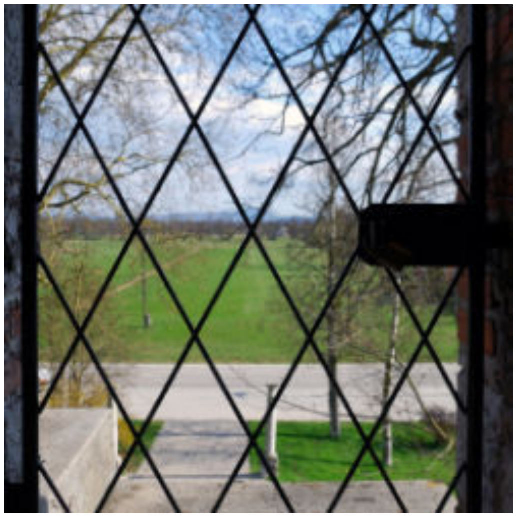
COMMUNICATIONS OF THE ACM TRUSTED INSIGHTS FOR COMPUTING'S LEADING PROFESSIONALS

RESEARCH HIGHLIGHTS

Computing Arbitrary Functions of Encrypted Data

Craig Gentry

Communications of the ACM
Vol. 53 No. 3, Pages 97-105
10.1145/1666420.1666444


Credit: iStockPhoto.com

Suppose that you want to delegate the ability to *process* your data, without giving away *access* to it. We show that this separation is possible: we describe a "fully homomorphic" encryption scheme that keeps data private, but that allows a worker that *does not have the secret decryption key* to compute any (still encrypted) result of the data, even when the function of the data is very complex. In short, a third party can perform complicated processing of data without being able to see it. Among other things, this helps make cloud computing compatible with privacy.

1. Introduction

Is it possible to delegate *processing* of your data without giving away *access* to it?

This question, which tests the tension between convenience and privacy, has always been important, but seems especially so now that we are headed toward widespread use of cloud computing. To put everything online "in the cloud," unencrypted, is to risk an Orwellian future. For certain types of data, such as medical records, storing them off-site unencrypted may be illegal. On the other hand, encrypting one's data seems to nullify the benefits of cloud computing. Unless I give the cloud my secret decryption key (sacrificing my privacy), what can I expect the cloud to do with my encrypted data except send it back to me, so that I can decrypt it and process it myself?

Fortunately, this is a false dilemma, or at least convenience and privacy can be reconciled to a large extent. For data that is encrypted with an "ordinary" encryption scheme, it is virtually impossible for someone without the secret decryption key (such as the cloud) to manipulate the underlying data in any useful way. However, some encryption schemes are *homomorphic* or *malleable*. They let anyone manipulate (in a meaningful way) what is encrypted, even without knowing the secret key!

In this paper, we describe the first *fully homomorphic* encryption (FHE) scheme, where "fully" means that there are no limitations on what manipulations can be performed. Given ciphertexts $c_1, ..., c_t$ that encrypt $m_1, ..., m_t$ with our scheme under some key, and given any efficiently computable function f, anyone can efficiently compute a ciphertext (or set of ciphertexts) that encrypts $f(m_1, ..., m_t)$ under that key. In short, this permits general computations on encrypted data. No information about $m_1, ..., m_t$ or the value of $f(m_1, ..., m_t)$ is leaked.

This means that cloud computing is consistent with privacy. If I want the cloud to compute for me some function f of my (encrypted) data $m_1, ..., m_t$—for example, this function could be "all files containing 'CACM' or 'Communications' within three words of 'ACM' "—I send a description of f to the cloud, which uses the scheme's malleability to compute an encryption of $f(m_1, ..., m_t)$, which I decrypt. The cloud never sees any unencrypted data. If I want, I can even use the scheme to encrypt a description of f, so that the cloud does not even see what I am searching for.

Rivest, Adleman, and Dertouzos[5] suggested that fully homomorphic encryption may be possible in 1978, shortly after the invention of the RSA cryptosystem,[6] but were unable to find a secure scheme. As an application, they described our private cloud computing scenario above, though of course they used different terminology. There are many other applications. Homomorphic encryption is useful whenever it is acceptable if a response (e.g., to a search engine query) is encrypted.

Below, we begin by describing homomorphic encryption in more detail. Then, we describe a concrete scheme due to van Dijk, Gentry, Halevi, and Vaikuntanathan,[9] which uses only simple integer operations, and is a conceptually simpler version of the first scheme by Gentry,[2, 3] which uses lattices. Toward the end, we discuss the scheme's (rather slow) performance. Throughout, we try to make the ideas more tangible by constantly returning to a physical analogy: a jewelry store owner, Alice, who wants her workers to *process* raw precious materials into intricately designed rings and necklaces, but who is afraid to give her workers complete *access* to the materials for fear of theft.

2. Homomorphic Encryption

2.1. Alice's jewelry store

At first, the notion of processing data without having access to it may seem paradoxical, even logically impossible. To convince you that there is no fallacy, and to give you some intuition about the solution, let us consider an analogous problem in (a fictional version of) the "physical world."

Alice owns a jewelry store. She has raw precious materials—gold, diamonds, silver, etc.—that she wants her workers to assemble into intricately designed rings and necklaces. But she distrusts her workers and assumes that they will steal her jewels if given the opportunity. In other words, she wants her workers to *process* the materials into finished pieces, without giving them *access* to the materials. What does she do?

Here is her plan. She uses a transparent impenetrable glovebox, secured by a lock for which only she has the key. She puts the raw precious materials inside the box, locks it, and gives it to a worker. Using the gloves, the worker assembles the ring or necklace inside the box. Since the box is impenetrable, the worker cannot get to the precious materials, and figures he might as well return the box to Alice, with the finished piece inside. Alice unlocks the box with her key and extracts the ring or necklace. In short, the worker processes the raw materials into a finished piece, without having true access to the materials.

The locked impenetrable box, with raw precious materials inside, represents an encryption of the initial data $m_1, ..., m_t$, which can be accessed only with the secret decryption key. The gloves represent the homomorphism or malleability of the encryption scheme, which allows the raw data to be manipulated while it is inside the "encryption box." The completed ring or necklace inside the box represents the encryption of $f(m_1, ..., m_t)$, the desired function of the initial data. Note that "lack of access" is represented by lack of physical access, as opposed to lack of visual access, to the jewels. (For an analogy that uses lack of visual access, consider a photograph developer's darkroom.)

Of course, Alice's jewelry store is only an analogy. It does not represent some aspects of homomorphic encryption well, and taking it too literally may be more confusing than helpful. We discuss some flaws in the analogy at the end of this section, after we describe homomorphic encryption more formally. Despite its flaws, we return to the analogy throughout, since it motivates good questions, and represents some aspects of our solution quite well—most notably, "bootstrapping," which we discuss in Section 4.

2.2. Homomorphic encryption: functionality

An encryption scheme ε has three algorithms: $\text{KeyGen}_\varepsilon$, $\text{Encrypt}_\varepsilon$, and $\text{Decrypt}_\varepsilon$, all of which must be *efficient*—that is, run in time poly(λ), polynomial in a security parameter λ that specifies the bit-length of the keys. In a *symmetric*, or *secret key*, encryption scheme, $\text{KeyGen}_\varepsilon$ uses λ to generate a single key that is used in both $\text{Encrypt}_\varepsilon$ and $\text{Decrypt}_\varepsilon$, first to map a message to a ciphertext, and then to map the ciphertext back to the message. In an *asymmetric*, or *public key*, encryption scheme, $\text{KeyGen}_\varepsilon$ uses λ to generate two keys—a public encryption key pk, which may be made available to everyone, and a secret decryption key sk. As a physical analogy for an asymmetric encryption scheme, one can think of Alice's public key as a padlock, which she constructs and distributes, that can be locked without a key. Anyone can put a message inside a box secured by Alice's padlock (encrypt), and mail it via a public channel to Alice, but only Alice has the key needed to unlock it (decrypt).

A homomorphic encryption scheme can be either symmetric or asymmetric, but we will focus on the asymmetric case. It has a fourth algorithm $\text{Evaluate}_\varepsilon$, which is associated to a set $\mathcal{F}_\varepsilon$ of *permitted functions*. For any function f in $\mathcal{F}_\varepsilon$ and any ciphertexts $c_1, \ldots, c_t$ with $c_i \leftarrow \text{Encrypt}_\varepsilon$ (pk, m_i), the algorithm $\text{Evaluate}_\varepsilon$ (pk, f, $c_1, \ldots, c_t$) outputs a ciphertext c that encrypts $f(m_1, \ldots, m_t)$—i.e., such that $\text{Decrypt}_\varepsilon$ (sk, c) = $f(m_1, \ldots, m_t)$. (For convenience, we will assume that f has one output. If f has k outputs, then $\text{Evaluate}_\varepsilon$ outputs k ciphertexts that encrypt $f(m_1, \ldots, m_t)$ collectively.) As shorthand, we say that ε can *handle* functions in $\mathcal{F}_\varepsilon$. For a function f not in $\mathcal{F}_\varepsilon$, there is no guarantee that $\text{Evaluate}_\varepsilon$ will output anything meaningful. Typically $\text{Evaluate}_\varepsilon$ is undefined for such a function.

As described thus far, it is trivial to construct an encryption scheme that can handle all functions. Just define $\text{Evaluate}_\varepsilon$ as follows: simply output $c \leftarrow (f, c_1, \ldots, c_t)$, without "processing" the ciphertexts at all. Modify $\text{Decrypt}_\varepsilon$ slightly: to decrypt c, decrypt $c_1, \ldots, c_t$ to obtain $m_1, \ldots, m_t$, and then apply f to these messages.

But this trivial solution obviously does not conform to the spirit of what we are trying to achieve—to *delegate* the data processing (while maintaining privacy). The trivial solution is as if, in Alice's jewelry store, the worker simply sends the box (which need not have gloves) back to Alice without doing any work on the raw precious materials, and Alice unlocks the box, extracts the materials, and assembles the ring or necklace herself.

So, how do we formalize what it means to *delegate?* Intuitively, the purpose of delegation is to reduce one's workload. We can formalize this in terms of the running times (i.e., complexity) of the algorithms. Specifically, we require that decrypting c (the ciphertext output by Evaluate$_\varepsilon$) takes the *same amount of computation* as decrypting c_1 (a ciphertext output by Encrypt$_\varepsilon$). Moreover, we require that c is the same size as c_1. We refer to these as the *compact ciphertexts* requirement. Again, the size of c and the time needed to decrypt it do not grow with the complexity of f; rather, they are *completely independent* of f (unless f has multiple outputs). Also, of course, the complexity of Decrypt$_\varepsilon$, as well as the complexity of KeyGen$_\varepsilon$ and Encrypt$_\varepsilon$, must remain polynomial in λ.

ε is *fully homomorphic* if it can handle all functions, has compact ciphertexts, and Evaluate$_\varepsilon$ is efficient in a way that we specify below. The trivial solution above certainly is not fully homomorphic, since the size of the ciphertext output by Evaluate$_\varepsilon$, as well as the time needed to decrypt it, depend on the function being evaluated. In terms of Alice's jewelry store, our definition of fully homomorphic captures the best-case scenario for Alice: her workers can assemble arbitrarily complicated pieces inside the box, but the work needed to assemble has no bearing on the work Alice needs to do to unlock the box and extract the piece.

We want our fully homomorphic scheme to be efficient for the worker, as well. In particular, we want the complexity of Evaluate$_\varepsilon$—like the other algorithms of ε—to depend only polynomially on the security parameter. But clearly its complexity must also depend on the function being evaluated. How do we measure the complexity of f? Perhaps the most obvious measure is the running time T_f of a Turing machine that computes f. We use a related measure, the size S_f of a *boolean circuit* (i.e., the number of AND, OR, and NOT gates) that computes f. Any function that can be computed in T_f steps on a Turing machine can be expressed as a circuit with about T_f gates. More precisely, $S_f < \mathrm{k} \cdot T_f \cdot \log T_f$ for some small constant k. Overall, we say that Evaluate$_\varepsilon$ is efficient if there is a polynomial g such that, for any function f that is represented by a circuit of size S_f, Evaluate$_\varepsilon$ (pk, f, c_1, ..., c_t) has complexity at most $S_f \cdot g(\lambda)$.

The circuit representation of *f* is also useful because it breaks the computation of *f* down into simple steps—e.g., AND, OR, and NOT gates. Moreover, to evaluate these gates, it is enough to be able to add, subtract, and multiply. (In fact, it is enough if we can add, subtract and multiply *modulo* 2.) In particular, for $x, y \in \{0, 1\}$, we have AND$(x, y) = xy$, OR$(x, y) = 1 - (1 - x)(1 - y)$ and NOT$(x) = 1 - x$. So, to obtain a fully homomorphic encryption scheme, all we need is a scheme that operates on ciphertexts so as to add, subtract, and multiply the underlying messages, indefinitely.

But is the circuit representation of *f*—or some arithmetized version of it in terms of addition, subtraction, and multiplication—necessarily the *most efficient* way to evaluate *f*? In fact, some functions, like binary search, take much longer on a Turing machine or circuit than on a random access machine. On a random access machine, a binary search algorithm on t ordered items only needs to "touch" $O(\log t)$ of its inputs.

A moment's thought shows that random-access speed-ups cannot work if the data is encrypted. Unless we know something a priori about the relationship between *f* and $m_1, \ldots, m_t$, the algorithm Evaluate$_\varepsilon$(pk, *f*, $c_1, \ldots, c_t$) must touch all of the input ciphertexts, and therefore have complexity at least linear in the number of inputs. To put it another way, if Evaluate$_\varepsilon$ (for some reason) did not touch the second half of the ciphertexts, this would leak information about the second half of the underlying messages—namely, their irrelevance in the computation of *f*—and this leakage would contradict the security of the encryption scheme. While Evaluate$_\varepsilon$ must have running time at least linear in t as an unavoidable cost of the complete privacy that homomorphic encryption provides, a trade-off is possible. If I am willing to reveal—e.g., in the cloud computing context—that the files that I want are contained in a certain 1% of my data, then I may help the cloud reduce its work by a factor of 100.

Another artifact of using a fixed circuit representation of *f* is that the size of the output—i.e., the number of output wires in the circuit—must be fixed in advance. For example, when I request all of my files that contain a combination of keywords, I should also specify how much data I want retrieved—e.g., 1MB. From my request, the cloud will generate a circuit for a function that outputs the first megabyte of the correct files, where that output is truncated (if too much of my data satisfies my request), or padded with zeros (if too little). A moment's thought shows that this is also unavoidable. There is no way the cloud can avoid truncating or padding unless it knows something a priori about the relationship between the function and my data.

◆ 2.3. Homomorphic encryption: security

In terms of security, the weakest requirement for an encryption scheme is *one-wayness*: given the public key pk and a ciphertext c that encrypts unknown message m under pk, it should be "hard" to output m. "Hard" means that any algorithm or "adversary" A that runs in poly(λ) time has a negligible probability of success over the choices of pk and m (i.e., the probability it outputs m is less than $1/\lambda^k$ for any constant k).

Nowadays, we typically require an encryption scheme to have a stronger security property, called *semantic security* against chosen-plaintext attacks (CPA)[4]: given a ciphertext c that encrypts either m_0 or m_1, it is hard for an adversary to decide which, even if it is allowed to choose m_0 and m_1. Here, "hard" means that if the adversary A runs in polynomial time and guesses correctly with probability $1/2 + \epsilon$, then ϵ, called *A's advantage*, must be negligible. If this advantage is nonnegligible, then we say (informally) that the adversary *breaks* the semantic security of the encryption scheme.

If an encryption scheme is *deterministic*—i.e., if there is only one ciphertext that encrypts a given message—then it cannot be semantically secure. An attacker can easily tell whether c encrypts m_0, by running $c_0 \leftarrow$ Encrypt(pk, m_0) and seeing if c and c_0 are the same. A semantically secure encryption scheme must be *probabilistic*—i.e., there must be many ciphertexts that encrypt a given message, and Encrypt$_\varepsilon$ must choose one randomly according to some distribution.

One can *prove* the (conditional) one-wayness or semantic security of an encryption scheme by *reducing* a hard problem to breaking the encryption scheme. For example, suppose one shows that if there is an efficient algorithm that breaks the encryption scheme, then this algorithm can be used as a subroutine in an efficient algorithm that factors large numbers. Then, under the assumption that factoring is hard—i.e., that no poly(λ)-time algorithm can factor λ-bit numbers—the reduction implies that the encryption scheme must be hard to break.

Semantic security of a homomorphic encryption scheme is defined in the same way as for an ordinary encryption scheme, without reference to the Evaluate$_\varepsilon$ algorithm. If we manage to prove a reduction—i.e., that an attacker that breaks ε can be used to solve a hard problem like factoring—then this reduction holds whether or not ε has an Evaluate$_\varepsilon$ algorithm that works for a large set of functions.

To understand the power of semantic security, let us reconsider our cloud computing application. Sometime after storing her encrypted files in the cloud, Alice wants the cloud to retrieve the files that have a certain combination of keywords. Suppose that in its response, the cloud sends ciphertexts that encrypt the first three files. Can't the cloud just *see* that the first three encrypted files that it is storing for Alice happen to encrypt the same content as the three files that it sends to Alice? Not if the scheme is semantically secure. Even though some of the stored ciphertexts encrypt the same content as the sent ciphertexts, the cloud cannot *see* this, because semantic security guarantees that it is hard to tell whether two ciphertexts encrypt the same content.

Intuitively, it seems like the $\text{Evaluate}_{\varepsilon}$ algorithm should make ε easier to break, simply because this additional algorithm gives the attacker more power. Or, to put it in terms of the physical analogy, one would think that the easiest way to get inside the glovebox is to cut through the gloves, and that, the more flexible the gloves are, the easier the glovebox is to compromise; this suggests that, the more malleable the encryption scheme is, the easier it is to break. There is some truth to this intuition. Researchers[1, 8] showed that if ε is a *deterministic* fully homomorphic encryption scheme (or, more broadly, one for which it is easy to tell whether two ciphertexts encrypt the same thing), then ε can be broken in subexponential time, and in only polynomial time (i.e., efficiently) on a quantum computer. So, malleability seems to weaken the security of deterministic schemes. But these results do not apply to semantically secure schemes, such as ours.

2.4. Some flaws in the physical analogy

The physical analogy represents some aspects of homomorphic encryption poorly. For example, the physical analogy suggests that messages that are encrypted separately are in different "encryption boxes" and cannot interact. Of course, this interaction is precisely the purpose of homomorphic encryption. To fix the analogy, one may imagine that the gloveboxes have a one-way insertion slot like the mail bins used by the post office. Then, messages can be added to the same encryption box as they arrive. (Even this fix is not entirely satisfactory.)

Another flaw is that the output $f(m_1, ..., m_t)$ may have significantly fewer bits than $m_1, ..., m_t$, whereas in the analogy (absent significant nuclear activity inside the glovebox) the conservation of mass dictates that the box will have at least as much material inside when the worker is done as when he started. Finally, in Alice's jewelry store, even though a worker cannot extract the materials from a locked glovebox, he can easily tell whether or not a box contains a certain set of materials—i.e., the gloveboxes do not provide "semantic security."

3. A Somewhat Homomorphic Encryption Scheme

On our way to fully homomorphic encryption, we begin by constructing a *somewhat homomorphic* encryption scheme ε that can handle a limited class $\mathcal{F}_\varepsilon$ of permitted functions. Evaluate$_\varepsilon$ (pk, f, c_1, ..., c_t) does not work for functions f that are too complicated. Later, we will show to use ε to obtain fully homomorphic encryption.

3.1. Meanwhile in Alice's jewelry store

After figuring out how to use locked gloveboxes to get her workers to process her precious materials into fancy rings and necklaces, Alice puts in an order with Acme Glovebox Company. Unfortunately, the gloveboxes she receives are defective. After a worker uses the gloves for 1 min, the gloves stiffen and become unusable. But some of the fanciest pieces take up to an hour to assemble. Alice sues Acme, but meanwhile she wonders: Is there some way I can use these defective boxes to get the workers to securely assemble even the most complicated pieces?

She notices that the boxes, while defective, do have a property that might be useful. As expected, they have a oneway insertion slot, like post office mail bins. But they are also flexible enough so that it is possible to put one box inside another through the slot. She wonders whether this property might play a role in the solution to her problem, etc.

3.2. Our somewhat homomorphic scheme

Our somewhat homomorphic encryption scheme ε, described below, is remarkably simple.[9] We describe it first as a symmetric encryption scheme. As an example parameter setting, for security parameter λ, set $N = \lambda$, $P = \lambda^2$ and $Q = \lambda^5$.

An Encryption Scheme:

- KeyGen$_\varepsilon$ (λ): The key is a random P-bit odd integer p.
- Encrypt$_\varepsilon$(p, m): To encrypt a bit $m \in \{0, 1\}$, set m' to be a random N-bit number such that $m' = m \bmod 2$. Output the ciphertext $c \leftarrow m' + pq$, where q is a random Q-bit number.
- Decrypt$_\varepsilon$(p, c): Output ($c \bmod p$) mod 2, where ($c \bmod p$) is the integer c' in $(-p/2, p/2)$ such that p divides $c - c'$.

Ciphertexts from ε are *near-multiples* of p. We call (c mod p) the *noise* associated to the ciphertext c. It is the distance to the nearest multiple of p. Decryption works because the noise is m', which has the same parity as the message. We call a ciphertext output by Encrypt a *fresh* ciphertext, since it has small (N-bit) noise.

How is the scheme homomorphic? By simply adding, subtracting, or multiplying the ciphertexts as integers, we can add, subtract, or multiply (modulo 2) the underlying messages. However, complications arise, because these operations increase the noise associated to resulting ciphertexts. Eventually, the noise become so large that decryption no longer reliably returns the correct result.

Homomorphic Operations:

- $\text{Add}_\varepsilon(c_1, c_2)$, $\text{Sub}_\varepsilon(c_1, c_2)$, $\text{Mult}_\varepsilon(c_1, c_2)$: the output ciphertext c is $c_1 + c_2$, $c_1 - c_2$, or $c_1 \cdot c_2$.
- $\text{Evaluate}_\varepsilon(f, c_1, ..., c_t)$: Express the boolean function f as a circuit C with XOR and AND gates. Let $C^\dagger$ be the same circuit as C, but with XOR and AND gates replaced by addition and multiplication gates over the integers. Let $f^\dagger$ be the multivariate polynomial that corresponds to $C^\dagger$. Output $c \leftarrow f^\dagger(c_1, ..., c_t)$.

Let us check that ciphertexts output by $\text{Evaluate}_\varepsilon$ decrypt correctly. As a warm-up, let us consider Mult_ε. Let $c = c_1 \cdot c_2$, where c_i's noise is m'_i, which has the same parity as the message m_i. We have that

$$c = m'_1 \cdot m'_2 + pq'$$

for some integer q'. As long as the noises are small enough so that $| m'_1 \cdot m'_2 | < p/2$, we have that

$$(c \bmod p) = m'_1 \cdot m'_2$$

and therefore (c mod p) mod 2 = $m_1 \cdot m_2$, as it should be. We will consider the evaluation of more complicated functions momentarily, in Section 3.3.

So far we only described a symmetric homomorphic encryption scheme. Turning it into a public-key scheme is easy, but adds some complexity. As before, the secret key is p. The public key consists of a list of integers that are essentially "encryptions of zero." The list has length polynomial in λ. To encrypt a bit m, the ciphertext c is (essentially) m plus a random subset sum of the ciphertexts in the public key. If these ciphertexts have very small noise, the resulting ciphertext will also have small noise, and decryption will work properly: (c mod p) mod 2 will equal m, as before.

3.3. How homomorphic is it?

What is the set of permitted functions that our homomorphic encryption scheme ε can handle?

To answer this question, we need to analyze how the noise grows as we add and multiply ciphertexts. Encrypt$_\varepsilon$ outputs a *fresh* ciphertext with a small noise, at most N bits. As we Add$_\varepsilon$, Sub$_\varepsilon$, or Mult$_\varepsilon$ ciphertexts, the output ciphertext becomes more noisy. Multiplication tends to increase the noise faster than addition or subtraction. In particular, for ciphertexts c_1 and c_2 with k_1- and k_2-bit noises, the ciphertext $c \leftarrow c_1 \cdot c_2$ has (roughly) $(k_1 + k_2)$-bit noise.

What happens when we perform *many* Add$_\varepsilon$, Sub$_\varepsilon$, and Mult$_\varepsilon$ operations, as prescribed by the circuit representing a function f? Similar to what we saw above with multiplication, we have

$$f^{\dagger}(c_1, \ldots, c_t) = f^{\dagger}(m'_1, \ldots, m'_t) + pq'$$

for some integer q', where m'_t is the noise associated to c_i. If $|f^{\ddagger}(m'_1, \ldots, m'_t)| < p/2$, then $(f^{\ddagger}(c_1, \ldots, c_t)$ mod $p)$ equals $f^{\ddagger}(m'_1, \ldots, m'_t)$. And if we reduce this result modulo 2, we obtain the correct result: $f(m_1, \ldots, m_t)$.

In short, the functions that ε can handle are those for which $|f^{\ddagger}(a_1, \ldots, a_t)|$ is *always* less than $p/2$ if all of the a_i are at most N bits.

ε is already quite powerful. As an example, it can handle an elementary symmetric polynomial of degree d in t variables, as long as $2^{Nd} \cdot \binom{t}{d} < p/2$, which is true (roughly) when $d < P/(N \cdot \log t)$. For our suggested parameters, this degree can be quite large: $\lambda/(\log t) = \Omega(\lambda/\log \lambda)$. That ε can evaluate polynomials of such high degree makes it "homomorphic enough" for many applications. For example, it works well when f is a highly parallelizable function—e.g., a basic keyword search—in which case f has fairly low degree.

3.4. Semantic security and approximate GCDs

Euclid showed that, given two integers x_1 and x_2, it is easy to compute their greatest common divisor (gcd). But suppose that $x_1 = s_1 + p \cdot q_1$ and $x_2 = s_2 + p \cdot q_2$ are *near*-multiples of p, with s_1 and s_2 much smaller than p. When p is only an approximate gcd, is it still possible to compute p efficiently—i.e., in time polynomial in the bit-lengths of x_1 and x_2? Not in general, as far as we know.

In fact, if we sample s_i, p and q_i with λ, λ^2, and λ^5 bits (similar to our scheme ε), then the *approximate gcd problem* seems to remain hard even if we are given arbitrarily many samples $x_i = s_i + p \cdot q_i$, rather than just two. For these parameters, known attacks—including those using continued fractions and simultaneous diophantine approximation—take time essentially exponential in λ.

Moreover, we can reduce the approximate gcd problem to the security of our somewhat homomorphic encryption scheme. That is, we can prove that an attacker cannot efficiently break the semantic security of our encryption scheme unless the approximate gcd problem is easy.

4. Bootstrappable Encryption

4.1. Alice's eureka moment

One night, Alice dreams of immense riches, caverns piled high with silver, gold, and diamonds. Then, a giant dragon devours the riches and begins to eat its own tail! She awakes with a feeling of peace. As she tries to make sense of her dream, she realizes that she has the solution to her problem. She knows how to use her defective boxes to securely delegate the assembly of even the most intricate pieces!

Like before, she gives a worker a glovebox, box #1, containing the raw materials. But she also gives him several additional gloveboxes, where box #2 contains (locked inside) the key to box #1, box #3 contains the key to box #2, and so on. To assemble an intricate design, the worker manipulates the materials in box #1 until the gloves stiffen. Then, he places box #1 inside box #2, where the latter box already contains a the key to box #1. Using the gloves for box #2, he opens box #1 with the key, extracts the partially assembled trinket, and continues the assembly within box #2 until its gloves stiffen. He then places box #2 inside box #3, and so on. When the worker finally finishes his assembly inside box #*n*, he hands the box to Alice.

Of course, Alice observes, this trick does not work unless the worker can open box #*i* within box #(i + 1), and still have time to make a little bit of progress on the assembly, all before the gloves of box #(i + 1) stiffen. But as long as the unlocking operation (plus a little bit of assembly work) takes less than a minute, and as long as she has enough defective gloveboxes, then it is possible to assemble any piece, no matter how complicated!

4.2. A dream deciphered

In the analogy, the defective gloveboxes represent our somewhat homomorphic encryption scheme, which can perform Add, Sub, and Mult operations on ciphertexts for a little while—it can handle functions in a limited set $\mathcal{F}_\varepsilon$—until the noise becomes too large. What we would like to do is use this somewhat homomorphic scheme to construct a fully homomorphic one.

As before, box #1 with the precious materials inside represents the ciphertexts that encrypt the initial data. Box #(i + 1) with the key for box i inside represents an *encrypted secret decryption key*—i.e., sk_i encrypted under pk_{i+1}.

In the analogy, Alice discovers that there is only one thing that her workers really need to be able to do in less than 1 min with the gloves, aside from performing a very small operation on the piece: unlock box #i within box #(i + 1) and extract the piece. It will turn out that there is only one function that our scheme ε really needs to be able to handle, with a tiny bit of room left over to perform one more Add, Sub, or Mult: the decryption function (which is like unlocking the "encryption box").

If ε has this self-referential property of being able to handle its own decryption function (augmented by a single gate), we say that it is *bootstrappable*. As we will show, if ε is bootstrappable, then one can use ε to construct a fully homomorphic encryption scheme $\varepsilon^\dagger$.

4.3. Bootstrappable to fully homomorphic

Suppose that ε is bootstrappable. In particular, suppose that ε can handle the following four functions: the decryption function, expressed as a circuit D_ε of size polynomial in λ, as well as D_ε augmented by an Add, Sub, or Mult gate modulo 2. (D_ε augmented by Add consists of two copies of D_ε connected by an Add gate.) We will show that this is a *complete* set of circuits, in the sense that if these four circuits are in $\mathcal{F}_\varepsilon$, then one can construct from ε a scheme $\varepsilon^\dagger$ that is fully homomorphic.

As a warm-up, suppose that ciphertext c_1 encrypts the bit m under key pk_1. Suppose also that we have an encrypted secret key: let $\overline{sk_1}$ be a vector of ciphertexts that encrypt the bits of sk_1 under pk_2 via $Encrypt_{\varepsilon}(pk_2, sk_{1j})$. Consider the following algorithm.

$Recrypt_{\varepsilon}(pk_2, D_{\varepsilon}, \overline{sk_1}, c_1)$.

- Generate $\overline{c_1}$ via $Encrypt_{\varepsilon}$ (pk_2, c_{1j}) over the bits of c_1
- Output $c \leftarrow Evaluate_{\varepsilon}$ $(pk_2, D_{\varepsilon}, sk_1, \overline{c_1})$

The decryption circuit D_{ε} has input wires for the bits of a secret key and the bits of a ciphertext. Above, $Evaluate_{\varepsilon}$ takes in the bits of sk_1 and c_1, each encrypted under pk_2. Then, ε is used to evaluate the decryption circuit homomorphically. As long as ε can handle D_{ε}, the output c is an encryption under pk_2 of $Decrypt_{\varepsilon}(sk_1, c_1) = m$. $Recrypt_{\varepsilon}$ therefore outputs a new encryption of m, but under pk_2.

One fascinating thing about $Recrypt_{\varepsilon}$ is that the message m is doubly encrypted at one point, first under pk_1 and next under pk_2. Ordinarily, the only thing one can do with a doubly encrypted message is to peel off the outer encryption first, and then decrypt the inner layer. However, in $Recrypt_{\varepsilon}$, the $Evaluate_{\varepsilon}$ algorithm is used to remove the *inner* encryption, just like Alice unlocks box #i while it is inside box #$(i + 1)$.

It is also useful to imagine that ε is our somewhat homomorphic encryption scheme from Section 3, and consider what $Recrypt_{\varepsilon}$ does to the noise of the ciphertexts. Evaluating D_{ε} removes the noise associated to the first ciphertext under pk_1 (because, of course, decryption removes noise), but $Evaluate_{\varepsilon}$ simultaneously introduces new noise while evaluating the ciphertexts under pk_2. As long as the new noise added is less than the old noise removed, we have made "progress." A similar situation holds in Alice's jewelry store. When the worker extracts the piece from the used-up glovebox #i, this process simultaneously uses up the gloves of box #$(i + 1)$. We have made "progress" as long as the process does not leave box #$(i + 1)$'s gloves completely used-up.

Of course, our goal is to perform actual operations on underlying messages, not merely to obtain a new encryption of the same message. So, suppose that ε can handle D_{ε} augmented by some gate—e.g., Add; call this augmented circuit D_{Add}. If c_1 and c_2 are two ciphertexts that encrypt m_1 and m_2, respectively, under pk_1, and we compute $\overline{c_1}$ and $\overline{c_2}$ as before, as ciphertexts encrypting the bits of the ciphertexts under pk_2, then we have that

$$c \leftarrow \mathrm{Evaluate}_{\varepsilon}(\mathrm{pk}_2, D_{\mathrm{Add}}, \overline{\mathrm{sk}_1}, \overline{c_1}, \overline{c_2})$$

is an encryption under pk_2 of $m_1 \oplus m_2$.

By recursing this process, we get a fully homomorphic encryption scheme. The public key in $\varepsilon^{\dagger}$ consists of a sequence of public keys $(\mathrm{pk}_1, \ldots, \mathrm{pk}_{l+1})$ and a chain of encrypted secret keys $\overline{c_1}$, ..., $\overline{\mathrm{sk}_\ell}$, where sk_i is encrypted under pk_{i+1}. To evaluate a function *f* in $\varepsilon^{\dagger}$, we express *f* as a circuit, topologically arrange its gates into levels, and step through the levels sequentially. For a gate at level i + 1 (e.g., an Add gate), we take as input the encrypted secret key $\overline{\mathrm{sk}_i}$ and a couple of ciphertexts associated to output wires at level *i* that are under pk_i, and we homomorphically evaluate D_{Add} to get a ciphertext under pk_{i+1} associated to a wire at level i + 1. Finally, we output the ciphertext associated to the output wire of *f*.

Putting the encrypted secret key bits $\overline{c_1}$, ..., $\overline{\mathrm{sk}_\ell}$ in $\varepsilon^{\dagger}$'s public key is not a problem for security. These encrypted secret-key bits are indistinguishable from encryptions of 0 as long as ε is semantically secure.

4.4. Circular security

Strictly speaking, $\varepsilon^{\dagger}$ does not *quite* meet our definition of fully homomorphic encryption, since the complexity of $\mathrm{KeyGen}_{\varepsilon^{\dagger}}$ grows linearly with the maximum circuit depth we want to evaluate. (Fortunately, $\mathrm{Encrypt}_{\varepsilon^{\dagger}}$ and $\mathrm{Decrypt}_{\varepsilon^{\dagger}}$ do not depend at all on the function *f* being evaluated.)

However, suppose that ε is not only bootstrappable, but also *circular-secure*—that is, it is "safe" to reveal the encryption of a secret key sk_i under its own associated public key pk_i. Then, we can simplify $\mathrm{KeyGen}_{\varepsilon}{}^{\dagger}$. We do not need distinct public keys pk_i for each circuit level and an acyclic chain of encrypted secret keys. Instead, the public key in $\varepsilon^{\dagger}$ can consist merely of a single public key pk and a single encrypted secret key $\overline{\mathrm{sk}}$ (sk under pk), where pk is associated to all levels of the circuit. This approach has the additional advantage that we do not need to decide beforehand the maximal circuit depth complexity of the functions that we want to be able to evaluate.

For most encryption schemes, including our somewhat homomorphic scheme (as far as we know), revealing an encryption of sk under pk does not lead to any attack. However, it is typically difficult to *prove* that an encryption scheme is circular-secure.

The issue of circular security also fits within our physical analogy. Suppose that a key is locked inside the very same box that the key could open from the outside. Is it possible to use the gloves and key to open the box *from the inside*? If so, it would be a strange lock. Similarly, encryption schemes that are insecure in this setting tend to be contrived.

5. Somewhat Homomorphic to Bootstrappable

Is our somewhat homomorphic encryption scheme from Section 3 already bootstrappable? Can it handle its own decryption circuit? Unfortunately, as far as we can tell, ε can *almost* handle D_ε, but not quite. So, we modify ε slightly, constructing a new (but closely related) somewhat homomorphic scheme ε^* that can handle essentially the same functions that ε can, but whose decryption circuit is simple enough to make ε^* bootstrappable.

5.1. Alice gets her hands dirty

After her dream, Alice rushes to her store to see if her idea works. She locks box #1 and puts it inside box #2. Working with the gloves of box #2, she tries to unlock box #1 in less than 1 min. The thickness of the gloves and the stickiness of the lock combine to make it impossible.

She is despondent until she remembers that she has a special grease that makes her locks less sticky. This time, she locks box #3 and puts it inside box #4. She also puts her bottle of grease inside box #4. Working with the gloves of box #4, she squirts some grease on the lock and then tries to unlock it. But the gloves stiffen before she can finish.

Then, she thinks: why didn't I grease the box's lock *before* putting it inside the other box? That way, I wouldn't waste my valuable time with the gloves greasing the lock.

She locks box #5, greases its lock, and then puts it inside box #6. Working with gloves, she tries the lock again. This time it works, despite the clumsiness of the gloves!

At last, she has a system that lets her securely delegate the processing of her precious materials into arbitrarily complicated pieces! Her workers just need to apply the grease to each box before they put it inside the next box. She can hardly wait to put the system in place the following morning.

5.2. Greasing the decryption circuit

In our somewhat homomorphic encryption scheme ε from Section 3, the decryption function is:

$$m \leftarrow (c \bmod p) \bmod 2$$

Equivalently, but more simply, the equation is:

$$m \leftarrow \mathrm{LSB}(c)\ \mathrm{XOR}\ \mathrm{LSB}(\lfloor c/p \rceil),$$

where LSB takes the least significant bit and $\lfloor \cdot \rceil$ rounds to the nearest integer. This is equivalent, since $(c \bmod p) = c - p \cdot \lfloor c/p \rceil$. Since p is odd, we have that $(c \bmod p) \bmod 2 = c - \lfloor c/p \rceil \bmod 2$. This is just the XOR of the least significant bits of c and $\lfloor c/p \rceil$.

In the decryption circuit D_ε, computing the LSB is immediate: the circuit simply does not have output wires for the more significant bits. Computing an XOR also takes only one gate. If the decryption function is complicated, it must be because computing $\lfloor c/p \rceil$ is complicated. Is the function $f(p, c) = \lfloor c/p \rceil$ (with the few steps afterward) something that ε can handle? If so, ε is bootstrappable, and can be used to construct a fully homomorphic encryption scheme.

Unfortunately, even a single multiplication of long numbers—namely, c with $1/p$—seems to be too complex for ε to handle. The reason is that c and $1/p$ each need to be expressed with at least $P \approx \log p$ bits of precision to ensure that $f(p, c)$ is computed correctly. When you multiply two P-bit numbers, a bit of the result may be a high-degree polynomial of the input bits; this degree is also roughly P. We saw that ε can handle an elementary symmetric polynomial in t variables of degree (roughly) $d < P/(N \cdot \log t)$. However, ε cannot handle even a single monomial of degree P, where the noise of output ciphertext is upper-bounded by $(2^N)^P \approx p^N \gg p/2$. Consequently, ε does not seem to be bootstrappable.

However, if we are willing to get our hands dirty by tinkering with ε to make the decryption function simpler, we eventually get a scheme ε^* that is bootstrappable. The main idea of the transformation is to replace ε's decryption function, which multiplies two long numbers, with a decryption function that adds a fairly small set of numbers. In terms of the bits of the addends, this summation corresponds to a polynomial of fairly low degree that ε^* can handle.

Let us go through the transformation step by step, beginning with $\mathrm{KeyGen}_{\varepsilon^*}$. The transformation uses a couple of integer parameters: $0 < \alpha < \beta$.

- $\mathrm{KeyGen}_{\varepsilon^*}(\lambda)$: Run $\mathrm{KeyGen}_{\varepsilon}(\lambda)$ to obtain keys (pk, sk), where sk is an odd integer p.

Generate a set $\mathcal{F}_{\varepsilon} = \langle y_1, ..., y_\beta \rangle$ of rational numbers in [0, 2) such that there is a sparse subset $S \subset \{1, ..., \beta\}$ of size α with $\Sigma_{i \in S} y_i \approx 1/p \bmod 2$. Set sk* to be the sparse subset S, encoded as a vector $s \in \{0, 1\}^\beta$ with Hamming weight α. Set pk* ← (pk, $\vec{y}$).

The important difference between KeyGen$_{\varepsilon^*}$ and KeyGen$_\varepsilon$ is that KeyGen$_{\varepsilon^*}$ includes a *hint* about the secret integer p—namely, a set of numbers $\vec{y}$ that contains a (hidden) sparse subset that sums to $1/p$ (to within a very small error, and up to addition by an even number). This hint is the "grease," which will be used in Encrypt$_{\varepsilon^*}$ and Decrypt$_{\varepsilon^*}$. Although it is technically not the decryption key sk*, the integer p still can be used to decrypt a ciphertext output by Encrypt$_{\varepsilon^*}$, so revealing this hint obviously impacts security, a point we elaborate on in Section 5.4.

- Encrypt$_{\varepsilon^*}$(pk*, m): Run Encrypt$_\varepsilon$(pk, m) to obtain ciphertext c. For $i \in \{1, ..., \beta\}$, set $z_i \leftarrow c \cdot y_i \bmod 2$ keeping only about log α bits of precision after the binary point for each z_i. The ciphertext c^* consists of c and $\vec{z} = \langle z_1, ..., z_\beta \rangle$.

The important point here is that the hint $\vec{y}$ is used to *postprocess* a ciphertext c output by Encrypt$_\varepsilon$, with the objective of leaving less work remaining for Decrypt$_{\varepsilon^*}$ to do.

This sort of two-phase approach to decryption has been used before in *server-aided cryptography*. (See cites in Gentry[2].) In that setting, a user wants to minimize its cryptographic computation—e.g., because it is using a constrained device, such as a smartcard or handheld. So, it outsources expensive computations to a server. To set up this arrangement, the user (in some schemes) must give the server a hint $\vec{y}$ that is statistically dependent on its secret key sk, but which is not sufficient to permit the server to decrypt efficiently on its own. The server uses the hint to process a ciphertext directed to the user, leaving less work for the user to do. In our setting, the encrypter or evaluator plays the role of the server, postprocessing the ciphertext so as to leave less work for the decryption algorithm to do.

- Decrypt$_{\varepsilon^*}$(sk*, c^*): Output LSB(c) XOR LSB($\lfloor \Sigma_i s_i z_i \rceil$). Decryption works, since (up to small precision errors) $\Sigma_i s_i z_i = \Sigma_i c \cdot s_i y_i = c/p \bmod 2$.

To ensure that the rounding is correct despite the low precision, we need c to be closer (than the trivial $p/2$) to a multiple of p (say, within $p/16$). This makes $\mathcal{F}_{\varepsilon^*}$ smaller than $\mathcal{F}_\varepsilon$, since $\mathcal{F}_{\varepsilon^*}$ is limited to functions where $|f(a_1, ..., a_t)| < p/16$ when the a_i are N bits. This makes only a small difference.

The important point regarding $\text{Decrypt}_{\varepsilon^*}$ is that we replace the multiplication of c and $1/p$ with a summation that contains only α nonzero terms. The bits of this summation can be computed by a polynomial of degree $\alpha \cdot \text{polylog}(\alpha)$, which ε^* can handle if we set α to be small enough.

- $\text{Add}_{\varepsilon^*}$ $(\text{pk}^*, c^*_1, c^*_2)$: Extract c_1 and c_2 from c^*_1 and c^*_2. Run $c \leftarrow \text{Add}_\varepsilon(\text{pk}, c_1, c_2)$. The output ciphertext c^* consists of c, together with the result of postprocessing c with $\vec{y}$ · $\text{Mult}_{\varepsilon^*}(\text{pk}^*, c^*_1, c^*_2)$ is analogous.

5.3. How to add numbers

To see that ε^* can handle the decryption function plus an additional gate when α is set small enough, let us consider the computation of the sum $\Sigma_i\, s_i\, z_i$. In this sum, we have β numbers a_1, ..., a_β, each a_i expressed in binary $(a_{i,0}, ..., a_{i,-l})$ with $l = O(\log \alpha)$, where at most α of the a_i's are nonzero (since the Hamming weight of s is α). We want to express each bit of the output as a polynomial of the input bits, while minimizing the degree of the polynomial and the number of monomials.

Our approach to the problem is to add up the column of LSBs of the numbers—computing the Hamming weight of this column—to obtain a number in binary representation. Then, we add up the column of penultimate bits, etc. Afterward, we combine the partial results. More precisely, for $j \in [0, -l]$, we compute the Hamming weight b_j, represented in binary, of $(a_{1,j}, ..., a_{\beta j})$. Then, we add up the $l + 1$ numbers $b_0, ..., 2^{-l} b_{-l}$ to obtain the final correct sum.

Conveniently, the binary representation of the Hamming weight of any vector $\vec{x} \in \{0,1\}^t$ is given by

$$(e_{2^{\lfloor \log t \rfloor}}(x_1, ..., x_t) \bmod 2, ..., e_{2^0}(x_1, ..., x_t) \bmod 2)$$

where $e_i(x_1, ..., x_t)$ is the ith elementary symmetric polynomial over $x_1, ..., x_t$. These polynomials have degree at most t. Also, we know how to efficiently evaluate the elementary symmetric polynomials. They are simply coefficients of the polynomial $p(z) = \Pi^t_{i=1} (z - x_i)$. An important point is that, in our case, we only need to evaluate the polynomials up to degree α, since we know a priori that each of the Hamming weights is at most α. We saw in Section 3.3 that we can handle elementary symmetric polynomials in t variables of degree up to about $\lambda/\log t = \Omega(\lambda/\log \lambda)$ for our suggested parameters. We can set α to be smaller than this.

The final step of computing the sum of the b_j's does not require much computation, since there are only $l + 1 = O(\log \alpha)$ of them. We get that a ciphertext encrypting a bit of the overall sum has noise of at most $N \cdot \alpha \cdot g(\log \alpha)$ bits for some polynomial g of low degree. If the final sum modulo 2 is $(b'_0, b'_{-1}, \ldots)$ in binary, then the rounding operation modulo 2 is simply b'_0 XOR b'_{-1}. With the additional XOR operation in decryption, and possibly one more gate, the noise after evaluating the decryption function plus a gate has at most $N \cdot \alpha \cdot h(\log \alpha)$ bits for some polynomial h.

The scheme ε^* becomes bootstrappable when this noise has at most $\log(p/16) = P - 4$ bits. For example, this works when $\alpha = \lambda/\text{polylog}(\lambda)$, $N = \lambda$, and $P = \lambda^2$.

5.4. Security of the transformed scheme

The encryption key of ε^* contains a hint about the secret p. But we can prove that ε^* is semantically secure, unless either it is easy to break the semantic security of ε (which implies that the approximate gcd problem is easy), or the following sparse (or low-weight) subset sum problem (SSSP) is easy: given a set of β numbers $\vec{y}$ and another number s, find the sparse (α-element) subset of $\vec{y}$ whose sum is s.

The SSSP has been studied before in connection with server-aided cryptosystems. If α and β are set appropriately, the SSSP is a hard problem, as far as we know. In particular, if we set α to be about λ, it is hard to find the sparse subset by "brute force," since there are $(\beta_\alpha) \approx \beta^\alpha$ possibilities. If the sparse subset sum is much closer to $1/p$ than any other subset sum, the problem yields to a lattice attack. But these attacks fail when we set β large enough (but still polynomial in λ) so that an *exponential* (in λ) number of subset sums are as close to $1/p$ as the sparse subset. Concretely, we can set $\beta = \lambda^5 \cdot \text{polylog}(\lambda)$.

6. Conclusions

We now know that FHE is possible. We already have the scheme presented here, the lattice-based scheme by Gentry,[2,3] and a recent scheme by Smart and Vercauteren.[7]

There is still work to be done toward making FHE truly practical. Currently, all known FHE schemes follow the blueprint above: construct a bootstrappable somewhat homomorphic encryption scheme ε, and obtain FHE by running $\text{Evaluate}_\varepsilon$ on ε's decryption function. But this approach is computationally expensive. Not only is the decryption function expressed (somewhat inefficiently) as a circuit, but then Evaluate$_\varepsilon$ replaces each bit in this circuit with a large ciphertext that encrypts that bit. Perhaps someone will find a more efficient blueprint.

The scheme presented here, while conceptually simpler, seems to be less efficient than the lattice-based scheme. To get 2^λ security against known attacks—e.g., on the the approximate gcd problem—ciphertexts are $\lambda^5 \cdot$ polylog(λ) bits, which leads to $\lambda^{10} \cdot$ polylog(λ) computation to evaluate the decryption function. The lattice-based scheme with comparable security has $\lambda^6 \cdot$ polylog(λ) computation. This is high, but not totally unreasonable. Consider: to make RSA 2^λ-secure against known attacks—in particular, against the number field sieve factoring algorithm—you need to use an RSA modulus with approximately λ^3 bits. Then, RSA decryption involves exponentiation by a λ^3-bit exponent—i.e., about λ^3 multiplications. Even if one uses fast Fourier multiplication, this exponentiation requires $\lambda^6 \cdot$ polylog(λ) computation. Also, unlike RSA, the decryption function in our scheme is highly parallelizable, which may make an enormous difference in some implementations.

7. Epilogue

The morning after her dream, Alice explains her glovebox solution to her workers. They are not happy, but they wish to remain employed. As the day progresses, it becomes clear that the gloveboxes are slowing down the pace of jewelry construction considerably. The main problem seems to be the thick gloves, which multiply the time needed for each assembly step. After a few days of low output, Alice curtails her use of the gloveboxes to pieces that contain the most valuable diamonds.

Alice loses her suit against Acme Glovebox Company, because, as far as anyone knows in Alice's parallel world, gloves in gloveboxes are always very stiff and stiffen completely after moderate use. The old judge explains this to her in a patronizing tone.

But Alice refuses to give up. She hires a handsome young glovebox researcher, and tasks him with developing a glove flexible enough to permit the nimble assembly of jewels and unlocking of boxes, but sturdy enough to prevent the boxes from being easily compromised. The researcher, amazed at his good fortune, plunges into the problem.

References

1. Boneh, D., Lipton, R.J. Algorithms for black-box fields and their application to cryptography (extended abstract). In *CRYPT* (1996), 283–297.

2. Gentry, C. *A fully Homomorphic Encryption Scheme*. Ph.D. thesis, Stanford University, 2009. crypto.stanford.edu/craig.

3. Gentry, C. Fully homomorphic encryption using ideal lattices. *STOC*. M. Mitzenmacher ed. ACM, 2009, 169–178.

4. Goldwasser, S., Micali, S. Probabilistic encryption. *J. Comp. Syst. Sci. 28*, 2 (1984), 270–299.

5. Rivest, R.L., Adleman, L.M., Dertouzos, M.L. On data banks and privacy homomorphisms. In *Foundations of Secure Computations* (1978), 169–180.

6. Rivest, R.L., Shamir, A., Adleman, L.M. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM 21*, 2 (1978), 120–126.

7. Smart, N.P., Vercauteren, F. Fully homomorphic encryption with relatively small key and ciphertext sizes, 2009. http://eprint.iacr.org/2009/571.

8. van Dam, W., Hallgren, S., Ip, L. Quantum algorithms for some hidden shift problems. *SIAM J. Comp. 36*, 3 (2006), 763–778.

9. van Dijk, M., Gentry, C., Halevi, S., Vaikuntanathan, V. Fully homomorphic encryption over the integers, 2009. http://eprint.iacr.org/2009/616.

Author

Craig Gentry (cbgentry@us.ibm.com), IBM T.J. Watson Research Center, Hawthorne, NY.

Footnotes

This paper draws from the STOC 2009 paper "Fully Homomorphic Encryption Using Ideal Lattices," my thesis, and a recent manuscript co-authored with van Dijk, Halevi, and Vaikuntanathan.

DOI: http://doi.acm.org/10.1145/1666420.1666444