1\_large[]$ and its XOR with the plain text from $M2$, using secret key $K2\_large[]$.
24: **return** The XOR of the two bit sequences obtained above, as the original plain text.
25: **END**
26: **METHOD BasicEncryptDecrypt**$(mode, n, P, K[], M, T, t, params)$
27: \{Performs encryption of P into M based on K[] or decryption from M into P based on K[], depending on mode\}
28: \{Initializations\}
29: Initialize some constants based on the input and the security parameters. Of particular interest are the following. The values for when $dmod = 1$ differ slightly, but their semantics remains the same.
30: $k \leftarrow (secrbase * (1 + secrtwo) + (ppx) * mtsize * bopf + secrbpp) * bopf + secrbpp$ \{Size of a key element in bits\}
31: $f \leftarrow 1 + ([1/((1 - 1/2^w))] - 1) * 10$ \{Number of pairs of words sufficient to generate one random bit\}
32: $reqBitsPerKey \leftarrow k * 8 * (1 + secrtwo) * secrbpb$ \{Number of bits required by a single key element, for extension\}
33: $reqWords \leftarrow (reqBitsPerKey * (secrbpn/ppx) * n/7 + secrbpn * n) * 2 * f$ \{Total number of words consumed by the algorithm\}
34: $keyExtFactor \leftarrow 8$ \{Number of new key elements to which a single old key element is extended\}
35: $mtSize \leftarrow Min(maxMTsize, NextPrime(2^{mtsize}))$ \{Size of a small master table. Chosen to be a prime number, around 2^{mtsize}\}
36: $noMTs \leftarrow Min(\lfloor \frac{m/2}{mtSize} \rfloor, 2 * f * reqBitsPerKey)$ \{Total number of small master

tables}
for $i = 0$ **to** $noMTs * mtSize - 1$ **do**
  Perm.Add(i) {Initialize Perm to be the identity permutation}
end for
{Security Parameters Validations}
if $vrfy = 1$ **then**
  Ensure that the following constrains are respected.
  $ppx * K[].Count * mtsize \geq reqsec + mtsize$
  $ppx * K[].Count * 4 \geq reqBitsPerKey/ppx$
  $mtsize + (mtsize - 1) * 2 + 3 \geq reqsec$
  $maxMTsize \geq NextPrime(2^{mtsize})$
  $noMTs \geq 2 * f * reqBitsPerKey$
end if
{Key Extension}
while There are $\leq secrbpn * n$ key elements **do**
  for All old key elements and for increasing indexes of atoms within a key element and of small master tables **do**
    Use $ExtractJthLocation()$ and $GetLocation()$ to obtain two locations $lp1$ and $lp2$ within the permutation list $Perm$, using a single atom of a single old key element.
    Use $Perm$ to obtain two locations $lm1$ and $lm2$ in the grand master table M form $lp1$ and $lp2$.
    Use $BurnLocation()$ to mark the locations $lm1$ and $lm2$ as used, and replenish $Perm$ acccordingly.
    Use $GetBit()$ to obtain a random bit $b2$ from the values $M[lm1]$ and $M[lm2]$.
    Distribute $b2$ to the appropriate old key element, to be used for its extension.
  end for
  if A sufficient number of bits has been emitted to allow for extension of all old key elements **then**
    XOR every *secrbpb* bits from those distributed to each key element for extension, to obtain a usable bit.
    Extend each old key element into $keyExtFactor$ new key elements, using $ExtendKey()$, thus concluding one key extension round.
  end if
end while
{Getting Locations For Encryption}
Use the last round key elements to emit a number of $secrbpn * n$ bits from about twice as many locations in M, keeping a record of both.
{Encrypt / Decrypt}
Use the XOR of every *secrbpn* from the emitted bits above to represent a single bit of plain-text.
At encryption time, if this bit does not correspond to the desired one from P, swap the values at a single, randomly chosen, pair of locations in M, from the *secrbpn* used to generate it. Use the randomness well T to pick the exact pair.
{Return result}
return (P, M, t)
END
METHOD ExtendKey$(K, ExtendBits, t, NewVals, k, count, params)$
{Extends the key element K, by adding *count* new key elements to the $NewVals$ list}
$(\_, KeyBits) \leftarrow ExpandKey(K, k, params)$
$lp \leftarrow 2^{k - secrbpp * (bopf + 1)}$

for $i = 0$ **to** $count - 1$ **do**
  $x \leftarrow 17$, $y \leftarrow 14$, $v1 \leftarrow 17$, $v2 \leftarrow 28$
  $q \leftarrow 0$
  $(po, t) \leftarrow BuildValue(ExtendBits, t, secrbpp)$
  $p \leftarrow NextPrime(lp + po)$
  $(v1, q) \leftarrow BuildValue(KeyBits, q, secrbase)$
  if secrtwo > 0 **then**
    $(v2, q) \leftarrow BuildValue(KeyBits, q, secrbase)$
  end if
  $(x, t) \leftarrow GenerateRandomFromBits(ExtendBits, t, p, params)$
  if secrtwo > 0 **then**
    $(y, t) \leftarrow GenerateRandomFromBits(ExtendBits, t, p, params)$
  end if
  $newK \leftarrow (20 + [(v1 + 14)^{28+x} + (17 + v2)^{17+y} + 11431]^{-1}) \mod p$
  $newBits \leftarrow \emptyset$
  $GetBits(po, secrbpp, newBits)$
  $GetBits(newK, k - secrbpp, newBits)$
  $(genK, \_) \leftarrow BuildValue(newBits, 0, k)$
  $NewValus.Add(genK)$
end for
return t
END
METHOD ExtractJthLocation$(K, k, j, params)$
{Gets the value associated with the j-th atom used to index the small master tables, from key element K}
static $lastresult \leftarrow 14$ {Retains value between method calls}
$(\_, KeyBits) \leftarrow ExpandKey(K, k, params)$
$p \leftarrow NextPrime(2^{mtsize})$ {Gets the smallest prime larger than this value}
$q \leftarrow secrbase * (1 + secr\_two) + j * mtsize * bopf$
$(l, \_) \leftarrow GenerateRandomFromBits(KeyBits, q, p, params)$
if $lastresult < 2$ **then**
  $lastresult \leftarrow 28$
end if
$l \leftarrow [17 + (lastresult^{l+28} + 14)^{-1}] \mod p$
$lastresult \leftarrow [17 + (lastresult^{l+20} + 11431)^{-1}] \mod p$
return l
END
METHOD BurnLocation$(Perm, l, m, noMTs, mtSize)$
{Marks the location $Perm[l]$ from the grand master table as used and performs some minor shuffling of $Perm$, based on l}
$q \leftarrow \lfloor l/mtSize \rfloor$
$j1 \leftarrow \lfloor \frac{q}{2} \rfloor$
$j2 \leftarrow q + \lfloor \frac{noMTs-1-q}{2} \rfloor$
$l1 \leftarrow j1 * mtSize + \lfloor \frac{l}{2} \rfloor$
$l2 \leftarrow j2 * mtSize + l + \lfloor \frac{mtSize-l}{2} \rfloor$
$Perm[l1] \leftrightarrow Perm[l2]$
$Perm[l] \leftarrow m - 1$
$m \leftarrow m - 1$
return m
END

123: **METHOD GetLocation**$(l, x, noMTs, orgNoMTs, mtSize, params)$
124: {Gets a location in $Perm$ based on location index l in a small master table of size $mtSize$ and a value x, with $0 \leq x < orgNoMTs$}
125: **END**
126: **METHOD ExpandKey**$(K, k, params)$
127: {Expands key element K into its constituent parts so they can be used directly. These parts are the prime modulus used and the bits which represent all the atoms}
128: **END**
129: **METHOD GetBit**$(w1, w2)$
130: {Returns a uniformly distributed bit based on two random, but potentially not uniformly distributed, words $w1$ and $w2$}
131: **if** $w1 = w2$ **then**
132: **return** *null*
133: **end if**
134: **if** $w1 < w2$ **then**
135: **return** 0
136: **end if**
137: **return** 1
138: **END**
139: **METHOD GenerateRandomFromBits**$(Bits, t, l, params)$
140: {Returns an almost uniformly distributed value between 0 and $l-1$ based on some uniformly distributed random bits found in the sequence $Bits$, starting at index t}
141: $a \leftarrow GetReqGenerateBits(l, params)$
142: $x \leftarrow 0$
143: **for** $i = 0$ **to** $a - 1$ **do**
144: $x \leftarrow x + Bits[t] * 2^i$
145: $t \leftarrow t + 1$
146: **end for**
147: $step \leftarrow 2^a / l$ {Noninteger value with double precision}
148: $q \leftarrow \lfloor x/step \rfloor$
149: **if** $r \neq q$ **and** $q < l$ **and** $(q+1) * step - x \geq x - q * step$ **then**
150: $q \leftarrow q + 1$
151: **end if**
152: **return** (q, t)
153: **END**
154: **METHOD GenerateRandomBitsFromP**(a, p, l)
155: {Returns the bits of an almost uniformly distributed value between 0 and $2^a - 1$ based on some uniformly distributed value, between 0 and $l-1$}
156: The method proceeds analogously to GenerateRandomFromBits.
157: **END**
158: **METHOD GetReqGenerateBits**$(l, params)$
159: {Returns the number of uniformly distributed random bits required to generate an almost uniformly distributed value between 0 and $l-1$}
160: **return** $bopf * \lceil log(l) \rceil$
161: **END**
162: **METHOD GetBits**$(val, noBits, Bits)$ {Adds $noBits$ bits from val to a bit list}
163: **METHOD BuildValue**$(Bits, t, noBits)$ {Builds a value from a bit list}
164: **METHOD NextPrime**(val) {Returns the smallest prime number $\geq val$}

Discussion: The values 14,28,20,11431, and 17 were chosen to be arbitrary beautiful constants ≥ 2, which are used in such a way so as to have no impact on the security of the

scheme. They can be replaced with anything else the reader finds more to his tastes, if desired.

2.2 Discussion and Claims

A detailed version of this discussion can be found in Appendix B.

SKREM proceeds as follows. It encrypts the two large keys $K1\_large[]$ and $K2\_large[]$ using the secret key $K\_small[]$. This step is performed in order to allow for shorter keys in practice. SKREM then proceeds to encrypt the plain text, split in two via an OTP, into grand master tables $M1$ and $M2$. Each of these is, in effect, when taken alone, truly random.

Actual encryption is performed in $BasicEncryptDecrypt()$. The security strength parameter - commonly identified with the key length in bits in most other ciphers - is $reqsec$. This must be taken such that 2^{reqsec} steps is beyond what is tractable by any adversary, within the intended lifetime of the cipher text. The security constrains put in place, and enforced via validation, are rather stringent. They are meant to be generously sufficient to allow us claim that SKREM can be proven to offer unbreakable security.

Encryption proceeds as follows. Firstly, a key extension stage takes place, where the number of available k-bit key elements is extended to $secrbpn*n$. Finally, each atom of the resulting last key elements is used to determine an unused location within the grandmaster table. The values at these locations are than altered to represent the plain text, encoded in the pair-wise relative order of consecutive such. A number of $secrbpn$ bits, represented by $secrbpn$ pairs of locations are XORed together to encode a single bit of P. When the existent bit does not correspond to the desired one, any of these pairs, randomly chosen, will have its values switched. Knowledge of the plain text could potentially be speculated only starting at the last stage, where actual encryption takes place. All transformations performed until then by SKREM are fully independent from it: the list of potential pairs of swappable locations from M is the same for any plain-text. As such, adaptive plain text attacks, as well as chosen cipher text attacks, should offer no noteworthy added benefit whatsoever, over simple known plain text attacks.

During the key extension round, the following occurs. Each old key element is processed by $ExtractJthLocation$ (which uses some modular algebra to spice the result up a bit) to obtain a number of ppx locations between 0 and the size of a small master table. Each such location is used to reference $2 * f * reqBitsPerKey$ locations from the grand master table M, using $GetLocation$ and the indirection vector $Perm$. Each successive pair of locations in M encode 1 (truly random) bit, with probability $(1 - 1/2^w)$, failing $1/2^w$ of the time, when the sampled values are equal. In order to ensure with reasonable probability that we obtain the required number of $reqBitsPerKey$ from each key atom, the number of pairs of words is increased by a factor of f, which is taken to be close to the inverse of the former. The exact probability of failure for the entire process was not computed, but is instead left for further research.

Once $reqBitsPerKey$ bits are generated for each of the new key elements (by all of the old key elements together), they are used to expand each such original key, into $keyExtFactor$ (8) new ones. The bits used for extension, while generated by locations determined deterministically from original key K_small and the grand master table M, are understood to appear truly random and independent to the adversary: since M is truly random, unless we got astronomically unlucky for it to be something predictable (e.g. all 0s), the distribution of values across almost all of its permutations should be just as random. Do note, however, that the entropy (in terms of Shannon Entropy [7]) of the ensuing permutation of M will also never exceed the entropy of the secret key (which is

large enough to exclude brute force guessing, fortunately).

The manner in which a single k-sized key element K is extended is as follows. At least 1 (potentially 2) k-sized, random values x, y, are generated based on the grand master table M, and on part of the key atoms from the other key elements, excluding itself. We expect each of these 8-16 values to be, effectively, indistinguishable from random by the adversary. The exact extension formula consists of the exponentiation of some primitive of $GF(p)$ to a random exponent based on the x,y values, thus producing a random result, regardless of the base used. The summing of two such exponentiations, as well as of the beautiful offset 11431 are meant to prevent a sequence of operations from simplifying to just one. Finally, the modular inverse operation was employed, given its apparent even-today-valid strength with regard to cryptographic usages. It is known that the AES [2] scheme relies fundamentally on it. Given the OSINT available to the author, it seems the security of modular inversion holds in practice. We chose to work in $GF(p)$ (the Galois Field of order p) with varying, (almost) truly random primes p-s, of suitable length, rather than $GF(2^c)$ for some fixed c for three reasons, detailed in Appendix B.

The method used for bit emission, by some key atom A, is designed to deny the attacker the possibility to guess more than a single one for any new key atoms, resulting from extension, by making some guesses about as many old key atoms as his computing power allows. We claim that no succinct characterization or useful property can be determined (except with astronomically low probability) by the adversary, with regard to the relationship between the set of new atoms and the set of old ones. This claim rests on the fact any such relationship will need to depend heavily on the existence of structure and order within M itself - which is, by the definition randomness involving constructive martingales, excluded (except with astronomically low probability).

Guessing all the $mtsize$ bits of the location represented by an atom, could, under some pessimistic scenarios, potentially be used to determine one bit for all new atoms in an extension round. An attacker would then be able to take $\approx 2^{2*(mtsize-1)}$ random guesses for all the remaining bits (of which there are at least $mtsize-1$ per atom - and this value could be increased as per the ideas in Section 3), in order to know with 1σ confidence ($\approx$ 68.5%) if he had guessed right (see [8] and [9]). Each time such an experiment is performed, the set of plausible initial guesses (2^{mtsize}) is reduced to the corresponding fraction (e.g. 0.3% for 3σ). We consider that the security check requring $mtsize$ to be no lesser than $\approx reqsec/3$ is sufficient from a theoretical perspective to allow for the desired level of voracity in our claims that SKREM offers security that can be formally proven to be unbreakable, as per the detailed discussion in Appendix B. Note that any speculated (guessed) property of any round's set of atoms will be impossible to verify before the very last stage, where actual plain text encryption occurs.

At the encryption stage, the attacker is faced with a set of $O(n)$ key elements, which, by the prior arguments, he should not be able to characterize in any useful manner. Each group of $secrbpn$ key elements are used to encode a single bit of plain text. For $secrbrb >= ppx$ (which is the case in SKREM), an intractable number of locations would need to be guessed, for the attacker to be able to constrain the last-round key elements universe to account for even just 1 bit of plain text. The attacker should, thus, be unable to deduce any useful properties of this set, allowing him to characterize the original key $K[]$ or discover some yet-unknown bit of plain text. In effect, the last round keys function like a practical OTP for the plain text (not an actual OTP since its entropy is limited to about $reqsec$). By introducing uncertainty about which pair of locations had its values changed in M, we feel we have made most attacks seeking a useful representation of the set of last round key elements, reduce to the counting problem #SKREM instead of merely the decision one. The counting problem is generally considered harder than the decision

problem, as the difference in tractability between 2CNF-SAT and #2CNF-SAT illustrates [10].

Other, less crucial, decisions add some additional uncertainty over the entire execution of the algorithm. For example, the simple shuffling of just 2 locations used in $BurnLocation()$, we expect to cause the indirection arrangement $Perm$ to stray, before the start of the encryption stage, pretty significantly from anything which is very simple to describe. The usage of non-constant base in $ExtractJthLocation()$, based on effectively all the atoms encountered during the entire course of execution of the algorithm also complicates matters notably.

The method used by $GenerateRandomFromBits()$ and $GenerateRandomBitsFromP()$ to convert from one uniform probability distribution to another merits some attention. It involves dividing the source universe into even slots, each having an associated value from the target universe. This conceptual division entails non-integer, rational thresholds. Since each threshold is a rational number, it may fall between integers. Occasionally a value at a border between two slots is encountered. Since determining the true correspondent would require more precision than available, we simply employ some smart rounding. As such, each value in the target universe may have its distribution altered (increased or decreased) by inclusion / exclusion of a small part of it, situated around the two thresholds of the interval which represents it. Each such amounts to less than 0.5 (thanks to rounding), bringing the total to at most 1 value from the $\frac{2^{bopf*log(l)}}{l} = 2^{(bopf-1)*log(l)}$ representing it, thus making the suggested $bopf = 2$ overly generous. This method, along with the one in $GetBit()$, were discovered by the author in the context of this paper. Nevertheless, we strongly suspect the likes of Marius Zimand or Leonid Levin are also aware of them.

Decryption is identical to encryption, save the lack of need to switch values in the grand master tables. It warrants no separate discussion. Armed with the insights discussed above, we make the following claims.

Claim 1. *Any classical computer algorithm, using less running time than* 2^{reqsec}*, has a probability of less than a value below that corresponding to* 1σ *(*$\approx 68.5\%$*) of determining whether SKREM was used to encrypt some known, arbitrarily chosen plain text* P*.*

Discussion: Essentially, we claim that SKREM offers security *reqsec* against a classical computer. Note that this is a bit less than the size of the secret key, $K\_small[]$ (by no more than a polynomial factor). Also note, importantly, that the claim asserts an would-be attacker, not only is unable to characterize or determine the original key - or predict the unknown part (if any) of the plain text, but also that he is unable to discern the cipher text from random.

Claim 2. *Any quantum computer algorithm, using less running time than* $2^{reqsec/2}$*, has a probability of less than a value below that corresponding to* 1σ *(*$\approx 68.5\%$*) of determining whether SKREM was used to encrypt some known, arbitrarily chosen plain text* P*.*

Discussion: The reason for the reduction in the security strength against a quantum computer is obviously Grover's Algorithm [11]. If there was any doubt left, the discussion by Bernstein in [12] clarified this need. Although our key size is more than double *reqsec*, we choose to be conservative and consider the security strength of the secret to be just *reqsec*. Given Bennet et all [13], Grover's algorithm is asymptotically optimal for an arbitrary black box function - which is what we consider SKREM to be. Thus a reduction factor of 2 in the claimed strength should suffice. We, the author, consider humanity's current understanding of quantum physics incomplete and partially flawed. We believe it possible for physical phenomena to exist which allow computers, offering exponential

speed-ups in finding feasible arguments to black boxes, to be built. This is adds on top of speed-ups possible under existing theory, such as taking refuge near the event horizon of a black hole until a classical computer finishes breaking the encryption key, or sending the computer itself to a place where time flows relatively much faster. We believe that new encryption techniques will need to be developed, once this understanding is perfected. This highly speculative discussion is, however, outside the scope of the current paper.

In the "good old fashioned" practice of cryptology works, we provide no formal proofs of our claims. We do however claim that such formal proofs exist, which is, we believe, a far stronger assertion than simply saying they are true.

Claim 3. *There exists a formal proof of Claim 1.*

Claim 4. *There exists a formal proof of Claim 2.*

2.3 Performance Analysis of SKREM

Theorem 1. *The performance characteristics of SKREM are as follows:*

- *Total number of random words required:* $O(n * reqsec)$.
- *Minimum size of secret key:* $O(reqsec^2)$
- *Space complexity:* $O(n * reqsec)$.
- *Running time complexity:* $\widetilde{O}(n * reqsec^7)$

Proof. The proof is included in Appendix C, as Lemma 1. □

Discussion: The running time is thus polynomial in the security strength parameter and the plain text size and linear in just the latter. The required minimum secret key size is only quadric in the security strength parameter, which is $O(1)$ with regard to plain text size. The extra space required, besides the output and randomness well, can be implemented to go as low as the $O(2^{mtsize} * 2 * f * reqBitsPerKey)$. This is no more than $2^{O(reqsec)}$, which is $O(1)$ with regard to the plain text size.

3 Further Strengthening Security

In this section, we present a few important ideas to further strengthen security. We consider these to hold significant value with regard to any SKREM-like cipher, including the simplified version presented in Section 4. Consider the following.

1. **Pepper the plain text with random changes, making use of error-correction codes.** First, transform the plain-text into a redundant form by encoding it using error-correction codes. We suggest using simple codes, like Reed-Solomon [14], and quadrupling the size of the plain text with error correction data. After this, the randomness well T can be used to randomly alter a large portion of the words from the transformed plain text. Using Reed-Solomon codes should allow for modifications in about 30% of the data and still ensure it can be deciphered correctly. Not knowing where exactly a modification has taken place complicates the ability of an adversary to exploit any potential known plaintext advantage.
2. **Permute the indirection list** $Perm$ **using a more sophisticated approach.** For example, a longer permutation, determined by the sequence of burned locations, could be applied instead one involving a single swap currently used. One idea for such is to choose a fixed permutation of large order and to raise it to a power determined by the burned location before every application to $Perm$.

3. **Perform Key Extensions / Refreshes between encryption of successive plain text bits.** Refreshing some key elements (having them generate 1 new element instead of 8) after encoding each bit of plain text could add further uncertainty.

4. **Encrypt the plain text and / or the master tables using an existing symmetric cipher like AES beforehand.** This is a relatively common idea to any cipher. In our context, it can be regarded as a mean to degrade the capacity of the adversary to know the actual plain text. Like with any new scheme, the security of SKREM and SKREM-like ciphers should not rely on the security of this additional cipher. AES encryption is regarded in this context as mere obfuscation. Nevertheless, the design of SKREM allows such additional ciphers to be applied to both the plain text and the master tables safely, without degradation of security.

5. **Use a Chaos Machine or a CSPRNG instead of, or along side, modular algebra.** Currently, the transformations performed by $ExtendKey()$ are algebraically simple. However, SKREM could be altered to PUSH all emitted bits in a round into some Chaos Machine (see [6]). Whenever a certain value needs to be generated, its seed could simply be PULLed from this machine. If a full Chaos Machine is too slow, a different CSPRNG can employed in lieu of it.

6. **Change the location sampling method to something more sophisticated.** Currently, once the first location has been extracted from an atom, a simple, predictable, linear probing method is employed. However, a more sophisticated approach could be taken in $GetLocation()$ to translate l and x into an index in $Perm$. This could, for example, take the form of modular algebra again, combining the two in a manner similar to the expression in $ExtendKey()$. Alternatively, Chaos Machines, CSPRNGs or different one way function candidates (like the one in [1]) could be employed. Additionally, the size of one atom can be increased to something more suitable, irrespective of the size of a small master table, changing $ExtractJthLocation()$ accordingly. The current approach was chosen firstly to prevent the arguments pertaining to the security of SKREM depend on the security of this additional enhancement, which we feel is besides the core of the proposed scheme. Secondly, we wanted a practically feasible version to be easy to describe and understand, thus prevent it from resorting to sophisticated hacks and optimizations.

Finally, the following salt generation idea presents significant value, especially when secure transmissons are envisioned (rather than mere secure storage or computations): Transmit some number of grand master tables M - say 2^8. Among them, include the $M1$, $M2$ and $M3$ generated by $Encrypt()$. Each of these grand master tables will appear indistinguishable from truly random at any inspection (since indeed they are each actually truly random, taken apart) - for example when crossing a national border. The receiver will then try out all the $\binom{2^8}{3} \approx 2.7 * 10^6$ triplets and discover the correct plain-text, reasonably fast. For an attacker without knowledge of the secret key, discerning even if a single triplet is non-random is a very hard (claimed insurmountable) task. Performing this computationally expensive operation more than 2.7 million times should prove intractable.

4 Improving Performance

As powerful as SKREM is, its complexity (space in particular) - makes it impractical. In this section we present simplified version, resulting from choosing non-default values for the security parameters, meant to drastically improve performance characteristics.

It comes at the cost of only a constant factor reduction in the voracity with which we assert our educated security claims 1 and 2 (although we expect finding a formal proof for them to be harder). For lack of better name, we call it SKREMS (Short Key Random Encryption Machine Simplified). It is intended to be just an example of how practically feasible SKREM-like ciphers can be derived. We aim to provide 100-year security against the top existing conventional supercomputer, as of 2019. Based on publicly available statistics, this constrains the value of the security strength parameter to about $log(10^{30}) \approx 100$. Modifying SKREM to allow for defending against a quantum super-computer (thus requiring doubling this value) is possible, but it entails resorting to some refinements and hacks which are outside the scope of the current paper. Consider the following.

Algorithm 2. *Changes to SKREM entailed by the Encryption Scheme SKREMS.*

1: **STRUCT params_normal**
2: $reqsec \leftarrow 100$; $dmod \leftarrow 0$; $vrfy \leftarrow 0$; $mtsize \leftarrow 33$; $secrbase \leftarrow 4$; $secrtwo \leftarrow 0$; $secrbpp \leftarrow 20$; $secrbpb \leftarrow 5$; $secrbpn \leftarrow 6$; $ppx \leftarrow 2$; $w \leftarrow 8$; $bopf \leftarrow 1.1$
3: **STRUCT params_short : params_normal**
4: $dmod \leftarrow 1$; $secrtwo \leftarrow 1$; $ppx \leftarrow 5$;

Theorem 2. *The performance characteristics of SKREMS are the following.*

- *Total number of random words required for each of $M1$, $M2$ and $M3$:* $9030 * n$, $9030 * n$ *and* 2^{34} *respectively.*
- *Minimum size of secret key: 231 bits.*
- *Space complexity:* $18060 * n + 2^{34}$.
- *Running time complexity: $O(n)$, using about $n * 130/7$ CPU modular algebra operations over 127 bit numbers and about $18060 * n + 2^{34}$ disk and main memory operations.*

Proof. The proof is included in Appendix C, as Lemma 2. □

Discussion: Given than n is actually the number of bits, not bytes of plain text, the actual byte of storage per byte of plaintext is in fact $\approx 9030 * 8 \approx 71GB/MB$. Note that the secret key is just a factor of 2.31x larger than the theoretical minimum for this security strength. Also, note that it can be coded using about 42 alphanumeric characters. With some special training and a bit of practice such a key can be stored in a human brain.

As per the detailed discussion in Appendix D, based on performance data from [15] and [16], using two, commercially available, portable SSD drives of 2 TB each, one could encrypt about 28 MB (which is sufficient to contain about one minute of video, not to mention the full text of this paper) in about 1 day, using high-end, but still commodity hardware. The CPU running time can be reduced by a factor of $\approx 10^7$, using super-computer grade hardware, allowing the entire process to complete in about 3 minutes.

The above illustrates that practically feasible SKREM-like schemes exist. Ideas to further significantly improve performance also exist. Two, immediately obvious ones, involve using a hardcoded, precomputed list of primes of all required sizes (e.g. the first prime above $2^k + 11431 * x$ for increasing values of x) or performing algebraic operations in $GF(p^x)$ for some fixed small prime p (which we strongly dislike).

Finally, note that splitting the plain-text into blocks - one of the worst practices ever adopted in cryptography - is not required by SKREM-like ciphers: their complexities are linear in the plain text size. As such, problems associated with chaining methods (such as XTS [17]) are fully avoided. We strongly dislike having block ciphers be used to encrypt large plain-texts, as we feel chaining methods conceptually open up the output to code book attacks of various sorts, like the watermarking attack for CBC [18].

5 Hiding Multiple Plain-Texts In The Same Cipher Output

While SKREM-like schemes, including SKREMS consume $O(n)$ space, the constant factors are rather large. However, of the total space used, only a small portion, wf, is ever changed, namely $2 * f * n$ words in total. For SKREMS $wf \approx 0.03\%$, with only $\approx 2.08 * n$ words changing from the total of $9030 * n$. The fact $wf \ll 1$ can be used to encrypt more than one plain text into the same master table. Consider q plain texts, each of length n, to be encrypted in the same master table M. Let m denote the minimum size of a grand master table required for encryption of a single one. Decryption of each plain text will succeed so long as all the locations which are "touched" at encryption are left unmodified by the encryptions of the others.

Consider the following approach for achieving this desiderate, making use of a Universal Perfect Hashing (see [19]) scheme to avoid writing to forbidden areas. Two hashtables $H1$ and $H2$ are used, with $|M|$ available slots each. They associate to each of their slots a distinct location in M. After the encryption of the i-th plain-text, for all locations $\{loc\}$ touched during encryption, the values $\{(j, loc) | i < j < q\}$ are removed from $H1$, resulting in some up to j slots becoming unavailable. Similarly, all write-zone locations touched, $\{locw\}$ have the corresponding values $\{(j, locw) | i < j < q\}$ removed from $H2$. During key extension of the i-th plain-text, whenever a reference to a location loc in M is made, it is interpreted to mean location $H2[(i, loc)]$ instead. Furthermore, (i, loc) is removed from both $H1$ and $H2$ in $BurnLocation()$. During the encryption stage, a reference to location $locw$ is interpreted to mean location $H1[(i, locw)]$. Thereafter, $(i, locw)$ is similarly removed from both $H1$ and $H2$. This way, all q plain texts will write to disjoint zones, and they will all read from unaltered locations. If removal from $H2$ fails, a different random seed is retried for both hash tables and the process starts over (from the first plain-text). If a collision occurs for $H1$, it can be ignored: the slots must have already been invalidated by some prior plain-text. It is crucial however, that there be no collision at all between the values $(i, 0) \ldots (i, m)$ in either $H1$ or $H2$, for any i. This can be checked before the first encryption starts. The probability of collision for $H2$ should be rather small, since only $q * 2 * f * n = O(q * n)$ locations are removed. Having $|M| > O(m + wf * q * m)$ should suffice to get the probability of successful encryption for all q plain texts large enough. We leave calculation of the actual factor by which M needs to be increased to allow, with high probability, for precisely 0 collisions to occur in $H2$, for further research. Note that the hash tables $H1$ and $H2$ need to be common to all q plain-texts. The length of the random seed, needed for the theory of Universal Perfect Hashing to work, consists of just a few RAM words (over no less than $log(|M|)$ bits). As such, it can be encrypted alongside each pair of large secret keys using the respective plain text's K_small. The secret key size thus increases with only $log(q)$ bits, required to describe the index (or ID) i of the each plain text, from among the q possible.

We believe that a moderately ingenious application of the above could be used to modify SKREMS to allow it to encrypt 10-100x larger volumes of data using the same amount of space. We consider employing the above as a means to ensure "plausible deniability", of little practical value, both as a legal defense and as a defense against a torturous interrogator, considering that the cipher output, as per Claims 1 and 2 will appear random to an adversary anyway. Potentially, having it represent more than one plain text could play a role in informative intoxication operations involving double agents.

6 Usages In Lieu Of A One Way Function

Consider the transformation, defined using any SKREM-like scheme (like SKREM or SKREMS), mapping a secret key K_small to a plain text P resulting from decryption of a fixed, a priori chosen, cipher output M. For lack of better name, we shall call it SKREMOW (SKREM One Way). The above construction, while not formally a one way function itself (see [1]), suffices for almost all scenarios where one such is required. A detailed discussion is available in Appendix E.

7 Conclusions And Further Research

We have proposed two encryption schemes, which we claim offer the same level of security as the OTP in their strength parameter, while allowing for keys of constant size with regard to the plain text. For one of them, we further claim that formal proofs for these claims exist. Both schemes are claimed to make encrypted data indiscernible from random to any attacker having less computing power than 2^{reqsec} for some fixed, arbitrary parameter $reqsec$, thus achieving the main desiderate of encryption.

We conclude this paper here, noting that, even should our claims prove to be overly optimistic, the security of SKREM-like schemes still appears to be formidable and we expect them to become increasingly popular within cryptography. Thus, we believe introducing them to the scientific community now, if only for further scrutiny and cryptanalysis, to be an imperative regardless. In the "good old fashioned" tradition of cryptography works, we offer \$2.56 to the first 14 people who present an argument falsifying Claims 3 or 4 and an additional \$2.56 to the first 28 people who present an attack falsifying Claims 1 or 2.

8 Acknowledgments

Warm thanks to the three beautiful persons who inspired the author to strive to make the world a better place for them, consequently stirring his interest in cryptography. This paper would never have existed without them.

References

[1] Levin, L.A., "The tale of one-way functions", *Problems of Information Transmission*, **39(1)** (2003), 92-103.

[2] Daemen, J. and Rijmen, V., "The block cipher Rijndael", *In International Conference on Smart Card Research and Advanced Applications*, **Springer** (1998), 277-284.

[3] Rivest, R.L., Shamir, A. and Adleman, L., "A method for obtaining digital signatures and public-key cryptosystems", *Communications of the ACM*, **21(2)** (1978), 120-126.

[4] Zimand, M., "On the topological size of sets of random strings", *Mathematical Logic Quarterly*, **32(6)** (1986), 81-88.

[5] *Extracting the Kolmogorov complexity of strings and sequences from sources with limited independence. Zimand, M. arXiv preprint arXiv:0902.2141*, 2009.

[6] Armour, P.G., "The chaos machine", *Communications of the ACM*, **59(1)** (2015), 36-38.

[7] Shannon, C.E., "Prediction and entropy of printed English", *Bell system technical journal*, **30(1)** (1951), 50-64.

[8] *Standard deviation of n coin tosses. John Frain. Quora.*, 2018.

[9] *Chebyshev's & Empirical rules. CSUS. https://www.csus.edu/indiv/s/seria/lecturenotes/chebyshev.htm.*.

[10] Valiant, L.G., "The complexity of enumeration and reliability problems", *SIAM Journal on Computing*, **8(3)** (1979), 410-421.

[11] Grover, L.K., "A fast quantum mechanical algorithm for database search", *Proceedings, 28th Annual ACM Symposium on the Theory of Computing*, 1996, 212.

[12] Bernstein, D.J., "May. Grover vs. mceliece", *International Workshop on Post-Quantum Cryptography*, **Springer** (2010), 73-80.

[13] Bennett, C.H., Bernstein, E., Brassard, G. and Vazirani, U., "Strengths and weaknesses of quantum computing", *SIAM Journal on Computing*, **26(5)** (1997), 1510-1523.

[14] Reed, I.S. and Solomon, G., "Polynomial codes over certain finite fields", *Journal of the society for industrial and applied mathematics*, **8(2)** (1960), 300-304.

[15] Emmart, N., Zhengt, F. and Weems, C., "Faster Modular Exponentiation Using Double Precision Floating Point Arithmetic on the GPU", *IEEE 25th Symposium on Computer Arithmetic (ARITH)*, 2018, 130-137.

[16] *Interactive Latency - Latency Numbers Every Programmer Should Know (for 2019). Berkley.*, 2019.

[17] Martin, L., "XTS: A mode of AES for encrypting hard disks", *IEEE Security & Privacy*, **8(3)** (2010), 68-69.

[18] *Redundancy, the Watermarking Attack and its Countermeasures. Markus Gattol. https://www.markus-gattol.name/ws/dm-crypt_luks.html*, 2015.

[19] Dietzfelbinger, M., Karlin, A., Mehlhorn, K., Meyer Auf Der Heide, F., Rohnert, H. and Tarjan, R.E., "Dynamic perfect hashing: Upper and lower bounds", *SIAM Journal on Computing*, **23(4)** (1994), 738-761.

Hiding Data in Plain Sight: Towards Provably Unbreakable Encryption with Short Secret Keys and One-Way Functions (Annex)

Mircea-Adrian Digulescu[1,2]

December 2019

Abstract

This document is the Annex to the "Hiding Data in Plain Sight: Towards Provably Unbreakable Encryption with Short Secret Keys and One-Way Functions" paper, containing Appendixes A - F.

Keywords: Encryption, Provable Security, Chaos Machine, Truly Random, One Time Pad, One Way Function.

A Full pseudocode of SKREM

The following represents the full pseudocode of SKREM, save for the prime number generation routines, whose implementation is considered to be outside the scope of this paper.

Algorithm 3. *Full pseudocode of the Encryption Scheme SKREM.*

1: **STRUCT params_normal**
2: $reqsec \leftarrow 256$\{Security strength parameter: Logarithm of computing power available to an adversary\}
3: $dmod \leftarrow 0$\{Used to specify direct mode of key extension, using only one round\}
4: $vrfy \leftarrow 1$\{Specifies whether to return an error in case the security parameters do not offer the specified security strength\}
5: $mtsize \leftarrow 85$\{Used to determine the size of one small master table\}
6: $secrbase \leftarrow 4$\{Number of additional secret bits hidden in each key element to be used as base during key extension\}
7: $secrtwo \leftarrow 1$\{Used to specify that an additional random exponent is to be sampled from the grand master table, during key extension\}
8: $secrbpp \leftarrow 256$\{Length in bits of the offset used to select a random prime. Must be $\leq reqsec$.\}
9: $secrbpb \leftarrow 9$\{Number of bits, whose XOR is used to represent 1 emitted bit, during key extension\}
10: $secrbpn \leftarrow 9$\{Number of bits, whose XOR is used to represent 1 bit of plain text\}
11: $ppx \leftarrow 9$\{Number of key atoms per secret key\}
12: $w \leftarrow 8$\{Number of bits in a word in M and T\}
13: $bopf \leftarrow 2$\{Multiplication factor for the number of bits in a number l, necessary to sample it with (almost) uniform distribution from uniformly distributed individual bits\}
14: **END**
15:
16: **STRUCT params_short : params_normal**
17: $dmod \leftarrow 1$\{Override\}

END

METHOD Encrypt$(P, K1\_large[], K2\_large[], K\_small[], M1, M2, M3, T)$
$t \leftarrow 0$, $Ks \leftarrow \emptyset$, $Q \leftarrow \emptyset$
$Ks.AddRange($**bitsof**$(K1\_large[])$
$Ks.AddRange($**bitsof**$(K2\_large[])$
$(\_, M3, t) \leftarrow BasicEncryptDecrypt(0, |Ks|, Ks, K\_small[], M3, T, t, params\_short)$
$t \leftarrow ExtractBitsFromWords(T, t, Q, |P|)$
$(\_, M1, t) \leftarrow BasicEncryptDecrypt(0, |P|, Q, K1\_large[], M1, T, t, params\_normal)$
$(\_, M2, t) \leftarrow BasicEncryptDecrypt(0, |P|, P \oplus Q, K2\_large[], M2, T, t, params\_normal)$
return $(M1, M2, M3, |P|,$ **sizeof**$(K1\_large[]))$
END

METHOD Decrypt$(M1, M2, M3, n, szLKey, K_small)$
$Ks \leftarrow \emptyset$, $Q1 \leftarrow \emptyset$, $Q2 \leftarrow \emptyset$
$(Ks, \_, \_) \leftarrow BasicEncryptDecrypt(1, szLKey, Ks, M3, K\_small[], \emptyset, 0, params\_short)$
$K1\_large[] \leftarrow keyConcat[0 \ldots szLKey - 1]$
$K2\_large[] \leftarrow keyConcat[szKeyLarge \ldots 2 * szKeyLarge - 1]$
$(Q1, \_, \_) \leftarrow BasicEncryptDecrypt(1, n, Q1, M1, K1\_large[], \emptyset, 0, params\_normal)$
$(Q2, \_, \_) \leftarrow BasicEncryptDecrypt(1, n, Q2, M2, K2\_large[], \emptyset, 0, params\_normal)$
return $(Q1 \oplus Q2)$
END

METHOD BasicEncryptDecrypt$(mode, n, P, K[], M, T, t, params)$
{Performs encryption of P into M based on K[] or decryption from M into P based on K[], depending on mode}
{Initializations}
$Perm \leftarrow \emptyset$, $Vals \leftarrow \emptyset$, $NewVals \leftarrow \emptyset$, $Locs \leftarrow \emptyset$, $m \leftarrow |M|$, $j \leftarrow 0$, $x \leftarrow 0$, $bk \leftarrow 0$, $noEmittedBits \leftarrow 0$
$k \leftarrow (secrbase * (1 + secrtwo) + (ppx) * mtsize * bopf + secrbpp) * bopf + secrbpp$ {Size of a key element}
$g \leftarrow 1/((1 - 1/2^w))$
$f \leftarrow 1 + (g - 1) * 10$ {Number of pairs of words sufficient to generate one random bit}
$reqBitsPerKey \leftarrow k * 8 * (1 + secrtwo) * secrbpb$ {Number of bits required by a single key element, for extension}
$reqWords \leftarrow (reqBitsPerKey * (secrbpn/ppx) * n/7 + secrbpn * n) * 2 * f$ {Total number of words consumed by the algorithm}
$keyExtFactor \leftarrow 8$ {Number of new keys elements to which a single old key element is extended}
if $dmod = 1$ **then**
  $reqBitsPerKey \leftarrow k * (n * secrbpn/ppx) * (1 + secrtwo) * secrbpb$
  $reqWords \leftarrow (reqBitsPerKey + (secrbpn) * n) * 2 * f$
  $keyExtFactor \leftarrow secrbpn * n$
end if
if $m \leq 2 * reqWords$ **then**
  return error "Size of Grand Master Table is too modest"
end if
$maxMTsize \leftarrow PriorPrime(\lfloor \frac{m/2}{2*f} \rfloor)$
$mtSize \leftarrow Min(maxMTsize, NextPrime(2^{mtsize}))$ {Size of a small master table}
$noMTs \leftarrow Min(\lfloor \frac{m/2}{mtSize} \rfloor, 2 * f * reqBitsPerKey)$ {Total number of small master

```
   tables}
62: for i = 0 to noMTs * mtSize − 1 do
63:   Perm.Add(i) {Initialize Perm to be the identity permutation}
64: end for
65: Vals.AddRange(K[])
66: for i = 0 to secrbpn * n do
67:   EmittedBits[i] ← ∅
68: end for
69: {Security Parameters Validations}
70: if vrfy = 1 then
71:   if ppx * K[].Count * mtsize < reqsec + mtsize then
72:     return error "Number of secret pairs is too small for desired security level"
73:   end if
74:   if ppx * K[].Count * 4 < reqBitsPerKey/ppx then
75:     return error "Number of secret pairs is too small for desired security level"
76:   end if
77:   if mtsize + (mtsize − 1) * 2 + 3 < reqsec then
78:     return error "Size of small master tables is too modest for desired security level"
79:   end if
80:   if maxMTsize < NextPrime(2^mtsize) then
81:     return error "Size of Grand Master Tables is too modest to accommodate small
       master tables of desired size"
82:   end if
83:   if noMTs < 2 * f * reqBitsPerKey then
84:     return error "Size of Grand Master Tables is too modest to accommodate enough
       small master tables"
85:   end if
86: end if
87: {Key Extension}
88: while true do
89:   if Vals.Count ≥ secrbpn * n then
90:     break
91:   end if
92:   for i = 0 to Vals.Count − 1 do
93:     l ← ExtractJthLocation(Vals[i], k, j, params)
94:     lp1 ← GetLocation(l, x, noMTs, 2 * f * reqBitsPerKey, mtSize, params)
95:     lp2 ← GetLocation(l, x + 1, noMTs, 2 * f * reqBitsPerKey, mtSize, params)
96:     lm1 ← Perm[lp1]
97:     m ← BurnLocation(Perm, lp1, m, noMTs, mtSize)
98:     lm2 ← Perm[lp2]
99:     m ← BurnLocation(Perm, lp2, m, noMTs, mtSize)
100:    b2 ← GetBit(M[lm1], M[lm2])
101:    if b2 = null then
102:      continue
103:    end if
104:    if dmod = 0 and bk = i then
105:      bk ← (bk + 1) mod keyExtFactor * Vals.Count
106:    end if
107:    EmittedBits[bk].Add(b2)
108:    bk ← (bk + 1) mod keyExtFactor * Vals.Count
```

```
109:      noEmittedBits ← noEmittedBits + 1
110:    end for
111:    j ← j + 1
112:    if j ≥ ppx then
113:      j ← 0
114:      x ← x + 2
115:    end if
116:    if noEmittedBits ≥ reqBitsPerKey ∗ secrbpb ∗ keyExtFactor ∗ Vals.Count then
117:      z ← 0
118:      for i = 0 to Vals.Count − 1 do
119:        reqBits ← reqBitsPerKey/secrbpb
120:        for i = reqBits to EmittedBits[i].Count − 1 do
121:          EmittedBits[i][j mod reqBits] ← EmittedBits[i][j mod reqBits] ⊕
          EmittedBits[i][j]
122:        end for
123:        z ← ExtendKey(Vals[i], EmittedBits[i], z, NewVals, k, keyExtFactor, params)
124:      end for
125:      for i = 0 to NewVals.Count − 1 do
126:        EmittedBits[i].Clear()
127:      end for
128:      Vals ↔ NewVals
129:      NewVals.Clear()
130:      x ← 0 , j ← 0
131:    end if
132:  end while
133:  {Getting Locations For Encryption}
134:  while true do
135:    if EmittedBits[0].Count ≥ secrbpn ∗ n then
136:      break
137:    end if
138:    for i = 0 to Vals.Count − 1 do
139:      l ← ExtractJthLocation(Vals[i], k, j, params)
140:      lp1 ← GetLocation(l, x, noMTs, 2 ∗ f ∗ reqBitsPerKey, mtSize, params)
141:      lp2 ← GetLocation(l, x + 1, noMTs, 2 ∗ f ∗ reqBitsPerKey, mtSize, params)
142:      lm1 ← Perm[lp1]
143:      m ← BurnLocation(Perm, lp1, m, noMTs, mtSize)
144:      lm2 ← Perm[lp2]
145:      m ← BurnLocation(Perm, lp2, m, noMTs, mtSize)
146:      b2 ← GetBit(M[lm1], M[lm2])
147:      if b2 = null then
148:        continue
149:      end if
150:      EmittedBits[0].Add(b2)
151:      Locs.Add(lm1)
152:      Locs.Add(lm2)
153:    end for
154:    j ← j + 1
155:    if j ≥ ppx then
156:      j ← 0
157:      x ← x + 2
```

```
    end if
end while
{Encrypt / Decrypt}
for i = 0 to n − 1 do
    b ← 0
    for j = i to i + secrbpn do
        b ← b ⊕ EmittedBits[0][j]
    end for
    if mode = 1 then
        P.Add(b) {Decryption mode}
        continue
    end if
    if b2 = P[i] then
        continue
    end if
    reqLocBits ← GetReqGenerateBits(secrbpn, params)
    (locBits, t) ← ExtractBitsFromWords(T, t, Bits, reqLocBits)
    (l, _) ← GenerateRandomFromBits(locBits, 0, secrbpn, params)
    M[Locs[i ∗ 2 ∗ secrbpn + 2 ∗ l]] ↔ M[Locs[i ∗ 2 ∗ secrbpn + 2 ∗ l + 1]]
end for
{Return result}
return (P, M, t)
END

METHOD ExtendKey(K, ExtendBits, t, NewVals, k, count, params)
{Extends the key element K, by adding count new key elements to the NewVals list}
(_, KeyBits) ← ExpandKey(K, k, params)
lp ← 2^(k−secrbpp∗(bopf+1))
for i = 0 to count − 1 do
    x ← 17, y ← 14, v1 ← 17, v2 ← 28
    q ← 0
    (po, t) ← BuildValue(ExtendBits, t, secrbpp)
    p ← NextPrime(lp + po)
    (v1, q) ← BuildValue(KeyBits, q, secrbase)
    if secrtwo > 0 then
        (v2, q) ← BuildValue(KeyBits, q, secrbase)
    end if
    (x, t) ← GenerateRandomFromBits(ExtendBits, t, p, params)
    if secrtwo > 0 then
        (y, t) ← GenerateRandomFromBits(ExtendBits, t, p, params)
    end if
    newK ← (20 + [(v1 + 14)^(28+x) + (17 + v2)^(17+y) + 11431]^−1) mod p
    newBits ← ∅
    GetBits(po, secrbpp, newBits)
    GetBits(newK, k − secrbpp, newBits)
    (genK, _) ← BuildValue(newBits, 0, k)
    NewValus.Add(genK)
end for
return t
END
```

208:
209: **METHOD ExtractJthLocation**$(K, k, j, params)$
210: {Gets the value associated with the j-th atom used to index the small master tables, from key element K}
211: **static** $lastresult \leftarrow 14$ {Retains value between method calls}
212: $(\_, KeyBits) \leftarrow ExpandKey(K, k, params)$
213: $p \leftarrow NextPrime(2^{mtsize})$
214: $q \leftarrow secrbase * (1 + secr\_two) + j * mtsize * bopf$
215: $(l, \_) \leftarrow GenerateRandomFromBits(KeyBits, q, p, params)$
216: **if** $lastresult < 2$ **then**
217: $\quad lastresult \leftarrow 28$
218: **end if**
219: $l \leftarrow [17 + (lastresult^{l+28} + 14)^{-1}] \mod p$
220: $lastresult \leftarrow [17 + (lastresult^{l+20} + 11431)^{-1}] \mod p$
221: **return** l
222: **END**
223:
224: **METHOD BurnLocation**$(Perm, l, m, noMTs, mtSize)$
225: {Marks the location $Perm[l]$ from the grand master table as used and performs some minor shuffling of $Perm$, based on l}
226: $q \leftarrow \lfloor l/mtSize \rfloor$
227: $j1 \leftarrow \lfloor \frac{q}{2} \rfloor$
228: $j2 \leftarrow q + \lfloor \frac{noMTs-1-q}{2} \rfloor$
229: $l1 \leftarrow j1 * mtSize + \lfloor \frac{l}{2} \rfloor$
230: $l2 \leftarrow j2 * mtSize + l + \lfloor \frac{mtSize-l}{2} \rfloor$
231: $Perm[l1] \leftrightarrow Perm[l2]$
232: $Perm[l] \leftarrow m - 1$
233: $m \leftarrow m - 1$
234: **return** m
235: **END**
236:
237: **METHOD GetLocation**$(l, x, noMTs, orgNoMTs, mtSize, params)$
238: {Translates a location in $Perm$ over $orgNoMTs$ master tables into a location in $Perm$, with $noMTs$ master tables of size $mtSize$ each}
239: $y \leftarrow x + \lfloor l/mtSize \rfloor$
240: $l \leftarrow l \mod mtSize$
241: $l \leftarrow [l + \lfloor mtSize/\lceil \frac{orgNoMTs}{noMTs} \rceil \rfloor * (y \mod \lceil \frac{orgNoMTs}{noMTs} \rceil)] \mod mtSize$
242: $y \leftarrow y \mod noMTs$
243: **return** $y * mtSize + 1$
244: **END**
245:
246: **METHOD ExpandKey**$(K, k, params)$
247: {Expands key element K into its constituent parts so they can be used directly. These parts are the prime modulus used and the bits which represent all the atoms}
248: $kBits \leftarrow \emptyset$, $t \leftarrow 0$
249: $GetBits(K, k, kBits)$
250: $lp \leftarrow 2^{k-secrbpp*(bopf+1)}$
251: $(po, t) \leftarrow BuildValue(kBits, t, secrbpp)$
252: $p \leftarrow NextPrime(lp + po)$
253: $(kVal, \_) \leftarrow BuildValue(kBits, t, k - secrbpp)$

254: $noGenBits \leftarrow (k - secrbpp)/bopf$
255: **return** $(p, GenerateRandomBitsFromP(k, noGenBits, p))$
256: **END**
257:
258: **METHOD GetBit**$(w1, w2)$
259: {Returns a uniformly distributed bit based on two random, but potentially not uniformly distributed, words $w1$ and $w2$}
260: **if** $w1 = w2$ **then**
261: **return** *null*
262: **end if**
263: **if** $w1 < w2$ **then**
264: **return** 0
265: **end if**
266: **return** 1
267: **END**
268:
269: **METHOD ExtractBitsFromWords**$(S, t, Bits, noBits)$
270: {Extracts *noBits* uniformly distributed bits based on a random sequence S, starting from index t}
271: **while** $noBits > 0$ **do**
272: $b \leftarrow GetBit(S[t], S[t+1])$
273: $t \leftarrow t + 2$
274: **if** $b = null$ **then**
275: **continue**
276: **end if**
277: $Bits.Add(b)$
278: $noBits \leftarrow noBits - 1$
279: **end while**
280: **return** t
281: **END**
282:
283: **METHOD GenerateRandomFromBits**$(Bits, t, l, params)$
284: {Returns an almost uniformly distributed value between 0 and $l - 1$ based on some uniformly distributed random bits found in the sequence *Bits*, starting at index t}
285: $a \leftarrow GetReqGenerateBits(l, params)$
286: $x \leftarrow 0$
287: **for** $i = 0$ **to** $a - 1$ **do**
288: $x \leftarrow x + Bits[t] * 2^i$
289: $t \leftarrow t + 1$
290: **end for**
291: $step \leftarrow 2^a/l$ {Noninteger value with double precision}
292: $q \leftarrow \lfloor x/step \rfloor$
293: **if** $r \neq q$ **and** $q < l$ **and** $(q+1) * step - x \geq x - q * step$ **then**
294: $q \leftarrow q + 1$
295: **end if**
296: **return** (q, t)
297: **END**
298:
299: **METHOD GenerateRandomBitsFromP**(a, p, l)
300: {Returns the bits of an almost uniformly distributed value between 0 and $2^a - 1$ based

on some uniformly distributed value, between 0 and $l-1$}
301: $Bits \leftarrow \emptyset$
302: $step \leftarrow l/2^a l$ {Noninteger value with double precision}
303: $q \leftarrow \lfloor x/step \rfloor$
304: **if** $r \neq q$ **and** $q < l$ **and** $(q+1) * step - x \geq x - q * step$ **then**
305: $q \leftarrow q+1$
306: **end if**
307: $GetBits(q, a, Bits)$
308: **return** $Bits$
309: **END**
310:
311: **METHOD GetBits**$(val, noBits, Bits)$
312: {Adds $noBits$ bits from val, in least-significant first order to $Bits$ list}
313: $Bits \leftarrow \emptyset$
314: **while** $noBits > 0$ **do**
315: $Bits.Add(val \mod 2)$
316: $val \leftarrow \lfloor val/2 \rfloor$
317: $noBits \leftarrow noBits - 1$
318: **end while**
319: **return** $Bits$
320: **END**
321:
322: **METHOD BuildValue**$(Bits, t, noBits)$
323: {Builds a value from the next $noBits$ bits from $Bits$, starting at index t, given in least-significant first order}
324: $retVal \leftarrow 0$
325: **for** $i = 0$ **to** $noBits - 1$ **do**
326: $retVal \leftarrow retVal + Bits[t] * 2^i$
327: $t \leftarrow t+1$
328: **end for**
329: **return** $(retVal, t)$
330: **END**
331:
332: **METHOD GetReqGenerateBits**$(l, params)$
333: {Returns the number of uniformly distributed random bits required to generate an almost uniformly distributed value between 0 and $l-1$}
334: **return** $bopf * \lceil log(l) \rceil$
335: **END**
336:
337: **METHOD NextPrime**(val)
338: {Returns the smallest prime number $\geq val$}
339: **return** Min$\{p \mid$ p is prime and $p \geq val\}$
340: **END**
341:
342: **METHOD PriorPrime**(val)
343: {Returns the largest prime number $\leq val$}
344: **return** Max$\{p \mid$ p is prime and $p \leq val\}$
345: **END**
346:

Discussion: The values 14,28,20,11431, and 17 were chosen to be arbitrary beautiful con-

stants ≥ 2, which are used in such a way so as to have no impact on the security of the scheme. They can be replaced with anything else the reader finds more to his tastes, if desired.

B Detailed discussion of SKREM

This appendix includes the detailed discussion of the SKREM encryption algorithm, including indepth arguments concerning its claimed security.

SKREM proceeds as follows.

It encrypts the two large keys $K1\_large[]$ and $K2\_large[]$ using the secret key $K\_small[]$. This step is performed in order to allow for shorter keys in practice. When encrypting the larger secret keys, we felt we can scale back a bit on the stringency of the security requirements, considering that an adversary cannot gain insight via a known plain attack on this first encryption, taken independently from the other two, since it takes truly random data as input. SKREM then proceeds to encrypt the plain text, split in two via an OTP, into grand master tables $M1$ and $M2$. Each of these is, in effect, when taken alone, truly random. Only when they are analyzed in correlation, can they be considered just *almost* truly random. This significantly complicates attacks, since an adversary now has to consider both (independent) encryption processes in order to be able to even attempt to put his knowledge of the plain text to use.

Actual encryption is performed in $BasicEncryptDecrypt()$. The security strength parameter - commonly identified with the key length in bits in most other ciphers - is $reqsec$. This must be taken such that 2^{reqsec} steps is beyond what is tractable by any adversary, within the intended lifetime of the cipher text. Also, it should be taken relatively tight to optimize performance. Several other parameters can be modified to obtain a number of SKREM-like schemes. The actual key size is not precisely $reqsec$, but instead the value k (computed in $BasicEncryptDecrypt()$) times the minimum number of initial key elements, constrained by the security validations. This factor is taken, such that there exists a sufficient number of new key atoms, even for the first extension round, to allow for each one to depend on at most one bit from those emitted by each of the old key atoms. This ensures that an attacker able to guess the set of locations such a key atom generates (and thus their associated emitted bits) cannot reduce more than one bit of entropy from each individual new key atom (he might, in fact, be theoretically able to reduce one entropy bit from each and every one of them). Also, this factor is constrained so that, considering the chosen small master table sizes, the secret within the key has entropy greater than the computing power of the adversary - thus preventing brute force attacks. An additional $mtsize$ key atoms are required for the secret key K_small to prevent an attacker from reducing more entropy than allowed, by guessing atoms from this smaller key, instead of the two large ones. Constraining a sufficient number of key atoms within the larger keys (for example by guessing), could theoretically allow for a few random trials (a rather large amount actually) to be used to determine whether such constrains are plausible or not. However, the values of $mtsize$ and ppx are chosen so that these attempts would require beyond the computing power available to the adversary. Ultimately, the total size of the secret key, for all its elements together, is a function of only the security strength parameter. Specifically, it is no more than about quadric the latter, regardless of plain text size.

The security constrains put in place, and enforced via validation, are rather stringent. They are meant to be generously sufficient to allow us claim that SKREM can be proven to offer unbreakable security. Carefully relaxing them in practice, by setting $vrfy = 0$, will most likely still allow for unbreakable ciphers, although formally proving they possess

this quality may become harder.

Encryption proceeds as follows. Firstly, a key extension stage takes place, where the number of available k-bit key elements is extended to $secrbpn*n$. Finally, each atom of the resulting last key elements is used to determine an unused location within the grandmaster table. The values at these locations are than altered to represent the plain text, encoded in the pair-wise relative order of consecutive such. A number of *secrbpn* bits, represented by *secrbpn* pairs of locations, are XORed together to encode a single bit of P. When the existent bit does not correspond to the desired one, any of these pairs, randomly chosen, will have its values switched. The number of such pairs per plain-text bit, *secrbpn*, was chosen such that it will be beyond the computing power of the adversary to guess even one combination of such (based potentially on his knowledge of the plain-text bit it encodes).

Knowledge of the plain text could potentially be speculated only starting at the last stage, where actual encryption takes place. All transformations performed until then by SKREM are fully independent from it: the list of potential pairs of swappable locations from M is the same for any plain-text. As such, adaptive plain text attacks, as well as chosen cipher text attacks, should offer no noteworthy added benefit whatsoever, over simple known plain text attacks. They all involve having to determine some property of the list of locations used at encryption, using the knowledge that they must encode some given plain-text. Since M and T are truly random (and assuming the imperative that they never be reused is upheld), having them encode one particular plain text instead of another should make no difference at all.

During the key extension round, the following occurs. Each old key element is processed by $ExtractJthLocation$ (which uses some modular algebra to spice the result up a bit) to obtain a number of *ppx* locations between 0 and the size of a small master table. Each such location is used to reference $2 * f * reqBitsPerKey$ locations from the grand master table M, using $GetLocation$ and the indirection vector $Perm$. Currently, a simple linear probing scheme is employed. Each successive pair from locations in M encode 1 (truly random) bit, with probability $(1 - 1/2^w)$, failing $1/2^w$ of the time, when the sampled values are equal. In order to ensure with reasonable probability that we obtain the required number of $reqBitsPerKey$ from each key atom, the number of pairs of words is increased by a factor of f, which is taken to be close to the inverse of the former. The exact probability of failure for the entire process was not computed, but instead left for further research. The actual factor f used in SKREM is expected to be generously large, given that the probability the actual number of successfully generated bits strains too far from its expected value decreases very fast.

Once $reqBitsPerKey$ bits are generated for each of the new key elements (by all of the old key elements together), they are used to expand each such original key, into $keyExtFactor$ (8) new ones. The bits used for extension, while generated by locations determined deterministically from original key K_small and the grand master table M, are understood to appear truly random and independent to the adversary, given that each word of the truly random M is uniformly distributed and independent of all the rest. As such, picking any two locations will yield a perfect 50%/50% chance of their relative order (when holding distinct values) being either of the two possibilities. This holds even as locations are eliminated from M, potentially changing the ratio between the frequencies of each of the 2^w possible word values. Furthermore, without first consulting both locations, no guess, no matter how educated could ever hope to predict anything about the resulting bit. Since M is truly random, unless we got astronomically unlucky for it to be something predictable (e.g. all 0s), the distribution of values across almost all of its permutations should be just as truly random. One can image M as being constituted from a sequence of closed boxes, each holding a word. The key extension round proceeds to gradually open

them, choosing the exact order in which it does this, based in part on the encountered values. Any ensuing permutation suffix (namely even the immediate next location of a box to be opened), is actually fully independent (not only almost) from both the prefix and the values in the already opened boxes: by definition of truly random, each box was filled uniformly and independent of the others (in particular those opened as part of the prefix). Also, the transformations in *ExtractJthLocation* and *ExtendKey* are bijective: as such, if the bits received as input are random, so will the output be. Thus, whatever correlation, if any, between the prefix of a partial permutation and its suffix, will be only spurious (coincidental). By the definition of randomness using constructive martingales, we argue that if M and K were indeed truly random, then any such correlations will be non-existent (in practice, at most very short). Do note that, since the transformed M is in fact known to the adversary, the contents of the all boxes is known in advance. As such, he could try to speculate weaknesses in the method used to generate a new set of locations from the old one. No such weaknesses are believed to exist for the particular method used in SKREM. Finally, the entropy (in terms of Shannon Entropy [3]) of the ensuing permutation of M will also never exceed the entropy of the secret key. This however, is greater than the computing power assumed to be available to the adversary.

The manner in which a single k-sized key element K is extended is as follows. At least 1 (potentially 2) k-sized, almost truly random values x, y, are generated based on the grand master table M, and on part of the key atoms from the other key elements, excluding itself. By the discussion in the above paragraph, we expect each of these 8-16 values to be, effectively, indistinguishable from random by the adversary. We could, just as well, have taken atoms of the new key elements to be represented by these x-s and y-s directly. However, we prefer to entangle them with the unused portion of K. This is in order to complicate and/or eradicate the possibility of trying to use precomputations of the encrypted M in order to try to deduce some future key elements, only by examining M, without needing to go all the way back to the original key $K[]$ with the deductions. The exact extension formula $((20 + [(v1 + 14)^{28+x} + (17 + v2)^{17+y} + 11431]^{-1}) \mod p)$ consists effectively of the exponentiation of some primitive of $GF(p)$, determined by the unused portion of the key element K, to a random exponent based on the x,y values. When such an exponent is truly random and uniformly distributed, then the result is also truly random and uniformly distributed. The summing of two such exponentiations, as well as of the beautiful offset 11431 are meant to increase the difficulty in obtaining any succinct description of the set of potential preimages or images of the formula. The existence of an offset is also used to prevent a sequence of operations from simplifying to just one. Finally, the modular inverse operation was employed, given its apparent even-today-valid strength with regard to cryptographic usages. It is known that the AES [1] scheme relies fundamentally on it. Given the OSINT available to the author, it seems the security of modular inversion holds in practice. We chose to work in $GF(p)$ (the Galois Field of order p) with varying, (almost) truly random primes p-s, of suitable length, rather than $GF(2^c)$ for some fixed c for three reasons. Firstly, we feel that the structure of $GF(2^c)$ might be the object of intense study and precomputations, particularly given its popular usage in cryptography. By choosing to work in a large number of simple fields, we preempt such possibilities. Secondly, the structure of $GF(p)$, simple as it may be, appeals more to us than the potential unknowns hidden by $GF(2^c)$. Thirdly, a primitive of $GF(p)$ is very be easy to find: any value ≥ 2 will do. The above formula produces a random result whenever x is random, regardless of the other values. Finally, we do not reuse the portion of K consumed to generate the exponent, because we want to keep the entanglement between the bits emitted from key elements across different rounds to a minimum. While any such entanglement happens with regard to the locations of M sampled, not to their

actual values (and thus emitted bits), a correlation between the two exists, considering that the modified M is known.

The method used for bit emission, by some key atom A, is designed to deny the attacker the possibility to guess more than a single one for any new key atoms, resulting from extension, by making some guesses about as many old key atoms as his computing power allows. This is achieved as follows. An attacker could try to either guess any number of bits from old key atoms, up to *reqsec*. Even if he knows which ones to guess (based on the specific new atom bit he wants to know), these would reveal to him only a very small number of locations generated by old atoms. Note that a single bit error in the value of a location will result in a totally independent sequence of emitted bits. Also, since each new atom is meant to use only 1 bit from each of the old atoms, this can translate to at most 1 emitted being deduced for any of the new atoms (note that deducing 1 bit for just one, might constrain 1 bit for each of the remaining). This is just a theoretical possibility, which would imply, among other things, that the modular algebra formulas used offer no added security whatsoever. Using the default security parameters, having $bpb >= ppx$, at least ppx emitted bits are XORed together to determine a single, effectively used, bit from x or y. Without knowing all $\geq ppx$ such bits, no information at all can be obtained about even a single bit from x or y. Guessing more than ppx atoms is beyond the computing power of the adversary. Thus, even that hypothetical one bit advantage can be eliminated. For formal completeness of our argument, we should have taken $ppx = 1$, and have the number of keys elements in $K[]$ increased accordingly. We felt however that, given the modular algebra formulas involved, allowing a small number of ppx atoms per key element adds more uncertainty in practice. Ideas in Section 3 describe alternatives for replacing the simple, linear probing method used for location sampling, with something more sophisticated.

Finally, an attacker could try to guess not a single bit of a particular key element K', but deduce some property of K', based on the manner in which it was constructed and perhaps some guesses about a few key atoms. However, even from an information-theoretical perspective, he would need to be able to reduce, by his assumptions, at least $mtsize$ bits of entropy from set of old atoms in an extension round, for him to be able to constrain even one bit from one new atom. K' however, was either some original key element from $K[]$ - thus truly random - or was built in a manner which, we claim, produces a result indistinguishable from random, given the limited computing power available to the adversary. More precisely, we claim that no succinct characterization or useful property can be determined (except with astronomically low probability) by the adversary, with regard to the relationship between the set of new atoms and the set of old ones. This claim rests on the fact any such relationship will need to depend heavily on the existence of structure and order within M itself - which is, by the randomness requirement, excluded (except with astronomically low probability).

Guessing all the $mtsize$ bits of the location represented by an atom, could, under some pessimistic scenarios, potentially be used to determine one bit for all new atoms in an extension round. An attacker would then be able to take $\approx 2^{2*(mtsize-1)}$ random guesses for all the remaining bits (of which there are at least $mtsize - 1$ per atom - and this value could be increased as per the ideas in Section 3), in order to know with 1σ confidence ($\approx 68.5\%$) if he had guessed right (see [4] and [5]). Increasing confidence to 2σ ($\approx 95.4\%$) or 3σ ($\approx 99.7\%$) requires increasing the number of trials by a relatively small, constant factor. Getting to 1σ is the computationally expensive part. Each time such an experiment is performed, the set of plausible initial guesses (2^{mtsize}) is reduced to the corresponding fraction (e.g. 0.3% for 3σ). The correct guess (if such exists - meaning the cipher does in fact encrypt the plain-text) will show up significantly more frequently

in these reduced sets. The exact number of trials required for getting a certain required level of confidence (e.g. 1σ) that such a guess is correct is left for further research. We feel it should be around $\frac{mtsize}{log(prob)}$ where $prob$ is the probability of error for the number of σ-s used (e.g. 0.3% for 3σ). Nevertheless it will be greater than 1. As such, the security check requiring $mtsize$ to be no lesser than $\approx reqsec/3$ is sufficient from a theoretical perspective. In practice, it is likely that taking more than one location from a 2^{mtsize}-sized small master table for bits emission of a single atom, presents little-to-no problem, if done carefully. Also, the ideas in Section 3 offer improvement options for the location sampling method used, potentially allowing for small master tables of lesser size than 2^{mtsize} to be used securely. In order to allow for the desired level of voracity in our claims that SKREM offers security that can be formally proven to be unbreakable, we include this constrain. Note that guesses pertaining to a secret exponent x or y are no more useful than guesses pertaining to the atom they generate, having the same number of bits. Also note that any speculated (guessed) property of any round's set of atoms will be impossible to verify before the very last stage, where actual plain text encryption occurs. Before this, all intermediate values should appear just as valid to the adversary.

At the encryption stage, the attacker is faced with a set of $O(n)$ key elements, which, by the prior arguments, he should not be able to characterize in any useful manner. Each group of $secrbpn$ key elements are used to encode a single bit of plain text. By the same arguments as used for key extension, for $secrbrb >= ppx$ (as is the case with the default parameters in SKREM), an intractable number of locations would need to be guessed, for the attacker to be able to constrain the last-round key elements universe to account for even just 1 bit of plain text. The attacker should, thus, be unable to deduce any useful properties of this set, which would allow him to characterize the original key $K[]$ or discover some yet-unknown bit of plain text. In effect, the last round keys function like an OTP for the plain text. While not an OTP from an entropy perspective, the fact that no discernable features can be (we claim) determined for it, makes it a practical OTP. By introducing uncertainty about which pair of locations, from several possible, had its values changed in M, we feel we have significantly increased the theoretical difficulty of the problem of attacking SKREM, by making most attacks seeking a useful representation of the set of last round key elements, reduce to the counting problem #SKREM instead of merely the decision one. The counting problem is generally considered harder than the decision problem, as the difference in tractability between 2CNF-SAT and #2CNF-SAT illustrates [6].

Finally, any theoretical advantage an attacker could have - based on something we might have missed in our theory - is further diluted by the usage of two grand master tables, encrypted with distinct, independent keys. Other, less crucial, decisions add some additional uncertainty over the entire execution of the algorithm. For example, the simple shuffling of just 2 locations used in $BurnLocation()$, we expect to cause the indirection arrangement $Perm$ to stray, before the start of the encryption stage, pretty significantly from anything which is very simple to describe. The usage of non-constant base in $ExtractJthLocation()$, based on effectively all the atoms encountered during the entire course of execution of the algorithm also complicates matters notably.

The method used by $GenerateRandomFromBits()$ and $GenerateRandomBitsFromP()$ to convert from one uniform probability distribution to another merits some attention. It involves dividing the source universe into even slots, each having an associated value from the target universe. This conceptual division entails non-integer, rational thresholds. Since each threshold is a rational number, it may fall between integers. Occasionally a value at a border between two slots is encountered. Since determining the true correspondent would require more precision than available, we

simply employ some smart rounding. As such, each value in the target universe may have its distribution altered (increased or decreased) by inclusion / exclusion of a small part of it, situated around the two thresholds of the interval which represents it. Each such amounts to less than 0.5 (thanks to rounding), bringing the total to at most 1 value from the $\frac{2^{bopf*log(l)}}{l} = 2^{(bopf-1)*log(l)}$ representing it. The maximum magnitude of the relative error in distribution, for the suggested $bopf = 2$, considering numbers over 20 bits or more is thus $< 1/2^{20}$. This can be considered overly generous, given that the effective absolute error will be less than $\pm 1/2^{40}$. Also, a slight inclination towards certain location indexes does not translate into any change in the perfectly uniform outcome resulting from sampling them. This method, along with the one in $GetBit()$, were discovered by the author in the context of this paper. Nevertheless, we strongly suspect the likes of Marius Zimand or Leonid Levin are also aware of them.

C Proofs of the theorems

This appendix includes proofs of the theorems stated throughout the paper.

Lemma 1. *The performance characteristics of SKREM are as follows:*

- *Total number of random words required:* $O(n * reqsec)$.
- *Minimum size of secret key:* $O(reqsec^2)$
- *Space complexity:* $O(n * reqsec)$.
- *Running time complexity:* $\widetilde{O}(n * reqsec^7)$

Proof. Consider the encryption using large keys first. Each key element has at most $k = (secrbase*(1+secrtwo)+(ppx)*mtsize*bopf+secrbpp)*bopf+secrbpp$ bits. During key extension, precisely this many bits are required to be emitted for the each of the 8x new key elements, times $(1 + secrtwo)$ - the number of secret exponents per new key element. Finally, for each useful bit required, *scrbpb* bits need to be emitted. This brings the total to $[(secrbase * (1 + secrtwo) + (ppx) * mtsize * bopf + secrbpp) * bopf + secrbpp] * 8 * (1 + secrtwo) * secrbpb$ required bits to be emitted by each old key element. For each such required bit, $2 * f$ words are consumed from M.

The total number of key expansions occurring is no more than the sum

$$(1 + 8 + 16 + \ldots) = \sum_{x=1}^{log(secrbpn*n)/log(8)-1} 8^x$$

. The upper bound is given by the need to have $secrbpn * n$ last-round key elements, for representing the n bits of plain-text. This resolves to $(secrbpn * n)/7$ (see [7]).

During encryption, no key extensions occur. However, an extra $secrbpn * n$ bits are emitted. For each such bit, $2 * f$ locations in M are used. Finally, one time more the full number of words consumed need to remain untouched at the end, doubling this value. The grand total number of locations used by a large key is thus: $[[(secrbase * (1 + secrtwo) + (ppx) * mtsize * bopf + secrbpp) * bopf + secrbpp] * 8 * (1 + secrtwo) * secrbpb * (secrbpn)/7 + (secrbpn)] * 2 * f * n * 2$. All the values in this rather long expression are constants, except $mtsize$ which is constrained by the security validations to be at $O(reqsec)$. In terms of this parameter and n, the expression becomes $O([[(O(1) + O(reqsec) + O(1)) * O(1) + O(1)] * O(1) + O(1)] * O(n))$. This resolves to $O(n * reqsec)$, representing the total for the grand master table.

The required size of the large key is at most $O(reqsec)/O(1)$ times k, which is $O(k) * O(reqsec) = O(reqsec) * O(reqsec) = O(reqsec^2)$. We can assume $n > reqsec^2$, since this is trivially true in practice. The length of the plain text representing the two large keys, encrypted by Encrypt is no more than $O(reqsec^2 * reqsec) = O(reqsec^3)$, when using *params_short* no more stringent than *params_normal*, as is the case for the default values in SKREM.

The total complexity in the number of words consumed is by SKREM for the master tables is thus $O(n*reqsec)+2*O(reqsec^3) = O(n*reqsec)+O(n*reqsec) = O(n*reqsec)$.

The randomness well is polled to determine the OTP used for P, consuming $O(n)$ words. Another up to $2 * O(n) + 2 * O(reqsec^2)$ words are used to determine a value up to *secrbpn* which describes which locations in the grand master table to switch, per bit of plain-text encoded. The total required number of words for the randomness well is thus $O(n) + O(n) + O(reqsec^2) = O(n)$.

The size of the secret key needs to be no larger than the size of a large key, which was computed above to be $O(reqsec^2)$.

The operations performed in SKREM are either direct reads or writes from or to one of M or T, temporary storage of bits emitted for either key extension or expansion, some modular algebra on $O(k) = O(reqsec)$ bit numbers and accessing the $Perm$ indirection vector. Since no super linear data structures are employed, the total complexity is dominated by the amount of random words consumed, which is $O(n * reqsec)$.

Time complexity is as follows.

SKREM essentially consists of key expansions and extensions, plus a number of grand master table accesses, only a constant factor times the total number of words consumed. The latter is $O(n * reqsec)$. The total number of key extensions, as computed above is $O(n)$. The number of key expansions is less than the total number of words consumed, thus only $O(n * reqsec)$. For all word accesses, at most a constant number of algebraic operations are performed, with numbers of at most $O(k) = O(reqsec)$ bits. These are additions, subtractions, multiplications and division, modular exponentiation and modular inverse. Furthermore, for key extension, sampling a prime less than some value is performed. The maximum total number of sampled values is *secrbpp* which is less than *reqsec*, thus being $O(reqsec)$. Additions and subtractions take $O(k)$, multiplications takes $O(k*log(k))$ using Fast Fourier Transformation (FFT), while divisions take $O(k*log^2(k))$ using recursive division due to Brunikel and Ziegler [8]. Fast exponentiation involves at most $O(log(k))$ multiplications and divisions, taking $O(log(k)*k*log^2(k))$=$O(k*log^3(k))$. Modular inverse can be computed in $O(log(k))$ using the Extended Euclid algorithm. Generating the first prime beyond or below some value can be achieved by brute force trial and error, given the high density of prime numbers (established thanks to the Prime Number Theorem). The number of trials required is logarithmic in the upper bound. The probabilistic Miller-Rabin primality testing algorithm [9] can be used to obtain a probable prime which can then be verified using AKS algorithm [10]. Because AKS has rather large - even if still polynomial - complexity, it is likely in practice this verification step will be skipped. Generation of a prime less than $2^{O(reqsec)}$ bits takes no more than $O(log(2^{O(reqsec)})) = O(reqsec)$ applications of a prime testing algorithm on $O(k) = O(reqsec)$ bit numbers. If Rabin-Miller [9] is used with $O(reqsec)$ rounds, this adds up to $\widetilde{O}(reqsec * reqsec * reqsec^2) = \widetilde{O}(reqsec^4)$, when using FFT for multiplications. There are no more than $O(reqsec)$ rounds in total, thus a single prime number generation takes no more than $O(reqsec) * \widetilde{O}(reqsec^4) = \widetilde{O}(reqsec^5)$ in total. If AKS is used instead, this becomes $\widetilde{O}(reqsec^7)$. The total running time complexity is thus no more than $O(n) * \widetilde{O}(reqsec^7) + O(n * reqsec) * O(reqsec * log(reqsec)^3) =$

$\widetilde{O}(n * reqsec^7) + O(n * reqsec^2 * log^3(reqsec)) = \widetilde{O}(n * reqsec^7)$. □

Lemma 2. *The performance characteristics of SKREMS are the following.*

- *Total number of random words required for each of M1, M2 and M3:* $9030 * n$, $9030 * n$ *and* 2^{34} *respectively.*
- *Minimum size of secret key: 231 bits.*
- *Space complexity:* $18060 * n + 2^{34}$.
- *Running time complexity:* $O(n)$*, using about* $n * 130/7$ *CPU modular algebra operations over* 127 *bit numbers and about* $18060 * n + 2^{34}$ *disk and main memory operations.*

Proof. The total number of words used, expressed precisely in terms of the security parameters is given in the proof of Lemma 1.

Substituting the effective values used for the security parameters, results in getting $k < 127$ and the total number of words consumed for each of the grand master tables M1 and M2 to be $< 9030 * n$.

The minimum number of elements in each of the large keys $K1\_Large$ and $K2\_large$, required for them to satisfy security constraints is $k * 8 * (1 + secrtwo) * secrbpb = 127 * 8 * (1 + 0) * 5 = 5080$.

Thus, the total number of bits for a single large key is $5080 * k = 5080 * 127 = 645160$.

It is thus straightforward to note that the number of words complexity of encrypting the two of large keys, the using $dmod = 1$ is: $[[(secrbase * (1 + secrtwo) + (ppx) * mtsize * bopf + secrbpp) * bopf + secrbpp] * [secrbpn]/ppx * (1 + two) * secrbpb + secrbpn] * 2 * f * (2 * 655360) * 2 = [[(4*(1+1)+5*33*1.1+20)*1.1+20]*6/5*(1+1)*5+6]*2*1.04*2*645160*2$. This is $< 2^{34}$.

Each element of the $K\_small[]$ private key will have $(secrbase*(1+secrtwo)+(ppx)*mtsize*bopf+secrbpp)*bopf+secrbpp)$ bits, which is $(4*(1+1)+5*33*1.1+20)*1.1+20) = 231$ bits, with ceiling. Each such element represents 5 secret locations, over 33 bits each, bringing the secret size beyond *reqsec*. Thus, we can allow $K\_small[]$ to contain only one element. Its total size is thus just 231 bits.

As with SKREM, SKREMS uses no data structures of super-linear size. The space consumed is thus expected to be only slightly above the $18060 * n + 2^{34}$ words consumed for the three master tables combined.

In terms of time complexity, SKREMS can be optimized to perform only a constant number of modular algebraic operations per key, by using memoization. Doing so, will bring the number of modular arithmetic operations to just a small constant factor (which we estimate to be 10) times the total number of keys ever in existence. The number of key expansions, was computed in the proof of Lemma 1 as $(secrbpn * n)/7$, which, resolves to $n * 6/7$. An additional $n + 1$ key elements account for the initial and final stages, bringing the total to about $n * 13/7$, or $n * 130/7$ considering the chosen constant factor. The hardest operation performed is prime number identification, taking $O(k * log^3(k))$ using Rabin-Miller [9]. However, given the small seed of only 20 bits, these could be all precomputed. The outstanding operations take at most $O(k * log^2(k))$ each. The time consumed for performing modular algebra is added on top of the linear running time (with a constant factor close to 1) in the total number of touched words (which is close to the $18060 * n + 2^{34}$ computed above), consisting of both disk and memory operations. The total running time is thus $O(n)$. □

D Practical Feasibility of SKREMS using commodity hardware

Using commercially available storage hardware, consisting of 6 interconnected 120 TB hard-disks, providing 720 TB of volume, which, as of 2019, would cost around $80,000, one could encrypt up to 5 GB of plain-text, with a careful implementation. This is the equivalent the capacity of about 1 DVD.

On the other hand, using two commercially available, portable SSD drives, of 2 TB each, one could encrypt about 28 MB, which is sufficient to contain about one minute of video, not to mention the full text of this paper.

Consider the information in [11] - stating that about $150*10^3$ modular exponentiations can be performed per second, using 1024 bit numbers. Based on it, we expect at least $1500 * 10^3$ modular operations per second to be achievable, for 127 bit numbers. The total delay thus introduced for the 28 MB example above, using two cores, will be $< 28 * 2^{23} * 130/7/(1500 * 10^3) \approx 2908$ seconds, or ≈ 49 minutes. The disk operations, involving processing $\approx 71 * 28 + 16$ GB on two SSDs in parallel, could take up 1 year if random reads are allowed. However, a careful trick exists which allows for only sequential external reads to be performed. The total running time for disk operations thus becomes $(28 * 71 + 16) * 2^{30} * 62 * 10^{-12} \approx 134$ seconds (based on data from [12]), which is a bit over 2 minutes. Additional memory operations should take between $(28 * 71 + 16) * 2^{30} * 100 * 10^{-9} \approx 216 * 10^3$ (about 60 hours) and 1/25x times this value (about 2.4 hours) over two cores of commodity hardware (based also on data from [12]). Overall, we expect encryption/decryption of the 28 MB to complete within about a day, using high-end, but still commodity hardware. According to public statistics, using super-computer grade hardware available as of 2019 will decrease CPU running time by a factor of 10^7, making the entire process complete in less than 3 minutes (running time dominated by disk accesses).

E Usages In Lieu Of A One Way Function

Consider the transformation, defined using any SKREM-like scheme, including SKREM and SKREMS themselves, mapping a secret key K_small to a plain text P resulting from decryption of a fixed, a priori chosen, cipher output M. For lack of better name, we shall call it SKREMOW (SKREM One Way). It is defined as follows: $SKREMOW_M : \{0,1\}^k \rightarrow \{0,1\}^n$, where $n = |P|$, an arbitrary chosen size for the plain text (small enough to be encryptable in M), k is the size of the secret key |K_small| and let $m = |M|$ be the size of the three grand master tables $M1$, $M2$ and $M3$ combined (chosen to be of the minimum size, required for the specific plain text size n). Note that for a fixed key size k (chosen large enough to satisfy the security requirements for a specific *reqsec* security strength parameter) and a fixed plain text length n, SKREMOW actually defines a family of transformations: one for each of the 2^m possible cipher outputs. Reversing a function in this family is claimed to be, with high probability, provably impossible for any adversary having less computing power than 2^{reqsec} (2^{100} for SKREMS). SKREMOW thus fits the security requirements for almost all practical applications of one way functions.

There are two aspects preventing us from being able to formally consider $SKREMOW_M$ a one-way function. Firstly, it is one way only with high probability: in cases where M is chosen in an astronomically unlucky manner (for example it is all 0-s), it can be that $SKREMOW_M^{-1}$ could be easily computed. Discerning if a certain M is suitable (the quality of its randomness is sufficient) is a different, not at all trivial, task in itself. Secondly, the hardness of computing its inverse is only with regard to an ad-

versary with a fixed, constant amount of computing power available (namely 2^{reqsec}). For a sufficiently large constant (2^{reqsec} suffices), a simple $O(m)$ (linear in input) algorithm exists computing $SKREMOW_M^{-1}$ for any M. Although such an algorithm is intractable in practice, for a constant security strength parameter, $SKREMOW^{-1}$ does not meet the complexity class requirements of a one way function. This second aspect could be remedied by taking $reqsec$ to be a $\Omega(n)$. However, this would make the key size for SKREMOW too large for some practical applications.

Also note that SKREMOW is not bijective. When n is too small, the same plain text P could be decrypted using several secret keys - making SKREMOW non-injective. When n is large enough, there will exist bit sequences from $\{0,1\}^n$ which do not admit any key to decrypt them from the chosen, fixed cipher output M, thus making SKREMOW non-surjective. Nevertheless, taking n to be large enough, and restricting the codomain of SKREMOW to its image $Im_{SKREMOW_M}$, makes it become, with high probability, bijective.

The above construction suffices for almost all scenarios where one way functions are required in practice, even though SKREMOW is not, per see, a one such itself.

F Practical Considerations

We include some remarks, pertaining to practical usages of SKREM-like encryption schemes, such as SKREMS. Do note that implementation and actual usage pattern can make all the difference between perfect, unbreakable security and no security at all, for any scheme. The following is not intended to be even an exhaustive enumeration of potential pitfalls related to SKREM specifically. Nevertheless, we recommend paying particularly close attention to the following, before anything else.

Firstly, consider truly random numbers. As cryptologists, we love them, we hate them, we need them - all at the same time. The security of SKREM relies on the high quality of the randomness of the grand master table M and of the randomness well T. These should be harnessed from nature, rather than merely pseudo-randomly generated from a short seed. We can suggest the following as potential sources of entropy: (i) measuring the value of a single qubit passed through a single Hadamard gate on any quantum computer (and repeating the experiment for the number of bits required), (ii) sampling the phase of inbound solar radiation; (iii) measuring the interval between successive alpha-particle emissions during radioactive decay of certain atoms (if one such radioactive source is available to the user); (iv) sampling mouse gestures provided by the human user using a mouse; and (v) sampling a compressed, large patch of random text, provided by the human user using the keyboard. The output from several such sources should be combined using a Kolmogorov extractor (see [2]). Finally, automated statistical tests should be performed both on the originally sampled bits and on the cipher output, to detect cases of obvious deficiencies in the quality of the randomness.

Note that SKREM-like schemes can be used to create encrypted volumes - that is they allow for the plain text to be accessed randomly and even changed on the fly, given the secret encryption key. When used in this scenario, once any cipher output has been revealed to an attacker, the volume should be considered read-only. Generally speaking, the grand master tables and the randomness well should never be reused once a cipher output generated using them has been revealed, just as with an OTP.

All randomness used in the scheme, including the discarded randomness well T and the original grand master table M, must be kept secret. Revealing the grand master table to an adversary marks the moment he can start to preprocess it. A simple side by side comparison of the original master table and the cipher output will reveal the locations

used to encrypt the plain text. Thus, the original sources of entropy used, as well as their outputs, must not become available to the attacker.

We are aware of several types of attack against some practical usage scenarios and implementations of SKREM-like ciphers, which do not break the encryption scheme itself. We, the author, choose not to share them as part of this paper, as being outside its scope.

Of particular concern can be that we, the author, suspect that it is possible for the laws of physics concerning quantum phenomena to allow for a particular kind of quantum computer to be build which allows for exponential speedups in search over a the universe given by the preimage of a function (thus being exponentially faster than Grover's algorithm). We fear that, an implementation of such, might, someday, exist, which would allow for an arbitrary circuit with imposed output to be modeled using a quantum computer. Such a theoretical computer would then be run to determine its input. While processing, we fear it might be possible for all non-feasible states to essentially "fade out of existence" (e.g. have cumulative probability $< 1/3$ of describing a particular evolution of the quantum system). Thus, such a theoretical computer would be able to provide, with high probability, a feasible input to the arbitrary circuit in a single run (although the universe of possible inputs can be exponential in the size of the circuit). While existing quantum computer models cannot, to the best of our knowledge, come even close to such a feat, physical phenomena might exists which allow it. The quantum phenomenon called "path-of-minimum-energy", believed to be involved in photosynthesis, and harnessed by some for optimal route computation, is a particular inspiration for this concern of ours.

Finally, no encryption scheme, no matter how secure - even provably unbreakable - can protect against spyware installed on the devices where the encryption key is entered, or on the machines which perform actual encryption / decryption. Given that the we, the author, have used a personal, commercially available, not particularly secured, laptop to write this paper, we expects that there is a high probability some foreign intelligence services had already gained access to this research, by the time it was released by us.

References

[1] Daemen, J. and Rijmen, V., "The block cipher Rijndael", *In International Conference on Smart Card Research and Advanced Applications*, **Springer** (1998), 277-284.

[2] *Extracting the Kolmogorov complexity of strings and sequences from sources with limited independence. Zimand, M. arXiv preprint arXiv:0902.2141*, 2009.

[3] Shannon, C.E., "Prediction and entropy of printed English", *Bell system technical journal*, **30(1)** (1951), 50-64.

[4] *Standard deviation of n coin tosses. John Frain. Quora. https://www.quora.com/Whats-the-mean-variance-and-standard-deviation-of-a-coin-tossed-15-times*, 2018.

[5] *Chebyshev's & Empirical rules. CSUS. https://www.csus.edu/indiv/s/seria/lecturenotes/chebyshev.htm..*

[6] Valiant, L.G., "The complexity of enumeration and reliability problems", *SIAM Journal on Computing*, **8(3)** (1979), 410-421.

[7] *Wolfram Alpha. WolframAlpha.com.*

[8] Hasselström, K., "Fast division of large integers", *Department of Numerical Analysis and Computer Science, Royal Institute of Technology*, 2003.

[9] Rabin, M.O., "Probabilistic algorithm for testing primality", *Journal of number theory*, **12(1)** (1980), 128-138.

[10] Lenstra Jr, H.W. and Pomerance, C., "Primality testing with Gaussian periods", *FSTTCS*, 2002, 1.

[11] Emmart, N., Zhengt, F. and Weems, C., "Faster Modular Exponentiation Using Double Precision Floating Point Arithmetic on the GPU", *IEEE 25th Symposium on Computer Arithmetic (ARITH)*, 2018, 130-137.

[12] *Interactive Latency - Latency Numbers Every Programmer Should Know (for 2019). Berkley. https://people.eecs.berkeley.edu/ rcs/research/interactive_ latency.html*, 2019.