Bad directions in cryptographic hash functions

Daniel J. Bernstein[1,2], Andreas Hülsing[2],
Tanja Lange[2], and Ruben Niederhagen[2]

[1] Department of Computer Science
University of Illinois at Chicago
Chicago, IL 60607–7045, USA
djb@cr.yp.to
[2] Department of Mathematics and Computer Science
Technische Universiteit Eindhoven
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
andreas.huelsing@googlemail.com
tanja@hyperelliptic.org
ruben@polycephaly.org

Abstract. A 25-gigabyte "point obfuscation" challenge "using security parameter 60" was announced at the Crypto 2014 rump session; "point obfuscation" is another name for password hashing. This paper shows that the particular matrix-multiplication hash function used in the challenge is much less secure than previous password-hashing functions are believed to be. This paper's attack algorithm broke the challenge in just 19 minutes using a cluster of 21 PCs.

Keywords: symmetric cryptography, hash functions, password hashing, point obfuscation, matrix multiplication, meet-in-the-middle attacks, meet-in-many-middles attacks

1 Introduction

Under normal circumstances, the system protected the passwords so that they could be accessed only by privileged users and operating system utilities. But through accident, programming error, or deliberate act, the contents of the password file could occasionally become available to unprivileged users. . . . For example, if the password file is saved on backup tapes, then those backups must be kept in a physically secure place. If a backup tape is stolen, then everybody's password needs to be changed. Unix avoids this problem by not keeping actual passwords anywhere on the system. —"Practical UNIX & Internet Security" [**23**, p. 84], 2003

This work was supported by the National Science Foundation under grant 1018836, by the Netherlands Organisation for Scientific Research (NWO) under grant 639.073.005, and by the European Commission through the ICT program under contract INFSO-ICT-284833 (PUFFIN). Permanent ID of this document: 7c4f480d7f090d69c58b96437b6011b1. Date: 2015.02.23.

Consider a server that knows a secret password `11000101100100`. The server could check an input password against this secret password using the following `checkpassword` algorithm (expressed in the Python language):

```
def checkpassword(input):
  return int(input == "11000101100100")
```

But it is much better for the server to use the following `checkpassword_hashed` algorithm (see Appendix A for the definition of `sha256hex`):

```
def checkpassword_hashed(input):
  return int(sha256hex(input) == (
      "ba0ab099c882de48c4156fc19c55762e"
      "83119f44b1d8401dba3745946a403a4f"
    ))
```

It is easy for the server to write down this `checkpassword_hashed` algorithm in the first place: apply SHA-256 to the secret password to obtain the string `ba0...a4f`, and then insert that string into a standard `checkpassword_hashed` template. (Real servers normally store hashed passwords in a separate database, but in this paper we are not concerned with superficial distinctions between code and data.)

There is no reason to believe that these two algorithms compute identical functions. Presumably SHA-256 has a second (and third and so on) preimage of SHA-256(`11000101100100`), i.e., a string for which `checkpassword_hashed` returns 1 while `checkpassword` returns 0. However, finding any such string would be a huge advance in SHA-256 cryptanalysis. The `checkpassword_hashed` algorithm outputs 1 for input `11000101100100`, just like `checkpassword`, and outputs 0 for all other inputs that have been tried, just like `checkpassword`.

The core advantage of `checkpassword_hashed` over `checkpassword` is that it is **obfuscated**. If the `checkpassword` algorithm is leaked to an attacker then the attacker immediately sees the secret password and seizes control of all resources protected by that password. If `checkpassword_hashed` is leaked to an attacker then the attacker still does not see the secret password without solving a SHA-256 preimage problem: the loss of confidentiality does not immediately create a loss of integrity.

Obfuscation is a broad concept. There are many aspects of programs that one might wish to obfuscate and that are not obfuscated in `checkpassword_hashed`: for example, one can immediately see that the program is carrying out a SHA-256 computation, and that (unless SHA-256 is weak) there are very few short inputs for which the program prints 1. In the terminology of some recent papers (see Section 2), what is obfuscated here is the key in a particular family of "keyed functions", but not the choice of family. Further comments on general obfuscation appear below. We emphasize password obfuscation because it is an important special case: a widely deployed application using widely studied symmetric techniques.

1.1. State-of-the-art password hashing. Of course, some preimage problems can be efficiently solved. If the attacker knows (or correctly guesses) that the

secret password is a string of 14 digits, each 0 or 1, then the attacker can simply try hashing all 2^{14} possibilities for that string. Even worse, if the attacker sees many `checkpassword_hashed` algorithms from many users' secret passwords, the attacker can efficiently compare all of them to this database of 2^{14} hashes: the cost of multiple-target preimage attacks is essentially linear in the *sum* of the number of targets and the number of guesses, rather than the *product*.

There are three standard responses to these problems. First, to eliminate the multiple-target problem, the server *randomizes* the hashing. For example, the server might store the same secret password `11000101100100` as the following `checkpassword_hashed_salted` algorithm, where `prefix` was chosen randomly by the server for storing this password:

```
def checkpassword_hashed_salted(input):
  prefix = "b1884428881e20fe61c7629a0f71fcda"
  return int(sha256hex(prefix + input) == (
      "5f5616075f77375f1e36e2b707e55744"
      "91a308c39653afe689b7a958455e65d2"
    ))
```

The attacker sees the prefix and can still find *this* password using at most 2^{14} guesses, but the attacker can no longer share work across multiple targets. (This benefit does not rely on randomness: any non-repeating prefix is adequate. For example, the prefix can be chosen as a counter; on the other hand, this requires maintaining state and raises questions of what information is leaked by the counter.)

Second, the server chooses a hash function that is much more expensive than SHA-256, multiplying the server's cost by some factor F but also multiplying the attack cost by almost exactly F, if the hash function is designed well. The ongoing "Password Hashing Competition" [9] has received dozens of submissions of "memory-hard" hash functions that are designed to be expensive to compute even for an attacker manufacturing special-purpose chips to attack those particular functions.

Third, users are encouraged to choose passwords from a much larger space. A password having only 14 bits of entropy is highly substandard: for example, the recent paper [14] reports techniques for users to memorize passwords with four times as much entropy.

1.2. Matrix-multiplication password hashing: the "point obfuscation" challenge. A "point obfuscation" challenge was announced by Apon, Huang, Katz, and Malozemoff [7] at the Crypto 2014 rump session. "Point obfuscation" is the same concept as password hashing: see, e.g., [33] (a hashed password is a "provably secure obfuscation of a 'point function' under the random oracle model").

The challenge consists of "an obfuscated 14-bit point function on Dropbox": a 25-gigabyte program with the promise that the program returns 1 for one secret 14-bit input and 0 for all other 14-bit inputs. The goal of the challenge

is to determine the secret 14-bit input: "learn the point and you win!" An accompanying October 2014 paper [5] described the challenge as having "security parameter 60", where "security parameter λ is designed to bound the probability of successful attacks by $2^{-\lambda}$".

We tried the 25-gigabyte program on a PC with the following relevant resources: an 8-core 125-watt AMD FX-8350 "Piledriver" CPU (about $200), 32 gigabytes of RAM (about $400), and a 2-terabyte hard drive (about $100). The program took slightly over 4 hours for a single input. A brute-force attack using this program would obviously have been feasible but would have taken over 65536 hours worst-case and over 32768 hours on average, i.e., an average of nearly 4 years on the same PC, consuming 500 watt-years of electricity.

1.3. Attacking matrix-multiplication password hashing. In this paper we explain how we solved the same challenge in just 19 minutes using a cluster of 21 such PCs. The solution is `11000101100100`; we reused this string above as our example of a secret password. Of course, knowing this solution allowed us to compress the original program to a much faster `checkpassword` algorithm.

The time for our attack algorithm against a worst-case input point would have been just 34 minutes, about 5000 times faster than the original brute-force attack, using under 0.2 watt-years of electricity. Our current software is slightly faster: it uses just 29.5 minutes on 22 PCs, or 35.7 minutes on 16 PCs.

More generally, for an n-bit point function obfuscated in the same way, our attack algorithm is asymptotically $n^4/2$ times faster than a brute-force search using the original program. This quartic speedup combines four linear speedups explained in this paper, taking advantage of the matrix-multiplication structure of the obfuscated program. Two of the four speedups (Section 3) are applicable to individual inputs, and could have been integrated into the original program, preserving the ratio between attack time and evaluation time; but the other two speedups (Section 4) share work between separate inputs, making the attack much faster than a simple brute-force attack.

See Section 1.6 for generalizations to more functions.

1.4. Matrix-multiplication password hashing vs. state-of-the-art password hashing. It is well known that a 2^n-guess preimage attack against a hash function, cipher, etc. does not cost *exactly* 2^n times as much as a single function evaluation: there are always ways to merge small amounts of initial work across multiple inputs, and to skip small amounts of final work. See, for example, [34] ("Reduce the DES encryption from 16 rounds to the equivalent of ≈9.5 rounds, by shortcircuit evaluation and early aborts"), [29] ("biclique" attacks against various hash functions), and [13] ("biclique" attacks against AES).

However, one expects these speedups to become less and less noticeable for functions that have more and more rounds. For any state-of-the-art cost-C password-hashing function, the cost of a 2^n-guess preimage attack is very close to 2^nC. The matrix-multiplication function is much weaker: the cost of our attacks is far below 2^n times the cost of the best method known to evaluate the function.

Even worse, the matrix-multiplication approach has severe performance problems that end up limiting the number n of input bits. The "obfuscated point function" includes $2n$ matrices, each matrix having $n+2$ rows and $n+2$ columns, each entry having approximately $4((\lambda+1)(n+4)+2)^2 \log_2 \lambda$ bits; recall that λ is the target "security parameter". If λ is just 60 and n is above 36 then a single obfuscated password does not fit on a 2-terabyte hard drive, never mind the time and memory required to print and evaluate the function.

Earlier password-hashing functions handle a practically unlimited number of input bits with negligible slowdowns; fit obfuscated passwords into far fewer bits (a small constant times the target security level); allow the user far more flexibility to select the amount of time and memory used to check a password; and do not have the worrisome matrix structure exploited by our attacks.

1.5. Context: obfuscating other functions. Why, given the extensive hashing literature, would anyone introduce a new password-obfuscation method with unnecessary mathematical structure, obvious performance problems, and no obvious advantages? To answer this question, we now explain the context that led to the Apon–Huang–Katz–Malozemoff point-obfuscation challenge; we start by emphasizing that their goal was *not* to introduce a new point-obfuscation method.

Point functions are not the only functions that cryptographers obfuscate. Consider, for example, the following fast algorithm to compute the pqth power of an input mod pq, where p and q are particular prime numbers shown in the algorithm:

```
def rsa_encrypt_unobfuscated(x):
  p = 3797522793694367392280887275544562785456553663819 9
  q = 4009469095092088103068373529276146838921489972406 1
  pinv = 2363694910949459936056866756236854555993480451479 3
  qinv = 1558776194385864648453462293550080408668460822715 3
  return (qinv*q*pow(x,q,p) + pinv*p*pow(x,p,q)) % (p*q)
```

The following algorithm is not as fast but uses only the product pq:

```
def rsa_encrypt(x):
  pq = int("152260502792253336053561837813263742971806811496 13"
           "806886579084945801229632589528976540003506920061 39")
  return pow(x,pq,pq)
```

These algorithms compute exactly the same function $x \mapsto x^{pq} \bmod pq$, but the primes p and q are exposed in `rsa_encrypt_unobfuscated` while they are obfuscated in `rsa_encrypt`. This obfuscation is exactly the reason that `rsa_encrypt` is safe to publish. In other words, RSA public-key encryption is an obfuscation of a secret-key encryption scheme.

(Note that this size of pq is too small for serious security. The particular pq shown here was introduced many years ago as the "RSA-100" challenge and was factored in 1991. See [3]. One should take larger primes p and q.)

In a FOCS 2013 paper [25], Garg, Gentry, Halevi, Raykova, Sahai, and Waters proposed an obfuscation method that takes *any* fast algorithm A as input and "efficiently" produces an obfuscated algorithm Obf(A). The security goal for Obf is to be an "indistinguishability obfuscator": this means that Obf(A) is indistinguishable from Obf(A') if A and A' are fast algorithms computing the *same* function.

For example, if Obf is an indistinguishability obfuscator, and if an attacker can extract p and q from Obf(`rsa_encrypt_unobfuscated`), then the attacker can also extract p and q from Obf(`rsa_encrypt`), since the two obfuscations are indistinguishable; so the attacker can "efficiently" extract p and q from pq, by first computing Obf(`rsa_encrypt`). Contrapositive: if Obf is an indistinguishability obfuscator and the attacker *cannot* "efficiently" extract p and q from pq, then the attacker cannot extract p and q from Obf(`rsa_encrypt_unobfuscated`); i.e., Obf(`rsa_encrypt_unobfuscated`) hides p and q at least as effectively as `rsa_encrypt` does.

Another example, returning to symmetric cryptography: It is reasonable to assume that `checkpassword` and `checkpassword_hashed` compute the same function if the input length is restricted to, e.g., 200 bits. This assumption, together with the assumption that Obf is an indistinguishability obfuscator, implies that Obf(`checkpassword`) hides a $\leq$200-bit secret password at least as effectively as `checkpassword_hashed` does.

These examples illustrate the generality of indistinguishability obfuscation. In the words of Goldwasser and Rothblum [27], efficient indistinguishability obfuscation is "best-possible obfuscation", hiding everything that ad-hoc techniques would be able to hide.

There are, however, two critical caveats. First, it is not at all clear that the Obf proposal from [25] (or any newer proposal) will survive cryptanalysis. There are actually two alternative proposals in [25]: the first relies on multilinear maps [24] from Garg, Gentry, and Halevi, and the second relies on multilinear maps [22] from Coron, Lepoint, and Tibouchi. In a paper [19] posted early November 2014 (a week after we announced our solution to the "point obfuscation" challenge), Cheon, Han, Lee, Ryu, and Stehlé announced a complete break of the main security assumption in [22], undermining a remarkable number of papers built on top of [22]. The attack from [19] does not seem to break the application of [22] to point obfuscation (since "encodings of zero" are not provided in this context), but it illustrates the importance of leaving adequate time for cryptanalysis. A followup work by Gentry, Halevi, Maji, and Sahai [26] extends the attack from [19] to some settings where no "encodings of zero" below the "maximal level" are available, although the authors of [26] state that "so far we do not have a working attack on current obfuscation candidates".

Second, the literature already contains much simpler, much faster, much more thoroughly studied techniques for important examples of obfuscation, such as password hashing and public-key encryption. Even if the new proposals in fact provide indistinguishability obfuscation for more general functions, there is no reason to believe that they can provide competitive security and performance for

functions where the previous techniques apply. We would expect the generality of these proposals to damage the security-performance curve in a broad range of real applications covered by the previous techniques, implying that these proposals should be used only for applications outside that range.

The goal of Apon, Huang, Katz, and Malozemoff was to investigate "the practicality of cryptographic program obfuscation". Their obfuscator is not limited to point functions; it takes more general circuits as input. However, after performance evaluation, they concluded that "program obfuscation is still far from being deployable, with the most complex functionality we are able to obfuscate being a 16-bit point function"; see [5, page 2]. They chose a 14-bit point function as a challenge.

1.6. Attacking matrix-multiplication-based obfuscation of any function. The real-world importance of password hashing justifies focusing on point functions, but we have also adapted our attack algorithm to arbitrary n-bit-to-1-bit functions. Specifically, we have considered the method explained in [5] to obfuscate an arbitrary n-bit-to-1-bit function, and adapted our attack algorithm to this level of generality. For the general case, with u pairs of $w \times w$ matrices using n input bits, we save a factor of roughly $uw/2$ in evaluating each input, and a further factor of approximately $n/\log_2 w$ in evaluating all inputs. The $n/\log_2 w$ increases to $n/2$ for the standard input-bit order described in [5], but for an arbitrary input-bit order our attack is still considerably faster than a simple brute-force attack. See Section 8.

We comment that standard cryptographic hashing can be used to obfuscate general functions. We suggest the following trivial obfuscation technique as a baseline for future obfuscation challenges: precompute a table of hashes of the inputs that produce 1; add fake random hashes to pad the table to size 2^n (or a smaller size T, if it is acceptable to reveal that at most T inputs produce 1); and sort the table for fast lookups. This does not take polynomial time as $n \to \infty$ (for $T = 2^n$), but it nevertheless appears to be smaller, faster, and stronger than all of the recently proposed matrix-multiplication-based obfuscation techniques for every feasible value of n.

2 Review of the obfuscation scheme

Since the initial Obf proposal by Garg, Gentry, Halevi, Raykova, Sahai, and Waters [25] a lot of research was spent on finding applications and improving the proposed scheme. The challenge from [5] which we broke uses the relaxed-matrix-branching-program method by Ananth, Gupta, Ishai, and Sahai [4] to generate a size-reduced obfuscated program and combines it with the integer-based multilinear map (CLT) due to Coron, Lepoint, and Tibouchi [22]. As mentioned in Section 1, the recent CLT attack by Cheon, Han, Lee, Ryu, and Stehlé [19] relies on "encodings of zero" and therefore does not apply to this point-obfuscation scheme. Our attack will also work for other matrix-multiplication-type obfuscation schemes with a similar structure, and in particular we see no obstacle to

applying the same attack strategy with the Garg–Gentry–Halevi [24] multilinear map in place of CLT.

Most of the Obf literature does not state concrete parameters and does not present computer-verified examples. The first implementations, first examples, and first challenge were from Apon, Huang, Katz, and Malozemoff in [5], [6], and [7], providing an important foundation for quantifying and verifying attack performance.

The challenge given in [5] is an obfuscation of a point function, so we first give a self-contained description of these obfuscated point-function programs from the attacker's perspective; we then comment briefly on more general functions. For details on how the matrices below are constructed, we refer the reader to [4], [22], and of course [5]; but these details are not relevant to our attack.

2.1. Obfuscated point functions. A point function is a function on $\{0,1\}^n$ that returns 1 for exactly one secret vector of length n and 0 otherwise. The obfuscation scheme starts with this secret vector and an additional security parameter λ related to the security of the multilinear map.

The obfuscated version of the point function is given by a list of $2n$ public $(n+2) \times (n+2)$ matrices $B_{b,k}$ for $1 \le b \le n$ and $k \in \{0,1\}$ with integer entries; a row vector s of length $n+2$ with integer entries; a column vector t of length $n+2$ with integer entries; an integer p_{zt} (a "zero test" value, not to be confused with an "encoding of zero"); and a positive integer q. All of the entries and p_{zt} are between 0 and $q-1$ and appear random. The number of bits of q has an essentially linear impact upon our attack cost; [5] chooses the number of bits of q to be approximately $4((\lambda+1)(n+4)+2)^2 \log_2 \lambda$ for multilinear-map security reasons.

The obfuscated program works as follows:

- Take as input an n-bit vector $x = (x[1], x[2], \ldots, x[n])$.
- Compute the integer matrix $A = B_{1,x[1]} B_{2,x[2]} \cdots B_{n,x[n]}$ by successive matrix multiplications.
- Compute the integer $y(x) = sAt$ by a vector-matrix multiplication and a dot product.
- Compute $y(x)p_{\mathrm{zt}}$ and reduce mod q to the range $[-(q-1)/2, (q-1)/2]$.
- Multiply the remainder by $2^{2\lambda+11}$, divide by q, and round to the nearest integer. This result is by definition the matrix-multiplication hash of x.
- Output 0 if this hash is 0; output 1 otherwise.

We have confirmed these steps against the software in [6].

The matrix-multiplication hash here is reminiscent of "Fast VSH" from [20]. Fast VSH hashes a block of input as follows: use input bits to select precomputed primes from a table, multiply those primes, and reduce mod something. The matrix-multiplication hash hashes a block of input as follows: use input bits to select precomputed matrices from a table, multiply those matrices, and reduce mod something. The matrices are secretly chosen with additional structure, but we do not use that structure in our attack.

2.2. Initial security analysis. A straightforward brute-force attack determines the secret vector by computing the matrix-multiplication hash of all 2^n vectors x. Of course, the computation stops once a correct hash is found.

Unfortunately [5] and [7] do not include timings for $\lambda = 60$ and $n = 14$, so we timed the software from [6] on one of our PCs and saw that each evaluation took 245 minutes, i.e., $2^{45.74}$ cycles at 4GHz. As the code automatically used all 8 cores of the CPU, this leads to a total of $2^{48.74}$ cycles per evaluation. A brute-force computation using this software would take $2^{14} \cdot 2^{48.74} = 2^{62.74}$ cycles worst-case, and would take more than 2^{60} cycles for 85% of all inputs. For comparison, recall that the CLT parameters were designed to just barely provide $2^\lambda = 2^{60}$ security, although the time scale for the 2^{60} here is not clear. If the time scale of the security parameter is close to one cycle then the cost of these two attacks is balanced.

In their Crypto 2014 rump-session announcement [8], the authors declared this brute-force attack to be infeasible: "The great part is, it's only 14 bits, so you think you can try all 2 to the 14 points, but it takes so long to evaluate that it's not feasible." The authors concluded in [5, Section 5] that they were "able to obfuscate some 'meaningful' programs" and that "it is important to note that the fact that we can produce *any* 'useful' obfuscations at all is surprising".

We agree that a 500-watt-year computation is a nonnegligible investment of computer time (although we would not characterize it as "infeasible"). However, in Section 3 we show how to make evaluation two orders of magnitude faster, bringing a brute-force attack within reach of a small computer cluster in a matter of days. Furthermore, in Section 4 we present a meet-in-the-middle attack that is another two orders of magnitude faster.

2.3. Obfuscation of general functions and keyed functions. The obfuscation scheme in [4] transforms any function into a sequence of matrix multiplications. At every multiplication the matrix is selected based on a bit of the input x but usually the bits of x are used multiple times. For general circuits of length ℓ the paper constructs an oblivious relaxed matrix branching program of length $n\ell$ which cycles ℓ times through the n entries of x in sequence to select from $2n\ell$ matrices. In that case most of the matrices are obfuscated identity matrices but the regular access pattern stops the attacker from learning anything about the function.

Sometimes (as in the password-hashing example) the structure of the circuit is already public, and all that one wants to obfuscate is a secret key. In other words, the circuit computes $f_z(x) = \phi(z, x)$ for some secret key z, where ϕ is a publicly known branching program; the obfuscation needs to protect only the secret key z, and does not need to hide the function ϕ. This is called "obfuscation of keyed functions" in [4]. For this class of functions the length of the obfuscated program equals the length of the circuit for ϕ; the bits of x are used (and reused as often as necessary) in a public order determined by ϕ.

The designer can drive up the cost of brute-force attacks by including additional matrices as in the general case, but this also increases the obfuscation time, obfuscated-program size, and evaluation time.

3 Faster algorithms for one input

This section describes two speedups to the obfuscated programs described in Section 2. These speedups are important for constructive as well as destructive applications.

Combining these two ideas reduced our time to evaluate the obfuscated point function for a single input from 245 minutes to under 5 minutes (4 minutes 51 seconds), both measured on the same 8-core CPU. The authors of [6] have recently included these speedups in their software, with credit to us.

3.1. Cost analysis for the original algorithm. Schoolbook multiplication of the two $(n+2) \times (n+2)$ matrices $B_{1,x[1]}$ and $B_{2,x[2]}$ uses $(n+2)^3$ multiplications of matrix entries. Similar comments apply to all $n-1$ matrix multiplications, for a total of $(n-1)(n+2)^3$ multiplications of matrix entries.

This quartic operation count understates the asymptotic complexity of the algorithm for two reasons, even when the security parameter λ is treated as a constant. The first reason is that the number of bits of q grows quadratically with n. The second reason is that the entries in $B_{1,x[1]}B_{2,x[2]}$ have about twice as many bits as the entries in the original matrices, the entries in $B_{1,x[1]}B_{2,x[2]}B_{3,x[3]}$ have about three times as many bits, etc. The paper [5] reports timings for point functions with $n \in \{8, 12, 16\}$ for security parameter 52, and in particular reports microbenchmarks of the time taken for each of the matrix products, starting with the first; these microbenchmarks clearly show the slowdown from one product to the next, and the paper explains that "each multiplication increases the multilinearity level of the underlying graded encoding scheme and thus the size of the resulting encoding".

We now account for the size of the matrix entries. Recall that state-of-the-art multiplication techniques (see, e.g., [11]) take time essentially linear in b, i.e., $b^{1+o(1)}$, to multiply b-bit integers. The original entries have size quadratic in n, and the products quickly grow to size cubic in n. More precisely, the final product $A = B_{1,x[1]} \cdots B_{n,x[n]}$ has entries bounded by $(n+2)^{n-1}(q-1)^n$ and typically larger than $(q-1)^n$; similar bounds apply to intermediate products. More than $n/2$ of the products have typical entries above $(q-1)^{n/2}$, so the multiplication time is dominated by integers having size cubic in n.

The total time to compute A is $n^{7+o(1)}$ for constant λ, equivalent to $n^{5+o(1)}$ multiplications of integers on the scale of q. This time dominates the total time for the algorithm.

3.2. Intermediate reductions mod q. We do better by limiting the growth of the elements in the computation. The final result $y(x)p_{\mathrm{zt}}$ is in $\mathbf{Z}/q$, the ring of integers mod q, and is obtained by a sequence of multiplications and additions, so we are free to reduce mod q at any moment in the computation. Any of the initial integer multiplications has inputs at most $q-1$; we allow the temporary values to grow to at most $(n+2)(q-1)^2$ by computing the sum of the products for one entry and then reduce mod q. Thus any future multiplication also has its inputs at most $q-1$.

State-of-the-art division techniques take time within a constant factor of state-of-the-art multiplication techniques, so $(n+2)^2$ reductions mod q take asymptotically negligible time compared to $(n+2)^3$ multiplications. The number of bits in each intermediate integer drops from cubic in n to quadratic in n.

More precisely, the asymptotic speedup factor is $n/2$, since the original multiplication inputs had on average about $n/2$ times as many bits as q. We observe a smaller speedup factor for concrete values of n, mainly because of the overhead for the extra divisions.

The total time to compute $A \bmod q$ is $n^{6+o(1)}$ for constant λ, dominated by $(n-1)(n+2)^3 = n^4 + 5n^3 + 6n^2 - 4n - 8$ multiplications of integers bounded by q, inside $(n-1)(n+2)^2 = n^3 + 3n^2 - 4$ dot products mod q.

3.3. Matrix-vector multiplications. We further improve the computation by reordering the operations used to compute $y(x)$: specifically, instead of computing A, we compute

$$y(x) = \left(\cdots \left((sB_{1,x[1]})B_{2,x[2]}\right) \cdots B_{n,x[n]}\right) t.$$

This sequence of operations requires n vector-matrix products and a final vector-vector multiplication.

This combines straightforwardly with intermediate reductions mod q as above. The total time to compute $y(x) \bmod q$ is $n^{5+o(1)}$, dominated by $n(n+2)+1 = (n+1)^2$ dot products mod q.

4 Faster algorithms for many inputs

A brute-force attack iterates through the whole input range and computes the evaluation for each possible input until the result of the evaluation is 1 and thus the correct input has been found. In terms of complexity our improvements from Section 3 reduced the cost of brute-forcing an n-bit point function from time $n^{7+o(1)}2^n$ to time $n^{5+o(1)}2^n$ for constant λ, dominated by $(n+1)^2 2^n$ dot products mod q. This algorithm is displayed in Figure 4.1.

This section presents further reductions to the complexity of the attack. These share computations between evaluations of many inputs and have no matching speedups on the constructive side (which usually only evaluates at a single point at once and in any case cannot be expected to have related inputs).

4.2. Reusing intermediate products. Recall that Section 3 computes $y(x) = sB_{1,x[1]} \cdots B_{n,x[n]}t \bmod q$ by multiplying from left to right: the last two steps are to multiply the vector $sB_{1,x[1]} \cdots B_{n-1,x[n-1]}$ by $B_{n,x[n]}$ and then by t.

Notice that this vector does not depend on the choice of $x[n]$. By computing this vector, multiplying the vector by $B_{n,0}$ and then by t, and multiplying the same vector by $B_{n,1}$ and then by t, we obtain both $y(x[1], \ldots, x[n-1], 0)$ and $y(x[1], \ldots, x[n-1], 1)$. This saves almost half of the cost of the computation.

Similarly, we need only two computations of $sB_{1,x[1]}$ for the two choices of $x[1]$; four computations of $sB_{1,x[1]}B_{2,x[2]}$ for the four choices of $(x[1], x[2])$; etc. Overall there are $2+4+8+\cdots+2^n = 2^{n+1}-2$ vector-matrix multiplications here, plus 2^n

```
execfile('subroutines.py')
import itertools

def bruteforce():
  for x in itertools.product([0,1],repeat=n):
    L = s
    for b in range(n):
      L = [dot(L,[B[b][x[b]][i * w + j] for j in range(w)])
           for i in range(w)]
    result = solution(x,dot(L,t))
    if result: return result

print bruteforce()
```

Fig. 4.1. Brute-force attack algorithm, separately evaluating $y(x) \bmod q$ for each x, including the speedups of Section 3: reducing intermediate matrix products mod q (inside `dot`) and replacing matrix-matrix products with vector-matrix products. See Appendix A for definitions of subroutines.

final multiplications by t, for a total of $(n+2)(2^{n+1}-2)+2^n = (2n+5)2^n-2(n+2)$ dot products mod q.

To minimize memory requirements, we enumerate x in lexicographic order, maintaining a stack of intermediate products. We reuse products on the stack to the extent allowed by the common prefix between x and the previous x. In most cases this common prefix is almost the entire stack. On average slightly fewer than two matrix-vector products need to be recomputed for each x. See Figure 4.3 for a recursive version of this algorithm.

4.4. A meet-in-the-middle attack. To do better we change the order of matrix multiplication yet again, separating ℓ "left" bits from $n-\ell$ "right" bits:

$$y(x) = (sB_{1,x[1]} \cdots B_{\ell,x[\ell]})(B_{\ell+1,x[\ell+1]} \cdots B_{n,x[n]}t).$$

We exploit this separation to store and reuse some computations. Specifically, we precompute a table of "left" products

$$L[x[1],\ldots,x[\ell]] = sB_{1,x[1]} \cdots B_{\ell,x[\ell]}$$

for all 2^ℓ choices of $(x[1],\ldots,x[\ell])$. The main computation of all $y(x)$ works as follows: for each choice of $(x[\ell+1],\ldots,x[n])$, compute the "right" product

$$R[x[\ell+1],\ldots,x[n]] = B_{\ell+1,x[\ell+1]} \cdots B_{n,x[n]}t,$$

and then multiply each element of the L table by this vector.

Computing a single left product $sB_{1,x[1]} \cdots B_{\ell,x[\ell]}$ from left to right, as in Section 3, takes ℓ vector-matrix products, i.e., $\ell(n+2)$ dot products mod q. Overall the precomputation uses $\ell(n+2)2^\ell$ dot products mod q.

```
execfile('subroutines.py')

def reuseproducts(xleft,L):
  b = len(xleft)
  if b == n: return solution(xleft,dot(L,t))
  for xb in [0,1]:
    newL = [dot(L,[B[b][xb][i * w + j] for j in range(w)])
            for i in range(w)]
    result = reuseproducts(xleft + [xb],newL)
    if result: return result

print reuseproducts([],s)
```

Fig. 4.3. Attack algorithm sharing computations of intermediate products across many inputs x.

Computing a single right product $B_{\ell+1,x[\ell+1]} \cdots B_{n,x[n]}t$ from right to left (starting from t) takes $n-\ell$ matrix-vector products, for a total of $(n-\ell)(n+2)$ dot products mod q. The outer loop in the main computation therefore uses $(n-\ell)(n+2)2^{n-\ell}$ dot products mod q in the worst case. The inner loop in the main computation, computing all $y(x)$, uses just 2^n dot products mod q in total in the worst case.

The total number of dot products mod q in this algorithm, including precomputation, is $\ell(n+2)2^\ell+(n-\ell)(n+2)2^{n-\ell}+2^n$. In particular, for $\ell = n/2$ (assuming n is even), the number of dot products mod q simplifies to $n(n+2)2^{n/2}+2^n$.

For a traditional meet-in-the-middle attack, the outer loop of the main computation simply looks up each result in a precomputed sorted table. Our notion of "meet" is more complicated, and requires inspecting each element of the table, but this is still a considerable speedup: each inspection is simply a dot product, much faster than the vector-matrix multiplications used before.

We comment that taking ℓ logarithmic in n produces almost the same speedup with polynomial memory consumption. More precisely, taking ℓ close to $2\log_2 n$ means that $2^{n-\ell}$ is smaller than 2^n by a factor roughly n^2, so the term $(n-\ell)(n+2)2^{n-\ell}$ is on the same scale as 2^n. The table then contains roughly n^2 vectors, similar size to the original $2n$ matrices. Taking slightly larger ℓ reduces the term $(n-\ell)(n+2)2^{n-\ell}$ to a smaller scale. A similar choice of ℓ becomes important for speed in Section 8.2.

4.5. Combining the ideas. One can easily reuse intermediate products in the meet-in-the-middle attack. See Figure 4.6. This reduces the precomputation to $2^{\ell+1}-2$ vector-matrix multiplications, i.e., $(n+2)(2^{\ell+1}-2)$ dot products mod q. It similarly reduces the outer loop of the main computation to $(n+2)(2^{n-\ell+1}-2)$ dot products mod q.

The total number of dot products mod q in the entire algorithm is now $(n+2)(2^{\ell+1}+2^{n-\ell+1}-4)+2^n$. For example, for $\ell = n/2$, the number of dot products mod q simplifies to $4(n+2)(2^{n/2}-1)+2^n$.

```
execfile('subroutines.py')

l = n // 2

def precompute(xleft,L):
  b = len(xleft)
  if b == l: return [(xleft,L)]
  result = []
  for xb in [0,1]:
    newL = [dot(L,[B[b][xb][i * w + j] for j in range(w)])
            for i in range(w)]
    result += precompute(xleft + [xb],newL)
  return result

table = precompute([],s)

def mainloop(xright,R):
  b = len(xright)
  if b == n - l:
    for xleft,L in table:
      result = solution(xleft + xright,dot(L,R))
      if result: return result
    return
  for xb in [0,1]:
    newR = [dot(R,[B[n - 1 - b][xb][j * w + i] for j in range(w)])
            for i in range(w)]
    result = mainloop([xb] + xright,newR)
    if result: return result

print mainloop([],t)
```

Fig. 4.6. Meet-in-the-middle attack algorithm, including reuse of intermediate products, using $\ell = \lfloor n/2 \rfloor$ bits on the left and $n - \ell$ bits on the right.

This is not much smaller than the meet-in-the-middle attack without reuse: the dominant term is the same 2^n. However, as above one can take much smaller ℓ to reduce memory consumption. The reuse now allows ℓ to be taken almost as small as $\log_2 n$ without significantly compromising speed, so the precomputed table is now much smaller than the original $2n$ matrices.

If memory consumption is not a concern then one should compute both an L table and an R table, interleaving the computations of the tables and obtaining each LR product as soon as both L and R are known. For equal-size tables this means computing L_0, R_0, L_0R_0, L_1, L_1R_0, R_1, L_0R_1, L_1R_1, etc. This order of operations does not improve worst-case performance, but it does improve average-case performance. The same improvement has been previously applied to other meet-in-the-middle attacks: for example, Pollard applied this improvement

to Shanks's "baby-step giant-step" discrete-logarithm method. Compare [37, pages 419–420] to [35, page 439, top].

5 Parallelization

We implemented our attack for shared-memory systems using OpenMP and for cluster systems using MPI. In general, brute-force attacks are embarrassingly parallel, i.e., the search space can be distributed over the computing nodes without any need for communication, resulting in a perfectly scalable parallelization. However, for this attack, some computations are shared between consecutive iterations. Therefore, some cooperation and communication are required between computing nodes.

5.1. Precomputation. Recall that the precomputation step computes all 2^ℓ possible cases for the "left" ℓ bits of the whole input space. A non-parallel implementation first computes ℓ vector-matrix multiplications for $sB_{1,0} \cdots B_{\ell,0}$ and stores the first $\ell - 1$ intermediate products on a stack. As many intermediate products as possible are reused for each subsequent case.

For a shared-memory system, all data can be shared between the threads. Furthermore, the vector-matrix multiplications expose a sufficient amount of parallelism such that the threads can cooperate on the computation of each multiplication. There is some loss in parallel efficiency due to the need for synchronization and work-share imbalance.

For a cluster system, communication and synchronization of such a workload distribution would be too expensive. Therefore, we split the input range for the precomputation between the cluster nodes, compute each section of the precomputed table independently, and finally broadcast the table entries to all cluster nodes. For simplicity, we split the input range evenly which results in some workload imbalance. (On each node, the workload is distributed as described above over several threads to use all CPU cores on each node.) This procedure has some loss in parallel efficiency due to the fact that each cluster node separately performs k vector-matrix multiplications for the first precomputation in its range, due to some workload imbalance, and due to the final all-to-all communication.

5.2. Main computation. For simplicity, we start the main computation once the whole precomputed table L is available. Recall that a non-parallel implementation of the main computation first computes the vector $R[0, \ldots, 0] = B_{\ell+1,0} \cdots B_{n,0} t$ using $n - \ell$ matrix-vector multiplications, and multiplies this vector by all 2^ℓ table entries. It then moves to other possibilities for the "right" $n - \ell$ bits, reusing intermediate products in a similar way to the precomputation and multiplying each resulting vector $R[\ldots]$ by all 2^ℓ table entries.

For a shared-memory system, the computations of $R[\ldots]$ are distributed between the threads the same way as for the precomputation. However, vector-vector multiplication does not expose as much parallelism as vector-matrix multiplication. Therefore, we distribute over the threads the 2^ℓ independent vector-vector multiplications of each of the 2^ℓ table entries with $R[0, \ldots, 0]$. As in the

parallelization of precomputation, there is some loss of parallel efficiency due to synchronization and work-share imbalance for the vector-matrix multiplications and some loss due to work-share imbalance for the vector-vector multiplications.

For a cluster system we again cannot efficiently distribute the workload of one vector-matrix multiplication over several cluster nodes. Therefore, we distribute the search space evenly over the cluster nodes and let each cluster node compute its share of the workload independently. This approach creates some redundant work because each cluster node computes its own initial $R[\ldots]$ using $n-\ell$ matrix-vector multiplications.

6 Performance measurements

We used 22 PCs in the Saber cluster [12] for the attack. Each PC is of the type described earlier, including an 8-core CPU. The PCs are connected by a gigabit Ethernet network. Each PC also has two GK110 GPUs but we did not use these GPUs.

6.1. First break of the challenge. We implemented the single-input optimizations described in Section 3 and used 20 PCs to compute 2^{14} point evaluations for all possible inputs. This revealed the secret point 11000101100100 after about 23 hours. The worst-case runtime for this approach on these 20 PCs is about 52 hours for checking all 2^{14} possible input points. On 18 October 2014 we sent the authors of [5] the solution to the challenge, and a few hours later they confirmed that the solution was correct.

6.2. Second break of the challenge. We implemented the multiple-input optimizations described in Section 4 and the parallelization described in Section 5. Our optimized attack implementation found the input point in under 19 minutes on 21 PCs; this includes the time to precompute a table L of size 2^7. The worst-case runtime of the attack for checking all 2^{14} possible input points is under 34 minutes on 21 PCs.

6.3. Additional latency. Obviously "19 minutes" understates the real time that elapsed between the announcement of the challenge (19 August 2014) and our solution of the challenge with our second program (25 October 2014). See Table 6.4 for a broader perspective.

The largest deterrent was the difficulty of downloading 25 gigabytes. Whenever a connection broke, the server would insist on starting from the beginning ("HTTP server doesn't seem to support byte ranges"), presumably because the server stores all files in a compressed format that does not support random access. The same restriction also meant that we could not download different portions of the file in parallel.

To truly minimize latency we would have had to overlap the download of the challenge, the broadcast of the challenge to the cluster, and the computation, and of course our optimizations and software would have had to be ready first. In this context, the precompute-L-table algorithm in Section 4 has a latency advantage compared to a bit-reversed algorithm that precomputes an R table

Attack component	Real time
Initial procrastination	a few days
First attempt to download challenge (failed)	82 minutes
Subsequent procrastination	40 days and 40 nights
Fourth attempt to download challenge (succeeded)	about an hour
Original program [6] evaluating one input	245 minutes
Original program evaluating all inputs on one computer	(extrapolated) 7.6 years
Copying challenge to cluster (without UDP broadcasts)	about an hour
Reading challenge from disk into RAM	2.5 minutes
Our faster program evaluating one input	4.85 minutes
First successful break of challenge on 20 PCs	23 hours
Further procrastination ("this is fast enough")	about half a week
Our faster program evaluating all inputs on 21 PCs	34 minutes
Second successful break of challenge on 21 PCs	19 minutes
Our current program evaluating all inputs on 1 PC	444.2 minutes
Our current program evaluating all inputs on 22 PCs	29.5 minutes
Time for an average input point on 22 PCs	19.9 minutes
Successful break of challenge on 22 PCs	17.5 minutes

Table 6.4. Measurements of real time actually consumed by various components of complete attack, starting from announcement of challenge.

instead of an L table: the portion of the input relevant to L is available sooner than the portion of the input relevant to R.

6.5. Timings of various software components. We have put the latest version of our software online at http://obviouscation.cr.yp.to. We applied this software to the same challenge on 22 PCs. The software took a total time of 1769 seconds (29.5 minutes) to check all 2^{14} input points. An average input point was checked within 1191 seconds (19.9 minutes). The secret challenge point was found within 1048 seconds (17.5 minutes).

The rest of this section describes the time taken by various components of this computation.

Each vector-matrix multiplication took 15.577 s on average (15.091 minimum, 16.421 maximum), using all eight cores jointly. For comparison, on a single core, a vector-matrix multiplication requires about 115 s. Therefore, we achieve a parallel efficiency of $\frac{115s/8}{15.577s} \approx 92\%$ for parallel vector-matrix multiplication.

Each y computation took 8.986 s on average (7.975 minimum, 9.820 maximum), using a single core. Each y computation consists of one vector-vector multiplication, one multiplication by p_{zt} (which we could absorb into the precomputed table, producing a small speedup), and one reduction mod q.

On a single machine (no MPI parallelization), after a reboot to flush the challenge from RAM, the timing breaks down as follows:

1. Loading the matrices for "left" bit positions: 83.999 s.
2. Total precomputation of $2^7 = 128$ table entries: 4055.408 s.

   (a) Computing the first $\ell = 7$ vector-matrix products: 107.623 s.
4. Loading the matrices for "right" bit positions: 78.490 s.
5. Total computation of all 2^{14} evaluations: 22518.900 s.
   (a) Computing the first $n - \ell = 7$ matrix-vector products: 109.731 s.

Overall total runtime: 26654 s (444.2 minutes). From these computations, steps 1, 2a, 4, and 5a are not parallelized for cluster computation. The total timing breakdown on 22 PCs, after a reboot of all PCs, is as follows:

1. Loading the matrices for "left" bit positions: 89.449 s average (75.786 on the fastest node, 104.696 on the slowest node). With more effort we could have overlapped most of this loading (and the subsequent loading) with computation, or skipped all disk copies by keeping the matrices in RAM.
2. Total precomputation of $2^7 = 128$ table entries: 253.346 s average (217.893 minimum, 295.999 maximum).
   (a) Computing the first $\ell = 7$ vector-matrix products: 107.951 s average (107.173 minimum, 109.297 maximum).
3. All-to-all communication: 153.591 s average (100.848 minimum, 199.200 maximum); i.e., about 53 s average idle time for the busier nodes to catch up, followed by about 101 s of communication. With more effort we could have overlapped most of this communication with computation.
4. Loading the matrices for "right" bit positions: 85.412 s average (73.710 minimum, 97.526 maximum).
5. Total computation of all 2^{14} evaluations: 1097.680 s average (942.981 minimum, 1169.520 maximum).
   (a) Computing the first $n - \ell = 7$ matrix-vector products: 108.878 s average (107.713 minimum, 110.001 maximum).
6. Final idle time waiting for all other nodes to finish computation: 80.277 s average (0.076 minimum, 80.277 maximum).

Overall total runtime, including MPI startup overhead: 1769 s (29.5 minutes). The overall parallel efficiency of the cluster parallelization thus is $\frac{26654\,\mathrm{s}/22}{1769\,\mathrm{s}} \approx 68\%$. Steps 1, 2a, 3, 4, and 5a, totaling 545.281 s, are those parts of the computation that contain parallelization overhead (in particular the communication time in step 3 is added compared to the single-machine case). Removing these steps from the efficiency calculation results in a parallel efficiency of $\frac{(26654\,\mathrm{s}-380\,\mathrm{s})/22}{1769\,\mathrm{s}-545\,\mathrm{s}} \approx 98\%$, which shows that those steps are responsible for almost all of the loss in parallel efficiency.

7 Further speedups

In this section we briefly discuss two ideas for further accelerating the attack. We considered further implementation work to evaluate the concrete impact of these ideas, but decided that this work was unjustified, given that solving the existing challenge on our cluster took only 19 minutes.

7.1. Reusing transforms. One fast way to compute an m-coefficient product of two univariate polynomials is to evaluate each polynomial at the mth roots of 1 (assuming that there is a primitive mth root of 1 in the coefficient ring), multiply the values, and interpolate the product polynomial from the products of values. The evaluation and interpolation take only $\Theta(m \log_2 m)$ arithmetic operations using a standard radix-2 FFT (assuming that m is a power of 2), and multiplying values takes only m arithmetic operations.

More generally, to multiply two $w \times w$ matrices of polynomials where each entry of the output is known to fit into m coefficients, one can evaluate each polynomial at the mth roots of 1, multiply the matrices of values, and interpolate the product matrix. Note that intermediate values are computed in the evaluation domain; interpolation is postponed until the end of the matrix multiplication. The evaluation takes only $\Theta(w^2 m \log_2 m)$ arithmetic operations; schoolbook multiplication of the resulting matrices of values takes only $\Theta(w^3 m)$ arithmetic operations; and interpolation takes only $\Theta(w^2 m \log_2 m)$ arithmetic operations. The total is smaller, by a factor $\Theta(\min\{w, \log_2 m\})$, than the $\Theta(w^3 m \log_2 m)$ that would be used by schoolbook multiplication of the original matrices.

Smaller exponents than 3 are known for matrix multiplication, but there is still a clear benefit to reusing the evaluations (called "FFT caching" in [11]) and merging the interpolations (called "FFT addition" in [11]). Similar, somewhat more complicated, speedups apply to multiplication of integer matrices; see, e.g., [38, Table 17].

Obviously FFT caching and FFT addition can also be applied to matrix-vector multiplication, dot products, etc. For example, in the polynomial case, multiplying a $w \times w$ matrix by a length-w vector takes only $\Theta(w^2 m)$ arithmetic operations on values and $\Theta(wm \log_2 m)$ arithmetic operations for interpolation, if the FFTs of matrix entries have already been cached. Similarly, computing the dot product of two length-w vectors takes only $\Theta(wm)$ arithmetic operations on values and $\Theta(m \log_2 m)$ arithmetic operations for interpolation, if the FFTs of vector entries have already been cached.

The speedup here is applicable to both the constructive as well as the destructive algorithms in this paper. We would expect the speedup factor to be noticeable in practice, as in [38]. We would also expect an additional benefit for the attack: a high degree of parallelization is supported by the heavy use of arithmetic on values at independent evaluation points.

7.2. Asymptotically fast rectangular matrix multiplication. The computation of many dot products between all combinations of left vectors and right vectors in our point-obfuscation attack can be viewed as a rectangular matrix-matrix multiplication.

An algorithm of Coppersmith [21] multiplies an $N \times N$ matrix by an $N \times \lfloor N^{1/\beta} \rfloor$ matrix using just $N^{2+o(1)}$ multiplications of matrix entries, where $\beta = (5 \log 5)/(2 \log 2) < 6$. With the same number of multiplications one can multiply an $N \times \lfloor N^{1/\beta} \rfloor$ matrix by a $\lfloor N^{1/\beta} \rfloor \times N$ matrix. See [31] for context, and for techniques to achieve smaller β.

Substitute $N = \lceil w^\beta \rceil$, and note that $\lfloor N^{1/\beta} \rfloor = w$, to see that one can multiply a $\lceil w^\beta \rceil \times w$ matrix by a $w \times \lceil w^\beta \rceil$ matrix, obtaining $\lceil w^\beta \rceil^2$ results, using $w^{2\beta+o(1)}$ multiplications. Note that this is $w^{1+o(1)}$ times faster than computing separate dot products between each of the $\lceil w^\beta \rceil$ vectors in the first matrix and each of the $\lceil w^\beta \rceil$ vectors in the second matrix.

Our attack has 2^ℓ left vectors and $2^{n-\ell}$ right vectors, each of length $w = n+2$. Asymptotically Coppersmith's algorithm applies to any choice of ℓ between $\beta \log_2 w$ and $n/2$, allowing all of the dot products to be computed using just $w^{o(1)}2^n$ multiplications, rather than $w2^n$.

Fast matrix multiplication has a reputation for hiding large constant factors in the $w^{o(1)}$, and we do not claim a speedup here for any particular w, but asymptotically $w^{o(1)}$ is much faster than w. Our operation count also ignores the cost of additions, but we speculate that a more detailed analysis would show a similar improvement in the total number of bit operations.

8 Generalizing the attack beyond point functions

This section looks beyond point functions: it considers the general obfuscation method explained in [5] for any program.

Recall from Section 2 that for general programs the number of pairs of matrices, say u, is no longer tied to the number n of input bits: usually each input bit is used multiple times. Furthermore, each matrix is $w \times w$ and each vector has length w for some $w > n$, where the choice of w depends on the function and is no longer required to be $n + 2$.

The speedups from Section 3 rely only on the general matrix-multiplication structure, not on the pattern of accessing input bits. Reducing intermediate results mod q saves a factor approximately $u/2$. Using vector-matrix multiplication rather than matrix-matrix multiplication saves a factor w.

However, the attacks from Section 4 rely on having each input bit used exactly once. We cannot simply reorder the matrices to bring together the uses of an input bit: matrix multiplication is not commutative. Usually many of the matrices are obfuscated identity matrices, but the way the matrices are randomized prevents these matrices from being removed or reordered; see [5] for details.

This section explains two attacks that apply in more generality. The first attack allows cycling through the input bits any number of times, and saves a factor approximately $n/2$ compared to brute force. The second attack allows using and reusing input bits any number of times in any pattern, and saves a factor approximately $n/(2 \log_2 w)$ compared to brute force. The first attack is what one might call a "meet-in-many-middles" attack; the second attack does not involve precomputations. Both attacks exploit the idea of reusing intermediate products, sharing computations between adjacent inputs; both attacks can be parallelized by ideas similar to Section 5.

8.1. Speedup $n/2$ for cycling through input bits. Our first attack applies to any circuit obfuscated as explained in [5, Section 2.2.1]. The obfuscated circuit

is constructed to "cycle through each of the input bits $x_1, x_2, \ldots, x_n$ in order, m times", using $u = mn$ pairs of matrices. In other words, $y(x)$ is defined as

$$s(B_{1,x[1]} \cdots B_{n,x[n]})(B_{n+1,x[1]} \cdots B_{2n,x[n]}) \cdots (B_{(m-1)n+1,x[1]} \cdots B_{mn,x[n]})t.$$

Evaluating $y(x)$ for one x from left to right takes mn vector-matrix multiplications and 1 vector-vector multiplication, i.e., $uw + 1$ dot products mod q. A straightforward brute-force attack thus takes $(uw + 1)2^n$ dot products mod q.

One can split the sequence of mn matrices at some position ℓ, and carry out a meet-in-the-middle attack as in Section 4. However, this produces at most a constant-factor speedup once $m \geq 2$: either the precomputation has to compute products at most of the positions for all 2^n inputs, or the main computation has to compute products at most of the positions for all 2^n inputs, or both, depending on ℓ.

We do better by splitting the sequence of *input bits* at some position ℓ. This means grouping the matrix positions into two disjoint "left" and "right" sets as follows, splitting each input cycle:

$$\begin{aligned}
y(x) = {} & \big(sB_{1,x[1]} \cdots B_{\ell,x[\ell]}\big)\big(B_{\ell+1,x[\ell+1]} \cdots B_{n,x[n]}\big) \\
& \big(B_{n+1,x[1]} \cdots B_{n+\ell,x[\ell]}\big)\big(B_{n+\ell+1,x[\ell+1]} \cdots B_{2n,x[n]}\big) \\
& \qquad \vdots \\
& \big(B_{(m-1)n+1,x[1]} \cdots B_{(m-1)n+\ell,x[\ell]}\big)\big(B_{(m-1)n+\ell+1,x[\ell+1]} \cdots B_{mn,x[n]}t\big) \\
= {} & L_1[x[1], \ldots, x[\ell]]R_1[x[\ell+1], \ldots, x[n]] \\
& L_2[x[1], \ldots, x[\ell]]R_2[x[\ell+1], \ldots, x[n]] \\
& \qquad \vdots \\
& L_m[x[1], \ldots, x[\ell]]R_m[x[\ell+1], \ldots, x[n]]
\end{aligned}$$

where

$$\begin{aligned}
L_1[x[1], \ldots, x[\ell]] &= sB_{1,x[1]} \cdots B_{\ell,x[\ell]}, \\
L_i[x[1], \ldots, x[\ell]] &= B_{(i-1)n+1,x[1]} \cdots B_{(i-1)n+\ell,x[\ell]} && \text{for } 2 \leq i \leq m, \\
R_i[x[\ell+1], \ldots, x[n]] &= B_{(i-1)n+\ell+1,x[\ell+1]} \cdots B_{in,x[n]} && \text{for } 1 \leq i \leq m-1, \\
R_m[x[\ell+1], \ldots, x[n]] &= B_{(m-1)n+\ell+1,x[\ell+1]} \cdots B_{mn,x[n]}t.
\end{aligned}$$

We exploit this grouping as follows. We use $2^{\ell+1} - 2$ vector-matrix multiplications to precompute a table of the vectors $L_1[x[1], \ldots, x[\ell]]$ for all 2^ℓ choices of $x[1], \ldots, x[\ell]$, as in Section 4. Similarly, for each $i \in \{2, \ldots, m\}$, we use $2^{\ell+1} - 4$ matrix-matrix multiplications to precompute a table of the matrices $L_i[x[1], \ldots, x[\ell]]$ for all 2^ℓ choices of $x[1], \ldots, x[\ell]$. The tables use space for $(w + (m-1)w^2)2^\ell$ integers mod q.

After this precomputation, the outer loop of the main computation runs through each choice of $x[\ell+1], \ldots, x[n]$, computing the corresponding matrices $R_1[\ldots], \ldots, R_{m-1}[\ldots]$ and vector $R_m[\ldots]$. The inner loop runs through each

choice of $x[1], \ldots, x[\ell]$, computing each $y(x)$ by multiplying $L_1, R_1, \ldots, L_m, R_m$; each x here takes $2m-2$ vector-matrix multiplications and 1 vector-vector multiplication.

Overall the precomputation costs $((m-1)w^2+w)(2^{\ell+1}-2)-2(m-1)w^2$ dot products mod q; the outer loop of the main computation costs $((m-1)w^2+w)(2^{n-\ell+1}-2)-2(m-1)w^2$ dot products mod q; and the inner loop costs $((2m-2)w+1)2^n$ dot products mod q.

In particular, taking $\ell = n/2$ (assuming as before that n is even) simplifies the total cost to $4w(2^{n/2}-1)+2^n$ for $m=1$, exactly as in Section 4, and $4w((m-1)w+1)(2^{n/2}-1)+((2m-2)w+1)2^n-4(m-1)w^2$ for general m. Recall that brute force costs $(uw+1)2^n = (mnw+1)2^n$. For large n, large w, and $m \geq 2$, the asymptotically dominant term has dropped from $mnw2^n$ to $2mw2^n$, saving a factor of $n/2$.

The same asymptotic savings appears with much smaller ℓ, almost as small as $\log_2 w$. Beware that this does not make the tables asymptotically smaller than the original $2mn$ matrices for $m \geq 2$: most of the table space here is consumed by matrices rather than vectors.

8.2. Speedup $n/\log_2 w$ for any order of input bits. One can try to spoil the above attack by changing the order of input bits. A slightly different order of input bits, rotating positions in each round, is already stated in [4, Section 3, Claim 2, final formula], but it is easy to adapt the attack to this order. It is more difficult to adapt the attack to an order chosen randomly, or an order that combinatorially avoids keeping bits together. Varying the input order is not a new idea: see, e.g., the compression functions inside MD5 [36] and BLAKE [10]. Many other orders of input bits also arise naturally in "keyed" functions; see Section 2.

The general picture is that $y(x)$ is defined by the formula

$$y(x) = sB_{1,x[\mathrm{inp}(1)]}B_{2,x[\mathrm{inp}(2)]}\cdots B_{u,x[\mathrm{inp}(u)]}t$$

for some constants $\mathrm{inp}(1), \mathrm{inp}(2), \ldots, \mathrm{inp}(u) \in \{1, 2, \ldots, n\}$. As a first unification we multiply s into $B_{1,0}$ and into $B_{1,1}$, and then multiply t into $B_{u,0}$ and into $B_{u,1}$. Now $B_{1,0}, B_{1,1}, B_{u,0}, B_{u,1}$ are vectors, except that they are integers if $u = 1$; and $y(x)$ is defined by

$$y(x) = B_{1,x[\mathrm{inp}(1)]}B_{2,x[\mathrm{inp}(2)]}\cdots B_{u,x[\mathrm{inp}(u)]}.$$

We now explain a general recursive strategy to evaluate this formula for all inputs without exploiting any particular pattern in $\mathrm{inp}(1), \mathrm{inp}(2), \ldots, \mathrm{inp}(u)$. The strategy is reducing the number of variable bits in x by one in each iteration.

Assume that not all of $\mathrm{inp}(1), \mathrm{inp}(2), \ldots, \mathrm{inp}(u)$ are equal to n. Substitute $x[n] = 0$ into the formula for $y(x)$. This means, for each i with $\mathrm{inp}(i) = n$ in turn, eliminating the expression "$B_{i,x[n]}$" as follows:

- multiply $B_{i,0}$ into $B_{i+1,0}$ and into $B_{i+1,1}$ if $i < u$;
- multiply $B_{i,0}$ into $B_{i-1,0}$ and into $B_{i-1,1}$ if $i = u$;
- set $B_i \leftarrow B_{i+1}$, then $B_{i+1} \leftarrow B_{i+2}$, ..., then $B_{u-1} \leftarrow B_u$;

- reduce u to $u-1$.

Recursively evaluate the resulting formula for all choices of $x[1],\ldots,x[n-1]$. Then do all the same steps again with $x[n]=1$ instead of $x[n]=0$.

More generally, one can recurse on the two choices of $x[b]$ for any b. It is most efficient to recurse on the most frequently used index b (or one of the most frequent indices b if there are several), since this minimizes the length of the formula to handle recursively. This is equivalent to first relabeling the indices so that they are in nondecreasing order of frequency, and then always recursing on the last bit.

Once n is sufficiently small (see below), stop the recursion. This means separately enumerating all possibilities for $(x[1],\ldots,x[n])$ and, for each possibility, evaluating the given formula

$$y(x) = B_{1,x[\mathrm{inp}(1)]}B_{2,x[\mathrm{inp}(2)]}\cdots B_{u,x[\mathrm{inp}(u)]}$$

by multiplication from left to right. Recall that $B_{1,x[\mathrm{inp}(1)]}$ is actually a vector (or an integer if $u=1$). Each computation takes $u-1$ vector-matrix multiplications, i.e., $(u-1)w$ dot products mod q. (Here we ignore the extra speed of the final vector-vector multiplication.) The total across all inputs is $(u-1)w2^n$ dot products mod q.

To see that the recursion reduces this complexity, consider the impact of using exactly one level of recursion, from n down to $n-1$. If index n is used u_n times then eliminating each $B_{i,x[n]}$ costs $2u_n$ matrix multiplications, and produces a formula of length $u-u_n$ instead of u, so each recursive call uses $(u-u_n-1)w2^{n-1}$ dot products mod q. The bound on the total number of dot products mod q drops from $(u-1)w2^n$ to $4u_nw^2+(u-u_n-1)w2^n$, saving $u_nw2^n-4u_nw^2$. This analysis suggests stopping the recursion when 2^n drops below $4w$, i.e., at $n=\lceil\log_2 w\rceil+1$.

More generally, the algorithm costs a total of

$$4u_nw^2+8u_{n-1}w^2+16u_{n-2}w^2+\cdots+2^{n-\ell+1}u_{\ell+1}w^2+2^n(u_\ell+\cdots+u_1-1)w$$

dot products mod q if the recursion stops at level ℓ. We relabel as explained above so that $u_n\ge u_{n-1}\ge\cdots\ge u_1$, and assume $n>\ell$. The sum $u_\ell+\cdots+u_1$ is at most $\ell u/n$, and the sum $u_n+2u_{n-1}+4u_{n-2}+\cdots+2^{n-\ell-1}u_{\ell+1}$ is at most $2^{n-\ell}u/(n-\ell)$, for a total of less than $(4w2^{-\ell}/(n-\ell)+\ell/n)uw2^n$. Taking $\ell=\lceil\log_2 w\rceil+1$ reduces this total to at most $(4/(n-\lceil\log_2 w\rceil-1)+(\lceil\log_2 w\rceil+1)/n)uw2^n$.

For comparison, a brute-force attack against the original problem (separately evaluating $y(x)$ for each x) costs $(u-1)w2^n$. We have thus saved a factor of approximately $n/\log_2 w$.

References

[1] — (no editor), *53rd annual IEEE symposium on foundations of computer science, FOCS 2012, New Brunswick, New Jersey, 20–23 October 2012*, IEEE Computer Society, 2012. See [31].

[2] — (no editor), *54th annual IEEE symposium on foundations of computer science, FOCS 2013, 26–29 October, 2013, Berkeley, CA, USA*, IEEE Computer Society, 2013. See [25].

[3] — (no editor), *RSA numbers*, Wikipedia page (2014). URL: https://en.wikipedia.org/wiki/RSA_numbers. Citations in this document: §1.5.

[4] Prabhanjan Ananth, Divya Gupta, Yuval Ishai, Amit Sahai, *Optimizing obfuscation: avoiding Barrington's theorem*, in ACM-CCS 2014 (2014). URL: https://eprint.iacr.org/2014/222. Citations in this document: §2, §2, §2.3, §2.3, §8.2.

[5] Daniel Apon, Yan Huang, Jonathan Katz, Alex J. Malozemoff, *Implementing cryptographic program obfuscation*, version 20141005 (2014). URL: https://eprint.iacr.org/2014/779. Citations in this document: §1.2, §1.5, §1.6, §1.6, §2, §2, §2, §2, §2.1, §2.2, §2.2, §3.1, §6.1, §8, §8, §8.1.

[6] Daniel Apon, Yan Huang, Jonathan Katz, Alex J. Malozemoff, *Implementing cryptographic program obfuscation (software)* (2014). URL: https://github.com/amaloz/obfuscation. Citations in this document: §2, §2.1, §2.2, §3, §6.3, §A, §A, §A.

[7] Daniel Apon, Yan Huang, Jonathan Katz, Alex J. Malozemoff, *Implementing cryptographic program obfuscation (slides)*, Crypto 2014 rump session (2014). URL: http://crypto.2014.rump.cr.yp.to/bca480a4e7fcdaf5bfa9dec75ff890c8.pdf. Citations in this document: §1.2, §2, §2.2, §A.

[8] Daniel Apon, Yan Huang, Jonathan Katz, Alex J. Malozemoff, *Implementing cryptographic program obfuscation (video)*, Crypto 2014 rump session, starting at 3:56:25 (2014). URL: https://gauchocast.ucsb.edu/Panopto/Pages/Viewer.aspx?id=d34af80d-bdb5-464b-a8ac-2c3adefc5194. Citations in this document: §2.2.

[9] Jean-Philippe Aumasson, *Password Hashing Competition* (2013). URL: https://password-hashing.net/. Citations in this document: §1.1.

[10] Jean-Philippe Aumasson, Luca Henzen, Willi Meier, Raphael C.-W. Phan, *SHA-3 proposal BLAKE (version 1.3)* (2010). URL: https://www.131002.net/blake/blake.pdf. Citations in this document: §8.2.

[11] Daniel J. Bernstein, *Fast multiplication and its applications*, in [**15**] (2008), 325–384. URL: http://cr.yp.to/papers.html#multapps. Citations in this document: §3.1, §7.1, §7.1.

[12] Daniel J. Bernstein, *The Saber cluster* (2014). URL: http://blog.cr.yp.to/20140602-saber.html. Citations in this document: §6.

[13] Andrey Bogdanov, Dmitry Khovratovich, Christian Rechberger, *Biclique cryptanalysis of the full AES*, in Asiacrypt 2011 [**30**] (2011), 344–371. URL: https://eprint.iacr.org/2011/449. Citations in this document: §1.4.

[14] Joseph Bonneau, Stuart E. Schechter, *Towards reliable storage of 56-bit secrets in human memory*, in USENIX Security Symposium 2014 (2014), 607–623. URL: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/bonneau. Citations in this document: §1.1.

[15] Joe P. Buhler, Peter Stevenhagen (editors), *Surveys in algorithmic number theory*, Mathematical Sciences Research Institute Publications, 44, Cambridge University Press, 2008. See [11].

[16] Christian Cachin, Jan Camenisch (editors), *Advances in cryptology—EUROCRYPT 2004, international conference on the theory and applications of cryptographic techniques, Interlaken, Switzerland, May 2–6, 2004, proceedings*, Lecture Notes in Computer Science, 3027, Springer, 2004. ISBN ISBN 3-540-21935-8. See [33].

[17] Ran Canetti, Juan A. Garay (editors), *Advances in cryptology—CRYPTO 2013—33rd annual cryptology conference, Santa Barbara, CA, USA, August 18–22, 2013, proceedings, part I*, Lecture Notes in Computer Science, 8042, Springer, 2013. See [22].

[18] Anne Canteaut (editor), *Fast software encryption—19th international workshop, FSE 2012, Washington, DC, USA, March 19–21, 2012, revised selected papers*, Lecture Notes in Computer Science, 7549, Springer, 2012. ISBN 978-3-642-34046-8. See [29].

[19] Jung Hee Cheon, Kyoohyung Han, Changmin Lee, Hansol Ryu, Damien Stehlé, *Cryptanalysis of the multilinear map over the integers* (2014). URL: https://eprint.iacr.org/2014/906. Citations in this document: §1.5, §1.5, §1.5, §2.

[20] Scott Contini, Arjen K. Lenstra, Ron Steinfeld, *VSH, an efficient and provable collision-resistant hash function*, in Eurocrypt 2006 [**39**] (2006), 165–182. URL: https://eprint.iacr.org/2005/193. Citations in this document: §2.1.

[21] Don Coppersmith, *Rapid multiplication of rectangular matrices*, SIAM Journal on Computing **11** (1982), 467–471. Citations in this document: §7.2.

[22] Jean-Sebastien Coron, Tancrede Lepoint, Mehdi Tibouchi, *Practical multilinear maps over the integers*, in Crypto 2013 [**17**] (2013), 476–493. URL: https://eprint.iacr.org/2013/183. Citations in this document: §1.5, §1.5, §1.5, §1.5, §2, §2.

[23] Simson Garfinkel, Gene Spafford, Alan Schwartz, *Practical UNIX & Internet security*, 3rd edition, O'Reilly, 2003. Citations in this document: §1.

[24] Sanjam Garg, Craig Gentry, Shai Halevi, *Candidate multilinear maps from ideal lattices*, in Eurocrypt 2013 [**28**] (2012), 40–49. URL: https://eprint.iacr.org/2012/610. Citations in this document: §1.5, §2.

[25] Sanjam Garg, Craig Gentry, Shai Halevi, Mariana Raykova, Amit Sahai, Brent Waters, *Candidate indistinguishability obfuscation and functional encryption for all circuits*, in FOCS 2013 [**2**] (2013), 40–49. URL: https://eprint.iacr.org/2013/451. Citations in this document: §1.5, §1.5, §1.5, §2.

[26] Craig Gentry, Shai Halevi, Hemanta K. Maji, Amit Sahai, *Zeroizing without zeroes: Cryptanalyzing multilinear maps without encodings of zero* (2014). URL: https://eprint.iacr.org/2014/929. Citations in this document: §1.5, §1.5.

[27] Shafi Goldwasser, Guy N. Rothblum, *On best-possible obfuscation*, Journal of Cryptology **27** (2014), 480–505. Citations in this document: §1.5.

[28] Thomas Johansson, Phong Q. Nguyen (editors), *Advances in cryptology—EUROCRYPT 2013, 32nd annual international conference on the theory and applications of cryptographic techniques, Athens, Greece, May 26–30, 2013, proceedings*, Lecture Notes in Computer Science, 7881, Springer, 2013. ISBN 978-3-642-38347-2. See [24].

[29] Dmitry Khovratovich, Christian Rechberger, Alexandra Savelieva, *Bicliques for preimages: attacks on Skein-512 and the SHA-2 family*, in FSE 2012 [**18**] (2011), 244–263. URL: https://eprint.iacr.org/2011/286. Citations in this document: §1.4.

[30] Dong Hoon Lee, Xiaoyun Wang (editors), *Advances in cryptology—ASIACRYPT 2011, 17th international conference on the theory and application of cryptology and information security, Seoul, South Korea, December 4–8, 2011, proceedings*, Lecture Notes in Computer Science, 7073, Springer, 2011. ISBN 978-3-642-25384-3. See [13].

[31] François Le Gall, *Faster algorithms for rectangular matrix multiplication*, in FOCS 2012 [**1**] (2012), 514–523. URL: https://arxiv.org/abs/1204.1111. Citations in this document: §7.2.

[32] Donald J. Lewis (editor), *1969 Number Theory Institute: proceedings of the 1969 summer institute on number theory: analytic number theory, Diophantine problems, and algebraic number theory; held at the State University of New York at Stony Brook, Stony Brook, Long Island, New York, July 7–August 1, 1969*, Proceedings of Symposia in Pure Mathematics, 20, American Mathematical Society, 1971. ISBN 0-8218-1420-6. MR 47:3286. See [37].

[33] Benjamin Lynn, Manoj Prabhakaran, Amit Sahai, *Positive results and techniques for obfuscation*, in Eurocrypt 2004 [**16**] (2004), 20–39. Citations in this document: §1.2.

[34] Dag Arne Osvik, Eran Tromer, *Cryptologic applications of the PlayStation 3: Cell SPEED*, Workshop record of "SPEED—Software Performance Enhancement for Encryption and Decryption" (2007). URL: https://hyperelliptic.org/SPEED/slides/Osvik_cell-speed.pdf. Citations in this document: §1.4.

[35] John M. Pollard, *Kangaroos, Monopoly and discrete logarithms*, Journal of Cryptology **13** (2000), 437–447. Citations in this document: §4.5.

[36] Ronald L. Rivest, *The MD5 message-digest algorithm*, RFC 1321 (1992). URL: https://tools.ietf.org/html/rfc1321. Citations in this document: §8.2.

[37] Daniel Shanks, *Class number, a theory of factorization, and genera*, in [**32**] (1971), 415–440. MR 47:4932. Citations in this document: §4.5.

[38] Joris van der Hoeven, Grégoire Lecerf, Guillaume Quintin, *Modular SIMD arithmetic in Mathemagix* (2014). URL: https://arxiv.org/abs/1407.3383. Citations in this document: §7.1, §7.1.

[39] Serge Vaudenay (editor), *Advances in cryptology—EUROCRYPT 2006, 25th annual international conference on the theory and applications of cryptographic techniques, St. Petersburg, Russia, May 28–June 1, 2006, proceedings*, Lecture Notes in Computer Science, 4004, Springer, 2006. ISBN 3-540-34546-9. See [20].

A Subroutines

The `sha256hex` function is defined as the following wrapper around Python's `hashlib`:

```
import hashlib

def sha256hex(input):
  return hashlib.sha256(input).hexdigest()
```

In other words, `sha256hex` returns the hexadecimal representation of the SHA-256 hash of its input.

The software from [**6**] stores nonnegative integers on disk in a self-delimiting format defined by GMP's `mpz_out_raw` function (for integers that fit into $2^{32}-1$ bytes): a 4-byte big-endian length b precedes a b-byte big-endian integer. The following `load_mpz` and `load_mpzarray` functions parse the same format and return `gmpy2` integers:

```
import struct
import gmpy2
```

```
def mpz_inp_raw(f):
  bytes = struct.unpack('>i',f.read(4))[0]
  if bytes == 0: return 0
  return gmpy2.from_binary('\x01\x01' + f.read(bytes)[::-1])

def load_mpzarray(fn,n):
  f = open(fn,'rb')
  result = [mpz_inp_raw(f) for i in range(n)]
  f.close()
  return result

def load_mpz(fn):
  return load_mpzarray(fn,1)[0]
```

Integers such as w, q, the s entries, etc. are then read from files as `gmpy2` integers:

```
w = load_mpz('size')
pzt = load_mpz('pzt')
q = load_mpz('q')
nu = load_mpz('nu')
s = load_mpzarray('s_enc',w)
t = load_mpzarray('t_enc',w)
n = w - 2
B = [[load_mpzarray('%d.%s' % (b,xb),w * w)
      for xb in ['zero','one']]
     for b in range(n)]
```

The file names are specified by the software from [6]. The challenge announced in [7] used an older version of the software from [6], using file name `x0` instead of `q`, so we copied `x0` to `q`. Note that the B array is indexed $0, 1, \ldots, n-1$ rather than $1, 2, \ldots, n$.

The `dot` function computes a dot product of two length-w vectors and reduces the result mod q:

```
def dot(L,R):
  return sum([L[i]*R[i] for i in range(w)]) % q
```

The `solution` function takes x and $y(x)$ as input, and returns x as a string of ASCII digits if the output of the corresponding obfuscated program is 1:

```
def solution(x,y):
  y *= pzt
  y %= q
  if y > q - y: y -= q
  if y.bit_length() > q.bit_length() - nu:
    return ''.join([str(xb) for xb in x])
```