

Electronic Voting: Mathematics and Electronic Security Algorithms that might be useful in election systems that try to achieve public auditability

Mike Amling
Programmer/Senior Analyst
RMCis Corporation Lisle, IL
mamling@rmcis.com

November 17, 2004

1 Hashes

A hash function takes as input a sequence of zero or more bits and produces as output a fixed number of bits. One good hash function is SHA-1, which produces 160 bits of output [3]. The properties that good hash functions have are

- 1.) Collision resistance; It should not be possible to find two different inputs that produce the same output.
- 2.) Preimage resistance; It should not be possible to find an input that produces a given output.
- 3.) Second preimage resistance; Given an input, it should not be possible to find another input that produces the same output. If a hash has collision resistance, it will also have second preimage resistance.

Note that if there are more than 2^{160} different inputs, there must be at least two that collide, that is, there must be at least two that produce the same output. The collision resistance property is not that colliding inputs do not exist, but merely that they cannot be found. SHA-1 has been around over ten years, and no two inputs that produce the same output have ever been found [4].

How would hashes be used for elections? One use would be in verifying that some software we're running here in Vanderburgh County is identical to the software that was certified. The certified software consists, no doubt, of some set of files, each of which is a sequence of bits. The direct way to compare the software to that which was certified would be to carry around a copy of each of the certified files on a read-only medium such as CD-ROM and compare them bit by bit with the files on the machine that's running. A somewhat more manageable way would be to carry around a list of the certified files' names and their SHA-1 hashes. Since most of the files are considerably longer than 160 bits, this saves space, and a list of hashes, due to the pre-image resistance, does not give

away the contents of the files. Then the SHA-1 hashes of the files on the machine that's running can be compared with those on the list. If the hash of a file is equal to the hash of some certified file, then the files' contents must be the same, since different inputs would not produce the same output. Of course, we still have to worry that the list of hashes that we're comparing to has not been modified since it was generated. Later we'll see how this can be simplified further.

2 Public key cryptography

Let's say Pete Caithamer and I agree on some thousand-bit integer, g . How big is a thousand-bit integer? If written out in decimal it would be some 300 decimal digits. If you printed 80 digits per line, it would occupy three complete lines and part of a fourth. It's not the kind of number you can do arithmetic on just with pencil and paper.

If Pete squares g , the result is roughly 2000 bits long, possibly a little shorter [1]. If he cubes g , the result is roughly 3000 bits long. Let's say Pete raises g to some large power. The result can be quite long. If he raises g to some large power, x , and gives me the result, it doesn't take too long for me to find x . My first guess at x would be the length of the number he gives me, divided by one thousand. I could take that first guess, raise g to that power, and compare what I get with the number that Pete gave me. If the result I get is lower than the number Pete gave me, I can increase my guess. If it's higher, I can decrease my guess. A modern computer would find x is less than a second.

Now let's make the problem of finding x a lot harder. Pete again raises g to the power x , but instead of giving me the result, he gives me the remainder when the result is divided by some thousand-bit prime number p . Instead of giving me g^x itself, Pete only gives me $g^x \bmod p$. Now neither of the two methods I used before works.

I can't get a first approximation to x by dividing the length of what Pete gives me by one thousand. The remainder is itself at most one thousand bits long, so dividing it by one thousand doesn't get me anywhere near x , which can be quite large.

And if I have a guess for x , I can't tell if my guess is too high or too low. As my guess increases, the value of $g^{\text{guess}} \bmod p$ jumps around in the range $1, 2, 3, \dots, p - 1$.

In fact, no one knows how to find x given g , p , and $g^x \bmod p$ for values of p over a thousand bits long [2]. Note that there is no proof that finding x is difficult. It's just that no has found any feasible method.

This forms the basis of a so-called public key cryptography system. A public-key cryptography system uses pairs of keys, one called the private key and one called the public key. Here, Pete's chosen value of x is his private key. The value $y = g^x \bmod p$ is Pete's public key. Pete can freely distribute his public key, namely g , p and y , particularly y , which is unique to Pete, without anyone being able to derive his private key. Someone who knows Pete's public key can encrypt a message that only Pete, or someone who knows Pete's private key, can decrypt [5]. And Pete can use his private key to form a digital signature, which we'll cover soon. This scheme, using g , p , the private key x , and the public key $g^x \bmod p$ is the Diffie-Hellman public key system. There's another public key system, known by its founder's initials.

3 The RSA public key system

In 1977 Ron Rivest, Adi Shamir and Len Adleman published their RSA public key system in Martin Gardner's Mathematical Games column in Scientific American. It works like this. Pete Caithamer selects two thousand-bit prime numbers, p and q . He multiplies them together and publishes their product, $N = pq$, as the important part of his public key. He also needs to select an exponent, denoted e . This e has nothing to do with 2.71828. It's just an integer greater than 2 and kept fairly small, at least compared with these thousand-bit numbers. The exponent e is typically 3 or 65537. We'll use $e = 3$ here [7].

Because p and q are so large, and because Pete selected them at random, someone who has only N would not be able to find its prime factors p and q .

I should qualify this by saying that there is no proof that finding the factors of such an N is difficult. It's just that the best algorithms known, running on all the computers in the fastest network, would not be able to find this large a p and q from N . The largest product of two randomly selected primes that has been factored is 576 bits long.

But Pete knows p and q , and using them he can find d , which he uses as his private key. The number d is related to e by these properties.

For any m , form

$$c = m^e \bmod N .$$

That is, c is the remainder when m cubed (assuming e is 3) is divided by N .

Then m can be recovered from c by calculating

$$m = c^d \bmod N .$$

Finding such a d is as difficult as factoring N [9]. Furthermore, it is conjectured [10] that there is no significantly faster way to find m from c than by finding d .

We can take advantage of this to encrypt a message that only Pete (or someone who has Pete's private key) can decrypt. The usual way is to pick a conventional cryptographic key k at random (k would typically be 128 bits or 256 bits), encrypt the message with k , using a conventional cipher such as AES [12], send the ciphertext and $k^e \bmod N$ to Pete [11], and destroy all record of k .

This is just what happens when your web browser uses SSL to establish a secure connection to a web site, so you can transmit your credit card number without any of the numerous intermediate points through which the packets pass being able to decrypt it.

4 Digital Signature

A digital signature is a sequence of bits with certain properties that allow it to act like a person's signature on a document. There are several digital signature protocols. Each protocol specifies a signing method that produces a digital signature from something which is to be signed and a private key, and a verification method that puts out "pass" or "fail" from something that was purportedly signed, the purported digital signature itself, and the purported public key of the signer. Both

methods are efficient, but forging a signature, that is, coming up with a digital signature that will pass the verification check using a particular public key, without knowing the corresponding private key, is infeasible.

The “something to be signed” is usually called a document or a message, but it’s really just any sequence of bits. Most protocols use only a hash of the message to be signed, although a few allow the signing of short messages that are not hashed. One of the things needed to prevent forgery is that the hash function used, if any, be collision resistant. Otherwise a digital of one message would also verify as the digital signature of a different message with the same hash.

To form an RSA signature of some message m , Pete, who has published his public key (N and e), uses his private key d .

- 1.) He finds the hash of m , the message he wants to sign. $h = \text{SHA1}(m)$
- 2.) He pads h using a standard padding function such as $\text{OAEP}(\cdot)$ to get $t = \text{OAEP}(h)$, which is the same length as N .
- 3.) He raises t to the d power mod N to get the signature $s = t^d \bmod N$.

This is quite straightforward, except for the padding, which is necessary to prevent certain attacks that we need not be concerned with here.

Someone who wants to verify an s' which is purported to be Pete’s signature of the document m goes through similar steps in the reverse order.

- 1.) Get Pete’s public key (N and e).
- 2.) Raise s' to the e power to get t' , where $t' = (s')^e \bmod N$. This is the only step that depends on Pete’s public key.
- 3.) Undo the padding, which also requires a check that the padding is internally consistent, to get $h' = \text{OAEP}^{-1}(t')$. If the padding is not consistent, the verification fails.
- 4.) Check that the h' obtained from step 3) matches the $h = \text{SHA1}(m)$ which is the hash of the document purportedly signed. If they don’t match, the verification fails.

If we’re lucky we can get the certifier of voting software, or any software, to generate a public/private key pair, and to keep the private key private. Then, when the certifier writes a list of the certified software’s files and their hashes, it can include a digital signature of the list. The task of ensuring that the list of files and their hashes has not been modified since it was written can then be replaced by the two simpler tasks of

- 1.) verifying a digital signature on that list, and
- 2.) ensuring that we used the right public key (the certifier’s) for that verification.

5 Zero-knowledge proof

The idea of a zero-knowledge proof is that a prover, whom I'll call Perry, convinces a verifier, whom I'll call Vera, of the truth of something, without giving Vera any information other than that the proposition to be proved is true.

The first zero-knowledge proof that I saw was a proof that the prover knew a Hamiltonian circuit in a particular graph. We'll use something closer to home. We'll have Perry prove to Vera that he knows the private key corresponding to some particular RSA public key.

To begin, Vera and Perry both know some RSA public key consisting of a modulus N and an exponent e . Perry wants to prove to Vera that he knows d , the corresponding private key.[13]

One way of doing that would be for Vera to send Perry an integer i that she has picked, perhaps at random, perhaps not, from the range $1 < i < N$, and have Perry send back $j = i^d \bmod N$. Then Vera could check that $j^e \bmod N$ is i . To be really convinced, Vera should pick her i at random. After all, even if Perry doesn't know d , he can still find pairs of numbers (i, j) such that $i^d = j \bmod N$, by picking j and calculating the corresponding i . Perry could have been running his computer 24x7, accumulating gigabytes of such pairs. However, even if Perry has used all the computers on Earth, he can only have covered a negligible fraction of the range 2 through $N - 1$, because N is so large. So, if Vera picks i at random, the chance that Perry will have found such a j is negligible.

In any event, this way of doing it would give Vera extra information. Since Vera gets to pick i and Perry tells her what $i^d \bmod N$ is, Vera might use that value to form a digital signature that innocent third parties could verify as being Perry's signature. Or maybe it would allow Vera to decrypt a message meant only for Perry.

Vera and Perry must use a more elaborate procedure to get a zero-knowledge proof. Here it is.

- 1.) Vera sends Perry an integer i that she has picked by any means, from the range $1 < i < N$.
- 2.) Perry sends Vera an integer k which he has picked at random from the range $1 < k < N$.
- 3.) Vera sends Perry a bit b , which is either 0 or 1, that she has picked at random.
- 4.) If b is 0, Perry sends Vera y_0 , which is $y_0 = k^d \bmod N$. Vera verifies that $k = y_0^e \bmod N$. If not, Vera concludes that Perry does not know d .
- 5.) If b is 1, Perry sends Vera y_1 , which is $y_1 = i^d k^d \bmod N$. Vera verifies that $ki = y_1^e \bmod N$. If not, Vera concludes that Perry does not know d .

Now, if Perry knows both y_0 and y_1 , then he knows $y_1/y_0 \bmod N$, which is $i^d \bmod N$. But if Perry does not know d , he might still pass Vera's test.

A cheating Perry can pick a j , send his k value as $k = j^e \bmod N$, and he then can pass Vera's test if $b = 0$ by sending $y_0 = j$. But he fails if $b = 1$ because he doesn't know $y_1 = i^d j \bmod N$.

Or a cheating Perry can pick a j , send k as $k = j^e/i \bmod N$, and pass Vera's test if $b = 1$ by sending $y_1 = j$ (ki is indeed $y_1^e \bmod N$). But he fails if $b = 0$, because he doesn't know $y_0 = k^d \bmod N$, which is $j/i^d \bmod N$.

Vera and Perry repeat this process many times. The probability that Perry can go for n rounds without failing is at most $1/2^n$ if he doesn't know d . By the time the number of rounds reaches 80, Vera should be convinced that Perry knows d [14].

But can Vera use the numbers she's received from Perry to convince Skip, the skeptic, that Perry knows d ? Perry passed all of Vera's tests. Each round has numbers (i, k, b, y_b) , where $y_b^e = i^b k \bmod N$. Vera presents Skip with the complete transcript of all the numbers she and Perry have sent each other. And Skip doesn't believe it.

Skip says, "How do I know that you didn't make these numbers up yourself, Vera, without any help from Perry? All you would have had to do is select i, b and y_b at random, then calculate $k = y_b^e / i^b \bmod N$."

Vera has no answer for Skip. Although Vera is convinced that Perry knows d , the transcripts all by themselves prove nothing, and if she had done what Skip suggests she has done, the fake transcripts would be indistinguishable from real transcripts.

A zero-knowledge proof is important because it has exactly the right psychology for voting. We want each voter to be convinced that with very high probability, his or her vote has been included correctly in the reported totals, and yet be unable to prove to any third party how he or she voted.

6 Oblivious transfer

In an oblivious transfer, one party, whom we'll call Alice, has two messages, m_1 and m_2 for another party, whom we'll call Bob. Bob can choose which of m_1 or m_2 he wants to receive, but he can't get both messages, and Alice never finds out which message Bob receives.

How can oblivious transfer be implemented? It's straightforward using Diffie-Hellman public/private keys, using a value for g and a large prime p that are selected before election day. Alice gives Bob a randomly selected constraint C in the range $0 \leq C < p$ [16]. Bob makes up a fresh private key, x , at random and calculates the corresponding public key Y as $Y = g^x \bmod p$. Bob gives Alice two numbers, whose order is determined by which message Bob has decided he wants to receive. If he wants the first message, he gives Alice Y and $C/Y \bmod p$. If he wants the second message, he gives Alice $C/Y \bmod p$ and Y .

Alice confirms that the product of the two numbers from Bob equals $C \bmod p$. Alice encrypts the first message using the first number from Bob as if it were Bob's public key. She encrypts the second message using the second number from Bob as if it were Bob's public key. She gives Bob both ciphertexts. Bob can decrypt only the message that was encrypted using Y . He has no clue what the private key corresponding to public key C/Y is, so he can't decrypt the message that Alice encrypted using C/Y . Alice can't tell which message Bob receives, because she doesn't know which of the two numbers from Bob was his public key.

Here's one voting scheme where oblivious transfer would be useful. Bob is a voter, and he's allowed to cast either zero votes or one vote for some particular candidate. He does this by picking two numbers that either add up to zero or add up to one. The voting booth software is supposed to

- (a) Verify that Bob's numbers sum to either zero or one.

- (b) Encrypt Bob's numbers, separately, using the public key of a tabulating authority, such as the county's Election Commission, forming two ciphertexts. (The voting booth does not show the voter both ciphertexts.)
- (c) At the end of election day, dump out all the ciphertexts from step (b) for all such numbers, from Bob and everybody else that votes in the same booth, in a scrambled order.

The ciphertexts are made public, and when the tabulating authority has all the ciphertexts from all the voting booths, the tabulating authority publicly reveals its private key, decrypts the ciphertexts to recover the voters' numbers and adds them all to get the candidate's total. Using the now-revealed tabulating authority private key, anyone can take the published ciphertexts, decrypt them and check that the numbers add up to the reported total.

But, how does Bob know that the voting booth actually encrypted his numbers? Maybe the voting booth encrypted two numbers that add up to something other than what Bob's add up to.

To assuage Bob's suspicion, the voting booth can provide Bob with one of the two ciphertexts and enough information to show that it is indeed the encryption with the tabulating authority's public key of one of the numbers that Bob provided. The booth must not give Bob enough information to prove that both his numbers were correctly enciphered, because then Bob could prove to third parties how he voted.

Let's say Bob's two numbers are 5 and -4. The voting booth assembles two messages, m_1 and m_2 . The number m_1 contains the ciphertext for the 5 that the voting booth will send to the tabulating authority. The number m_2 contains the ciphertext for the -4.

Using oblivious transfer, Bob chooses to get either the ciphertext for 5 or the ciphertext for -4. He checks that the message he receives from the voting booth is indeed a public key encryption of 5 or -4, respectively. If the voting booth has cheated or erred, Bob's probability of catching it is at least $1/2$.

On election night, when the ciphertexts are published, Bob checks the county's web site to make sure that the ciphertext he has is listed. If not, he files some kind of complaint whose nature is outside the scope of this document [15]. To preclude such complaints, the voting booth should send both of Bob's ciphertexts to the county, as it does not know which one Bob will check.

7 Schneier's password

Bruce Schneier is a well-known cryptographer. He proposes that voters be able to include a password of their own choosing on their ballots. Then, all ballots are posted publicly along with the corresponding passwords. A voter can observe that a ballot with his or her password containing his or her votes was posted, but does not have a way of proving to a third party that that is indeed his or her password.

This proposal need some polishing. For instance, how many voters beside Pete Caithamer would start their passwords with the letters "PMC"?

8 Visual Cryptography

Most of these tools require that some calculations be performed on behalf of the voter. There's a problem here, in that the voters using these schemes have to trust some computer to perform honestly and reliably. I regard it as an open question as to how a voter can develop trust in a computer.

One significant development is the application of visual cryptography. Visual cryptography allows someone to perform a significant cryptographic operation, namely decryption, without any help from a computer.

9 Mixes

Proposals to use mixes in voting have come up in the last few years. The basic idea is that the voting booth encrypts the voter's ballot using public keys from one or more tabulating authorities and gives the voter the ciphertext. On election night a list of all the ciphertexts is made public. Each voter can check that the ciphertext he or she received from the voting booth is in the list.

Then the tabulating authorities shuffle and decrypt the ciphertexts, recovering the original ballots as they existed in the voting booth. Because of the shuffling, the ballots, unlike the ciphertexts, cannot be associated with particular voters.

This system expects that there will be more than one tabulating authority, and it maintains voter anonymity as long as at least one of the tabulating authorities performs its shuffle at random and keeps its private key private.

To use this system, the voting booth needs to be configured with the public keys of all the tabulating authorities. The voting booth encrypts each ballot four times using the public key of the last tabulating authority. It then encrypts that ciphertext four times with the public key of the next-to-last tabulating authority. And so on, until at last it encrypts four times with the public key of the first tabulating authority. It is this final ciphertext that the voting booth dumps out when the polls close.[17]

Recovering the ballot requires the cooperation of all the tabulating authorities. Each of them, starting with the first, follows the same procedure.

The tabulating authority receives a set of ciphertexts that have been encrypted four times with its own public key. The first tabulating authority gets them from the voting booths. The other tabulating authorities get theirs as the (partially) decrypted output from the preceding tabulating authority. Call this list of ciphertexts list A.

The tabulating authority decrypts each ciphertext on list A once, using its private key. The resulting partially decrypted ciphertexts form list B. The tabulating authority scrambles the order of the ciphertexts in list B.

The tabulating authority then decrypts each ciphertext in list B, using its private key, to get list C. It randomly scrambles the order of the ciphertexts in list C.

It repeats this procedure on list C to get list D, and on list D to get list E. List E now contains ciphertexts that this tabulating authority can no longer decrypt, since the voting booth encrypted using this tabulating authority's public key only four times. The tabulating authority publishes all five lists. List E becomes the input to the next tabulating authority (or, if this is the last tabulating authority, list E is the recovered ballots).

After a tabulating authority has published its five lists, its work is audited by random partial checking (RPC) [18]. We want to verify that the tabulating authority's output is really a decryption of its input, to deter the tabulating authority from putting out fake ballots that it has encrypted using the public keys of the following tabulating authorities.

The list B ciphertexts are divided into two equal-sized batches at random[19]. For list B ciphertexts that get assigned to the first batch, the tabulating authority must identify the list A ciphertext that it came from and demonstrate that it is a proper decryption. For list B ciphertexts in the second batch, the tabulating authority must identify the list C ciphertext that it went to and demonstrate that the list C ciphertext is a proper decryption.

The tabulating authority then demonstrate the list C to list D correspondence for half (randomly selected) of the list C ciphertexts which now have their list B source revealed and half of the list C ciphertexts which did not have their list B source revealed.

Finally the tabulating authority must show the list E destination of the half of the list D ciphertexts whose list C provenance was not shown.

If a tabulating authority cheats at any stage, its probability of being caught is $1/2$ for each abused ballot.

[1] A one-thousand bit integer g is in the range $2^{1000-1} \leq g \leq 2^{1000} - 1$. The length, in bits, of g^x is between $1000x - x + 1$ and $1000x$ bits for $x \geq 1$.

[2] There are a few restrictions on g and p . The number g should be chosen so that $g^x \not\equiv 1 \pmod{p}$ for $1 \leq x \leq 2^{159}$. See the National Institute of Standards and Technology's Digital Signature Standard at <http://csrc.nist.gov/CryptoToolkit/tkdigsigs.html> Appendices 2 and 4 for more details.

[3] See <http://csrc.nist.gov/publications/fips/fips180-2/fips180-2withchangenotice.pdf> at <http://csrc.nist.gov/CryptoToolkit/tkhash.html> for the official definition of SHA-1.

[4] This is in itself not proof that SHA-1 is collision resistant. SHA-0, as the predecessor to SHA-1 is now known, is a year older than SHA-1, and until recently no collisions had ever been found for it either. In August, 2004, Weng, Fang, Lai, and Yu published the first collisions in the hash functions SHA-0, MD5, RIPEMD and HAVAL, and extended attacks on MD4 to the point where collisions can be found with pencil and paper. SHA-1 is apparently not susceptible to the techniques they used.

[5] Here is one way to use Pete's public key to encrypt a message to Pete that only Pete (or someone who knows Pete's private key) can decrypt:

- 1.) Pick a random r in a suitable range (namely, the same range that Pete picked his private key x from).
- 2.) Calculate $s = y^r \pmod{p}$, where y and p are from Pete's public key.
- 3.) Use the calculated value of s as a secret key to encrypt the message using a conventional (non-public key) encryption algorithm such as AES (Conventional encryption is outside the scope of this document).
- 4.) Send $R = g^r \pmod{p}$ and the ciphertext from step 3 to Pete.
- 5.) Pete uses his private key x to form $s = R^x \pmod{p}$. This is the same value of s as we found in step 2. Proof: The y in step 2 is $g^x \pmod{p}$, so the s calculated in step 2 is $(g^x)^r \pmod{p}$, which is $g^{xr} \pmod{p}$, which is $(g^r)^x \pmod{p}$, which is $R^x \pmod{p}$.
- 6.) Pete uses the calculated value of s from step 5 to decrypt the ciphertext back to the original message.

[7] The number e must be relatively prime to p and q . That is, $\gcd(e, p) = 1$ and $\gcd(e, q) = 1$.

[8] The number d is the inverse of $e \pmod{\text{lcm}(p-1, q-1)}$. Any d will work as long as de leaves a remainder of 1 when divided by the least common multiple of $p-1$ and $q-1$. There is a persistent myth that d must be the inverse of $e \pmod{\phi(N)}$, where $\phi(N)$ is just $(p-1)(q-1)$ when N is product of two primes p and q . The numbers $p-1$ and $q-1$ share at least a factor of 2, so $\text{lcm}(p-1, q-1)$ is strictly less than $(p-1)(q-1)$.

The proof depends on Fermat's theorem $a^p = a \pmod{p}$ for any a and any prime p .

[9] The number N can be factored fairly efficiently given e and d . The method involves the Jacobi symbol, which I don't understand.

[10] This is the so-called "RSA conjecture".

[11] It's almost that simple. Instead of $k^e \bmod N$, we actually form $\text{pad}(k)^e \bmod N$, where $\text{pad}(\cdot)$ is a padding function such as *OAEP* (<http://www.cs.ucdavis.edu/~rogaway/papers/oaep-abstract.html>) or *SAEP* (<http://crypto.stanford.edu/~dabo/abstracts/saep.html>).

[12] See <http://csrc.nist.gov/CryptoToolkit/tkencryption.html>.

[13] Strictly speaking, since Perry does not prove the RSA hypothesis to Vera, Perry only establishes that he can extract e -th roots mod N . But that's good enough.

[14] Fewer repetitions are needed if Vera increases b 's range from $0 \leq b \leq 1$ to something like $0 \leq b \leq 2^{80}$ or more. Perry should respond with $y_b = i^{db} k^d \bmod N$. Vera verifies that $ki^b = y_b^e \bmod N$. If Perry does not know d , he can still choose k so that he can pass Vera's verification for one value of b , but the probability of Vera choosing that value for b is negligible.

[15] There are some details that need to be filled in to make this work. To make his complaint stick, Bob needs to prove the ciphertext he says is not on the county's web site came from the voting booth. To allow such proof, the voting booth digitally signs each obviously transferred message, using the voting booth's own private key. All the voting booths' public keys are published before the election starts. Bob verifies the booth's signature on the message he gets before leaving the voting booth.

There also needs to be some kind of check that two voters who pick the same number are not both given the same ciphertext.

It may be better to split the vote up into more than two parts. Generalizations of oblivious transfer allow Bob to receive, say, any six of ten messages that Alice has for him. Instead of just one constraint, Alice requires that the numbers she gets from Bob, some of which are public keys for which Bob knows the private keys, meet w different independent constraints, where the number of messages that Bob is to receive is the total number of messages Alice provides minus w .

In particular, if Bob splits his vote into only two parts, and his numbers are 5 and -4, there has to be a 4 and a -5 among the other voters' numbers, so that Bob can plausibly be believed to have voted either way, no matter which of his numbers he gets the ciphertext for. Splitting the votes into more parts may ameliorate this problem.

It may help to evaluate the sum modulo $(n + 1)$, where n is the number of registered voters in the precinct.

[16] Alice must choose C to have the same order as g (and Y) in the multiplicative group mod p . Bob must check that $C^q = 1 \bmod p$, where q is the (prime) order of g . Otherwise Alice could distinguish Y (which has order q) from C/Y if C/Y did not have order q .

[17] The voting booth keeps the ciphertext until the polls close only to preclude premature decryption. There's no need for the voting booth to dissociate the final ciphertext of a ballot from the voter who cast it.

[18] <http://www.rsasecurity.com/rsalabs/node.asp?id=2062>

[19] The tabulating authorities can cooperate to generate the required random numbers. For instance, each can make up a 160-bit partial seed for a random number generator, publish the SHA-1 hash of the partial seed, and, when all tabulating authorities have published the hashes,

reveal the partial seed. The random number generator is then seeded with the sum mod 2^{160} of the partial seeds. If any tabulating authority's partial seed is selected randomly, the resulting sum is random.

For copies contact Peter Caithamer: caithamer@usi.edu