Computing with Data Privacy: Steps toward Realization

David W. Archer | Galois
Kurt Rohloff | New Jersey Institute of Technology

Two new cryptographic methods—linear secret sharing (LSS) and fully homomorphic encryption (FHE)—allow computing on sensitive data without decrypting it. LSS and FHE differ in speed, ease of use, computational primitives, and cost.

Users often don't trust computing environments such as shared clouds to perform computation on sensitive data. Only recently has it become possible to address this trust concern with general-purpose computation on encrypted data. In this article, we discuss two forms of such computation: linear secret sharing (LSS)[1] and fully homomorphic encryption (FHE).[2]

In LSS, a user or group of users, each with private data, encrypts the data and sends it to a group of untrusted servers. These servers share the computation without decrypting the data and return still-encrypted results. In FHE, a user encrypts data and sends it to a single untrusted server, which computes an encrypted answer and returns it to the user.

Computation time for both approaches is many orders of magnitude slower than computation "in the clear." In addition, LSS requires multiple servers to perform computation and significant communication bandwidth among them. FHE typically imposes significant expansion in ciphertext size relative to plaintext, which affects both memory utilization and network bandwidth.

We created prototypes including LSS- and homomorphic encryption (HE)–based variations of voice-over-IP (VoIP) teleconferencing systems using Amazon Elastic Cloud nodes to mix encrypted voice streams from iPhone handsets, an LSS-based email guard using regular expression search to determine which messages to transmit, and an FHE-based email guard using string comparison to filter email.

Protocols, Adversary Models, and Security Guarantees

Here, we describe our secure computation systems as well as applicable adversary models and security guarantees.

Linear Secret Sharing

In LSS, multiple proxies collaboratively compute a function on behalf of one or more clients.[1] Each client distributes to each proxy a share of its secret input. Each share is essentially random—a fixed linear function of the secret input and random values selected by the client. Thus, no proxies learn anything about the input. LSS works because its systems exhibit homomorphisms to mathematical structures of interest such as the integers, allowing parties holding shares to compute functions of secrets by arithmetically manipulating only their shares of those secrets.

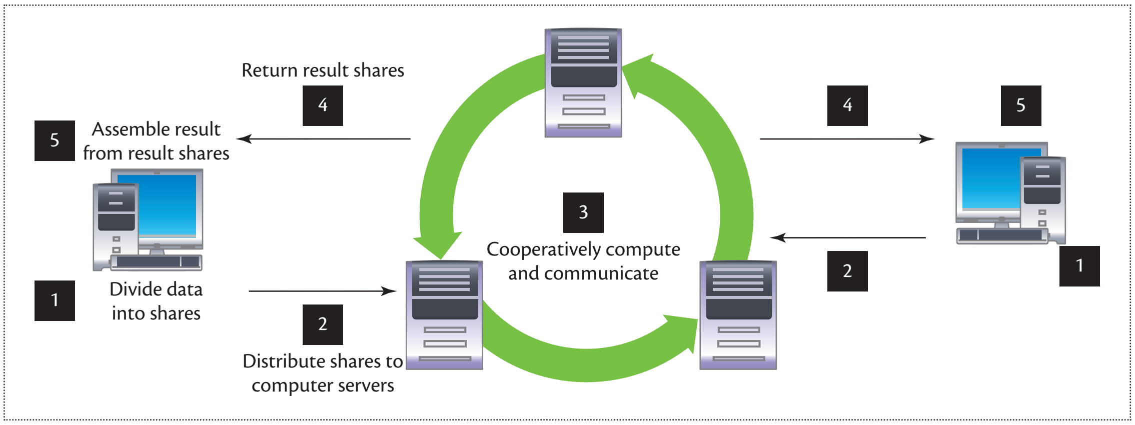
Return result shares
4
5
Assemble result from result shares
1
Divide data into shares
2
Distribute shares to computer servers
3
Cooperatively compute and communicate
4
5
2
1

Figure 1. Our linear secret sharing (LSS) protocol. Each client encrypts its input with three cipher streams, producing a metashare (step 1). Each client transmits its metashare to the coordination server (not shown) over a secure channel, which in turn distributes these metashares to the three proxies (step 2). Each proxy computes its share from each metashare by decrypting the metashare using one cipher stream (that it and the client providing the metashare both know), and then performs the desired computation, communicating with other proxies as needed over secure channels (step 3). Each proxy encrypts its result share using a distinct cipher stream it shares with the client, and then sends it to the coordination server, which computes the XOR of all result shares into a result metashare and forwards this to clients (step 4). Each client decrypts the metashare to obtain the computation result (step 5).

As a simple example, suppose clients Alice and Bob agree to add secret inputs X and Y that they respectively hold. Assume X and Y are in $[0 \ldots 2^n - 1]$ for natural number n. Alice computes three shares of X by choosing random X_1 and X_2 from $[0 \ldots 2^n - 1]$, and then choosing X_3 such that $X = (X_1 + X_2 + X_3) \bmod 2^n$. Alice then distributes these three shares to the proxies over secure channels, such that each proxy holds one distinct share. Bob does the same for Y. The proxies add their shares, resulting in each proxy holding one of $X_1 + Y_1$, $X_2 + Y_2$, or $X_3 + Y_3$. Note that none of these result shares reveal anything about $X + Y$ to the proxies that hold them. The proxies send these result shares to Alice or Bob over secure channels. Bob or Alice then adds them together to obtain $X + Y$.

While communication from clients to proxies in typical LSS systems is direct, we found that having mobile clients distribute shares directly to each proxy resulted in substantial loading of client Wi-Fi channels. To address such Wi-Fi overload, we extended our applications' communication model to introduce an untrusted coordination server. Clients cryptographically combine all three shares they compute into a single metashare that's sent to the coordination server. This server, which we locate in a richer bandwidth environment along with the proxies, distributes the metashare to all three proxies, which compute their own shares from the metashare and preshared key material.

The core of our LSS system, ShareMonad, consists of a Haskell-embedded (www.haskell.org) domain-specific language (DSL) for expressing LSS computation, a compiler to transform ShareMonad code into abstract syntax trees suitable for interpretation, and a three-proxy LSS interpreter. Each proxy in a ShareMonad application runs this interpreter. Clients and coordination servers run application code that interoperates with the proxy code. Thus, each of our LSS applications consists of a composition of code running on clients, coordination server code, and ShareMonad code running on proxies.

Our LSS DSL provides operations including addition, subtraction, multiplication, unsigned division, comparisons, bitwise shift right, conversion between $[0 \ldots 2^n - 1]$ and bit vector representations, table lookups, and operations on bit vectors. ShareMonad protocols currently assume an honest but curious adversary: proxies are assumed to compute and communicate as agreed but might observe attached channels and local computations.

As Figure 1 shows, our LSS protocols typically proceed in several steps:

1. Each client encrypts its input with three cipher streams, producing a metashare.
2. Each client transmits its metashare to the coordination server (not shown) over a secure channel, which in turn distributes these metashares to the three proxies.
3. Each proxy computes its share from each metashare by decrypting the metashare using one cipher stream (that it and the client providing the metashare both know), and then performs the desired computation, communicating with other proxies as needed over secure channels.

4. Each proxy encrypts its result share using a distinct cipher stream it shares with the client, and then sends it to the coordination server, which computes the XOR of all result shares into a result metashare and forwards this to clients.
5. Each client decrypts the metashare to obtain the computation result.

We compute metashares and shares as follows. We distribute in advance a cryptographic key between each client *CL* and each proxy *A*, *B*, and *C*. This key seeds stream ciphers used to form metashares from input data. To compute the metashare X_m of secret *X*, a stream cipher is used to generate a random value R_A that undergoes bitwise XOR with *X*. *CL* repeats this process with the cryptographic key it shares with *B* and *C*, obtaining $X_m = X \text{ XOR } R_A \text{ XOR } R_B \text{ XOR } R_C$, which it sends to the coordination server to be forwarded to all three proxies. *A* uses the key it shares with *CL* to compute R_A, which it uses to compute its share of X, $X_1 = X_m \text{ XOR } X_A = X \text{ XOR } R_B \text{ XOR } R_C$. *B* and *C* similarly compute their shares X_2 and X_3, respectively. Note that *X* can trivially be recovered from these shares: $X = X_1 \text{ XOR } X_2 \text{ XOR } X_3$.

Once shares are computed, computation of the desired function proceeds on the proxies. In the case of addition, no communication among proxies is required: *A* computes result share $R_1 = X_1 + Y_1$, *B* computes $R_2 = X_2 + Y_2$, and *C* computes $R_3 = X_3 + Y_3$. Once computation is complete, *A*, *B*, and *C* send R_1, R_2, and R_3, respectively, to the coordination server, which computes the values' bitwise XOR, and forwards this single metaresult to the clients for final decryption.

Note that naively following this return transmission protocol would reveal all shares of the computation result to the untrusted coordination server. We avoid this security lapse by having *A*, *B*, and *C* encrypt R_1, R_2, and R_3, respectively, using keys shared between *A*, *B*, *C*, and the client to enable decryption by the client.

Some computations, such as $X \times Y$, require communication among proxies. $X \times Y = (X_1 + X_2 + X_3) \times (Y_1 + Y_2 + Y_3)$ involves not only locally computable terms such as $X_1 \times Y_1$ but also terms such as $X_2 \times Y_3$. These terms require that each proxy communicate its share to one other proxy. We follow the method that Dan Bogdanov and his colleagues described: sharing among proxies occurs in symmetric rounds, and each proxy adds new entropy to its share before sending that share to a neighbor.[5] Thus, even though proxies communicate their shares to other proxies, the communicated values don't allow those proxies to gain any knowledge of the original secret.

We ensure privacy in each portion of our protocol in Figure 1, except those that execute on the trusted client platforms:

- Passphrase sharing prior to computation is handled by well-known asymmetric cryptographic (public-key infrastructure) protocols. The Advanced Encryption Standard (AES) and the National Institute of Standards and Technology SP 800-90 standard provide cryptographically secure random numbers for creating shares.
- Transmission of metashares X_m from client to coordination server and onward to proxies is protected by the entropy added during creation of X_m.
- Local computation on the proxies is protected from observation because it's performed only on cryptographic shares. We prevent accumulation of too many shares on a single proxy by introducing additional entropy during the sharing process, as we described.
- Transmission from proxies to the coordination server is protected by encryption of result shares introduced by the proxies, which prevents the server from combining result shares to obtain the result in the clear. Transmission from the coordination server to the clients is also protected by this encryption.

Homomorphic Encryption

Like all secure encryption schemes, secure HE schemes make it intractable, under certain computational hardness assumptions, to recover information about plaintext from its encrypted ciphertext.[2,3] We use a representative approach to HE that employs a multidimensional lattice over a finite field. We use a vector basis to represent the lattice. Each plaintext input to the computation is encrypted to a ciphertext encoded as a vector—represented as a large matrix—not in the lattice. Security is based on the *closest-vector problem*: a known hard problem of finding the lattice vector with the least distance to a given vector—in our case, the ciphertext vector.

Computation on encrypted data proceeds by manipulating ciphertext matrix representations. However, encryption embeds noise into these representations. As computation proceeds, this noise grows. If too much noise accrues, decryption might identify the wrong lattice vector and thus return the wrong plaintext. We can decrease ciphertext noise by increasing the dimensionality of the ciphertext's matrix while maintaining security. Increasing the dimensionality of the matrix allows for more computation to be performed before too much noise accumulates but also results in computationally difficult manipulations of large matrices. Even with such noise reduction, noise still accumulates, ultimately limiting the depth of the computation available. FHE systems such as ours avoid this limitation by *bootstrapping*—periodically performing a cryptographic operation that resets the noise level without compromising security. Craig Gentry described an early form

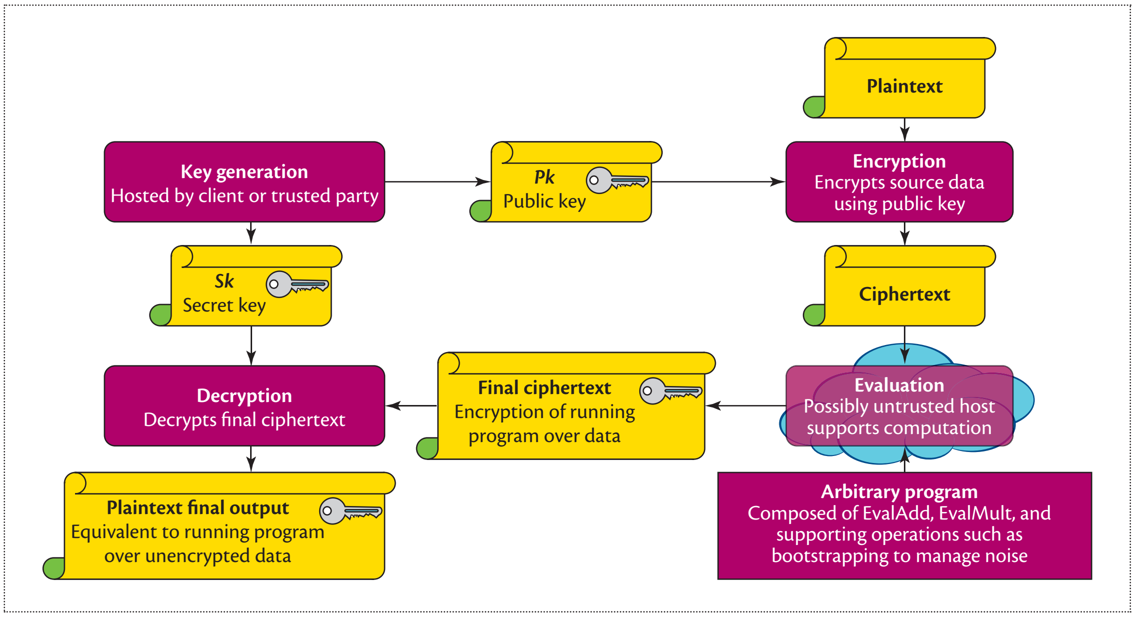
Plaintext
Key generation
Hosted by client or trusted party
Pk
Public key
Encryption
Encrypts source data using public key
Sk
Secret key
Ciphertext
Decryption
Decrypts final ciphertext
Final ciphertext
Encryption of running program over data
Evaluation
Possibly untrusted host supports computation
Plaintext final output
Equivalent to running program over unencrypted data
Arbitrary program
Composed of EvalAdd, EvalMult, and supporting operations such as bootstrapping to manage noise

Figure 2. Dataflow in our fully homomorphic encryption (FHE) system. The key infrastructure (upper left) runs on a trusted host and uses the NTRU public-key approach to generate key pairs public key (*Pk*) and secret key (*Sk*). *Pk* is shared with a data source (on the right) that encodes plaintext messages as mod *p* integers and then encrypts the data using that key. A program source (lower right) provides a program, implemented as a Boolean circuit, to be evaluated over the encrypted data. The ciphertext, a public-key encryption of *Sk*, and the program are sent to a computation host (cloud, lower right). The result ciphertext is sent to the client (lower left) that decrypts it using *Sk* to obtain the plaintext result.

of bootstrapping and the resulting capability to perform arbitrary-depth secure computation.[2]

Figure 2 shows our FHE system's high-level dataflow. The key infrastructure on the upper left runs on a trusted host and uses the NTRU[4] public-key approach to generate key pairs consisting of a public key *Pk* and secret key *Sk*. The *Pk* is shared with a data source (on the right) that encodes plaintext messages as mod *p* integers and then encrypts the data using that key to generate the initial ciphertext. A program source (on the lower right) provides a program, implemented as a Boolean circuit, to be evaluated over the encrypted data. The initial ciphertext, a public-key encryption of the corresponding *Sk*, and the program are sent to a computation host (shown as a cloud, on the lower right). The resulting final ciphertext is sent to the client (on the lower left) that decrypts it using *Sk* to obtain the plaintext result. The protocols we use are secure against "honest but curious" adversaries such as an untrusted host performing the computation honestly while seeking to discover secret inputs.

Our FHE programs comprise two computational primitives: *EvalAdd* (addition) and *EvalMult* (multiplication). We use these primitives to construct operations for encryption, decryption, and bootstrapping. We implement modulus reduction, ring reduction, and key-switching operations to enable larger depth of computation before bootstrapping, without decreasing security. (In this article, the term *ring* refers to a mathematical ring over the integers.) We also implement specialized primitives, such as ring addition, ring multiplication, and Chinese Remainder Theorem (CRT), because manipulating ciphertexts in CRT representation is more efficient than in power basis representations.

Some early homomorphic systems relied on encoding a single bit of plaintext in each ciphertext. EvalAdd and EvalMult operations were thus simplified into Boolean XOR and AND operations but offered no computation parallelism. Ciphertext-to-plaintext expansion in such systems is quite large: in one of our early examples, the ciphertext expansion ratio was 2^{23}. In contrast, our system encrypts mod *p* integers ($p > 2$) instead of single bits, and we leverage single-instruction, multiple data (SIMD) approaches to pack multiple mod *p* integers into each ciphertext, thus computing parallel operations on these packed integers. Although this approach offers more efficiency, leveraging its inherent parallelism can make algorithm design challenging.

We use a variation of the double-CRT approach along with a residue number system (based on the CRT over the integers) to circumvent the problem of large ciphertext moduli and correspondingly large ciphertext size. For ring dimension n, each ciphertext is represented by an $n \times t$ matrix of t length–n integer vectors of mod q_i values for pairwise coprime moduli q_i. This contrasts with some previous FHE systems that represent ciphertexts as a single integer vector mod Q, where $Q = q_1 * \cdots * q_t$. In our system, the number of moduli, t, grows to support the secure execution of larger programs, but the number of moduli $q_1, \ldots, q_t$ does not. With this representation, we securely represent ciphertexts as matrices of 64-bit integers yet still execute efficiently on commodity computing hardware that would make computation over the multihundred-bit or multithousand-bit single-vector integer representations used in previous systems infeasible.

The security level of lattice-based homomorphic encryption systems isn't often expressed in terms of the work factor used to describe security in typical cryptosystems. Instead, security is typically expressed as the *root Hermite factor* δ, a representation of the hardness of the closest-vector problem. A lattice-based encryption system becomes more secure as δ approaches 1. We selected the value $\delta = 1.007$ for our work, which corresponds roughly to the work factor required to crack AES 128-bit encryption.

The maximum depth of computation d that can be supported between bootstraps and the ring dimension n, which correlates directly to the length of ciphertext vectors, significantly impacts both δ and performance. We have found that with n = 1,6384 and d = 16, we achieve $\delta = 1.007$ while supporting significant computation, such as searching several pages of encrypted text for an encrypted keyword, between bootstraps. With n = 16,384 and efficient packing of ciphertexts, each ciphertext expands to between 10^3 and 10^6 times larger than the corresponding plaintext.

Our system runs in a compiled C environment auto-generated from Matlab implementations (www.mathworks.com/products/matlab). We use parallelism to take advantage of multicore processors in a Linux environment. At $\delta = 1.007$, we encrypt ciphertexts in less than 100 milliseconds in such environments and decrypt in approximately 1 millisecond. EvalAdd on ciphertexts takes several milliseconds, whereas EvalMult takes approximately 500 milliseconds and bootstrapping takes approximately five minutes.

Real-World Potential for FHE and LSS Implementations

Here, we present our prototype applications and their limitations.

VoIP Teleconferencing

Typical VoIP implementations don't provide end-to-end encryption. Instead, they rely on a trusted server to receive content from clients, decrypt that content, reencrypt it, and then distribute it to other clients. This trusted server is a weak point in securing VoIP communication.

Our teams independently developed LSS and FHE VoIP audio conferencing approaches that provide end-to-end security with performance suitable for three or more simultaneous users and high-quality audio. No prior work has demonstrated the application of these technologies to streaming applications such as VoIP. Both our prototypes use Apple iPhone 5s handsets, Amazon cloud-based virtual servers, and suitably modified open source VoIP client and server code.

LSS-based VoIP. Figure 3 shows our LSS VoIP architecture. Each iPhone runs a version of the Mumble VoIP client application (http://mumble.sourceforge.net) with the following modifications: Mumble audio processing samples the microphone at 16 Kbps and logarithmically compresses this to a standard 8-bit μ-LAW floating-point representation.[6] We added encryption for turning each sample into a metashare by computing XOR of each sample with elements drawn from three AES 128-bit counter-mode cipher streams seeded from pre-placed passphrases. The network interface packs 1,440 sample metashares (90 milliseconds of audio data) into each transmitted network packet.

As Figure 3 shows, each client creates and then sends each metashare packet via Wi-Fi (802.11ac) to an Apple Airport Extreme wireless access point, which forwards it to a virtualized coordination server in the Amazon Elastic Cloud Service (ECS). This virtual machine runs a modified version of uMurmur (https://code.google.com/p/umurmur) to handle user session management and audio stream routing. Our uMurmur variant distributes each client audio packet to each of three proxies, gathers result share packets from those proxies after computation, computes XOR on the result shares together sample-wise, and sends the resulting metashare to clients for decryption.

Our proxies, which are also virtual machines hosted in the Amazon ECS, run our ShareMonad audio processing application. Each proxy recreates one of the three entropy streams and uses this to compute its share of each sample from the received metashares. Collectively, the proxies obliviously decode each logarithmically compressed audio stream to a linear, integer representation; mix all decoded audio streams together; clip the resulting audio signal; and recompress the result for distribution.

This computation repeats for each participating client, omitting that client's audio stream so users don't hear their own voices. Each audio stream result share is sent back to the coordination server, where it undergoes XOR with shares from other proxies and is then sent to client handsets for decryption and playback.

A hand-optimized approach required 12 seconds of processing per 1,440-sample block for four users,

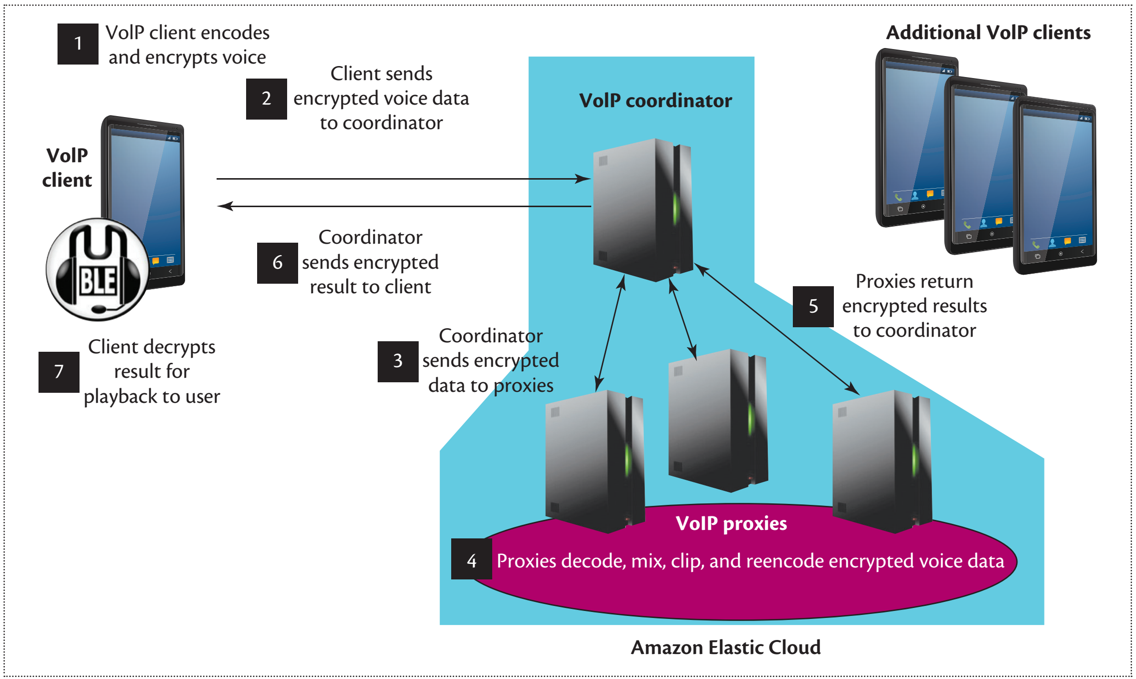
1 VoIP client encodes and encrypts voice
2 Client sends encrypted voice data to coordinator
VoIP coordinator
Additional VoIP clients
VoIP client
BLE
6 Coordinator sends encrypted result to client
5 Proxies return encrypted results to coordinator
7 Client decrypts result for playback to user
3 Coordinator sends encrypted data to proxies
VoIP proxies
4 Proxies decode, mix, clip, and reencode encrypted voice data
Amazon Elastic Cloud

Figure 3. LSS-based voice-over-IP (VoIP) system architecture. iPhone 5s VoIP clients sample audio input at 16 Kbps, encode samples to a standard 8-bit μ-LAW floating-point representation, and encrypt the resulting encoded samples using three Advanced Encryption Standard 128-bit counter-mode cipher streams. Clients send packets of 1,440 encrypted samples (90 ms of audio) over Wi-Fi 802.11ac and through the Internet to proxy servers in the Amazon Elastic Cloud that decode, add, and clip the sample streams without decrypting them. The resulting combined audio stream is reencrypted and sent back to the clients for decryption and playback.

exceeding the 90-millisecond limit required to maintain processing at streaming rates. Applying an LSS index lookup over a public table[7] of precomputed results for the decode-mix-clip-encode function let us reduce this delay to 25 milliseconds, allowing sufficient time to meet the 90-millisecond goal and compensate for network delays between handsets and servers. With this optimization, we achieved streaming throughput for up to four voices at 16 Kbps audio rates, enabling users to communicate clearly.

We used 16-core (C3 size) Amazon cloud servers as proxies, resulting in roughly 80 percent CPU utilization. In contrast, plaintext processing at this performance level requires only a small portion of a single CPU core. Memory use was small and not a constraining factor. Network bandwidth available in our Amazon cloud instances was sufficient with no special optimization.

In the absence of collusion among the proxies, our solution provides two layers of AES 128-bit security at each proxy. Each proxy receives metashares encrypted with three AES 128-bit counter-mode cipher streams yet has access to only one of these cipher streams. Thus, adversaries observing from any one proxy can learn nothing of the plaintext audio samples used as input. Adversaries observing from the coordination server can learn nothing about the input from the metashares it conveys, because that server holds none of the cipher streams used for encryption and decryption. Because each proxy adds new layers of encryption (using cipher streams to which the coordination server has no access) to the result shares it sends back to the coordination server, that server similarly can learn nothing of the computation result.

FHE-based VoIP. We developed an FHE-based approach to secure VoIP teleconferencing that requires only a single proxy. This advance is built on a vocoder technology that takes voice samples from each client as input and encodes those samples as vectors of integers that are then encrypted. This vocoder is linear and can be used with an additive HE scheme to provide an encrypted VoIP teleconferencing capability. Encoded voice samples are encrypted at each iPhone client with the client's public key, using the additive HE scheme.

For our prototype, all clients use the same key, because our focus is on demonstrating the practical feasibility of an FHE computation rather than on well-understood security concerns. The resulting ciphertexts are sent to a VoIP mixer that queues and adds the ciphertext from the clients without decrypting the data or sharing keys. The resulting added ciphertext is sent

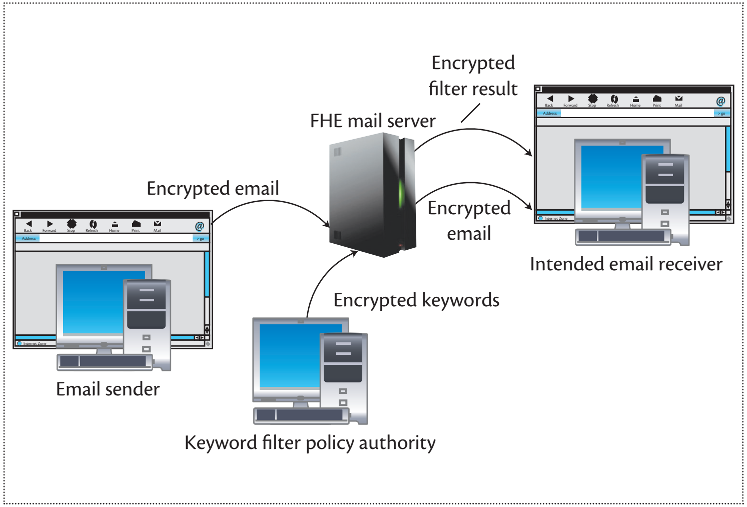

Figure 4. Email border guard system architecture. In the LSS version of the border guard, the mail server connects to three proxies (not shown in the figure). An email client plug-in computes a sent message's metashare using key material shared a priori with the proxies and sends the metashare to the mail server, which distributes it to the proxies. The proxies collectively search the encrypted message, producing shares of a Boolean indication of a regular expression match. In the FHE version, the client homomorphically encrypts the message and sends it to a single proxy that searches the encrypted message for matches with a predefined set of strings, also producing a Boolean indication of a match. The mail server (in the LSS case) or the client (in the FHE case) receives the computation result and uses it to determine whether to forward the message.

back to the clients. When decrypted with the clients' private key using the additive homomorphic decryption scheme, decoded using our decoding scheme, and played back to the clients, the resulting audio is a mix of all the clients' audio streams.

Our FHE-based VoIP uses a prototype architecture similar to the LSS-based VoIP teleconferencing capability, but with lower end-to-end latency. When we ran this system with a server in Virginia and clients in Massachusetts, the total latency was on the order of 80 milliseconds, with the latency roughly split among communication, encryption, and decryption. The mixing latency was nearly trivial, taking less than 1 millisecond.

With our FHE-based approach, no keys are stored on the teleconference server, so privacy is preserved even if adversaries view all communication links and server operations. Trust in the communication links or teleconference server isn't required to provide privacy. The security level provided in the current demo is roughly at the level of AES 128-bit encryption, but parallels between the security levels of our encryption scheme and other current standards aren't exact. We can increase our teleconference capability's security level arbitrarily at the expense of bandwidth requirements or voice quality by modifying the sampling rate and dynamic range of the sampled voice data.

Email Border Guards

Providing privacy using email encryption and achieving information security using trusted-party email filtering at network boundaries are mutually exclusive goals. Either email must be decrypted to verify compliance to InfoSec policies (compromising privacy), or those policies must be enforced by each user prior to message encryption (compromising trust in filtering). We explored solutions to this problem by studying applications in which transaction throughput is important. In our solutions, users encrypt email messages on their trusted computer. The messages are sent to an untrusted mail server for forwarding to a destination. This mail server also acts as a *border guard*, checking each email message for certain content and passing it on to its destination only if that content is absent. The border guard performs this content checking without decrypting the messages.

LSS-based regular expression search email guard. We use the Claws email client and a typical email server, along with plug-ins to each via standard APIs, to search each outbound encrypted email for occurrences of text that match a set of prespecified regular expressions, forwarding messages that do not include such matches and rejecting those that do.

Figure 4 shows our system architecture. In the LSS version, the mail server connects to three proxies that perform the LSS computation (not shown in the figure). When a user sends a message, a plug-in to the Claws client computes the message's metashare using key material shared a priori with the proxies. The email client sends the metashare to the mail server, where a Milter (www.milter.org) plug-in distributes it to the proxies, each of which derives its share. The regular expression set is compiled into a Boolean circuit and distributed to the proxies in advance. The proxies collectively compute the regular expression search on the message, using an adaptation of a mechanism that transforms regular expressions into finite automata.[8] Each server produces one share of the Boolean indication of whether any regular expressions match against any portion of the encrypted message corpus. Our Milter plug-in combines these shares to obtain a plaintext Boolean answer, which it passes to the mail server. The mail server then accepts and forwards the message, or it drops the message and informs the sender's client, as appropriate.

We performed several experiments on this system, optimizing the resulting Boolean circuit to consider different numbers of regular expression characters. Processing 16 message characters at a time was the point of diminishing returns. For a typical 1-Kbyte email ASCII message and a set of regular expressions that roughly represents classification markings that might be used in a government setting, checking an email message took approximately 90 seconds using quad-core, 3-GHz Intel architecture blade servers as proxies. CPU utilization averaged approximately 90 percent during processing, and memory utilization was minimal.

FHE-based encrypted keyword search email guard. We developed a prototype FHE application that searches for encrypted keywords in encrypted text. This method relies on a homomorphic string comparison operation that's repeated for all keywords in all locations of an encrypted message. As in the LSS method, we imported this technology into an email guard–type scenario to provide outsourced email filtering based on email clients' keywords of interest. Because the result of the string comparison is only available to the mail server in encrypted form, our protocol sends the encrypted result back to the client, where it's decrypted to reveal whether the message should be sent. Thus, our prototype assumes an honest sender and requires an extra round-trip between client and server. Figure 4 shows a sketch of this technology.

We're currently running this implementation at a low security level ($\delta = 1.08$) to enable the email system to be interactive with fast response times. Our initial implementation uses a ring dimension of $n = 512$ and encrypts emails with a supported depth of computation $d = 12$. This results in an effective ciphertext modulus q represented with 430 bits. With these parameter configurations, we can sort over encrypted paragraph-long emails with five- to six-character words in less than a minute. Result decryption runs in a matter of seconds.

We could tune this FHE-based email guard to an extremely secure setting ($\delta = 1.0055$ or less) using our current implementation with a similar depth of computation. We would choose a ring dimension of 16,384 and an effective ciphertext modulus Q represented with 521 bits. Encryption runtime at these settings is on the order of minutes, encrypted message filtering would take hours on a nonparallelized server, and decryption would take a matter of seconds.

In a world in which Bob and Alice need to work together but are no longer comfortable sharing their secrets, or where Alice needs Charlie's help to process data but feels uncomfortable with Charlie (or the ever-lurking Eve) seeing the data, secure computation holds promise. However, secure computation methods differ; each has its distinct tradeoffs, security models, and caveats. Our experiments show that some practical applications are emerging, but substantive work remains to be done to make secure computation practical for broad classes of applications. ■

Acknowledgments

We greatly appreciate the contributions of Drew Dean of SRI International in editing and reviewing this manuscript. The Defense Advanced Research Projects Agency (DARPA), the Air Force Research Laboratory (AFRL), and the Office of Naval Research (ONR) under contracts FA8750-11-C-0098 and N00014-11-C-0333 sponsored this work. The views expressed are those of the authors and do not necessarily reflect the official policy or position of the US Department of Defense or the US government. Distribution statement "A" (approved for public release, distribution unlimited).

References

1. R. Cramer, I. Damgard, and U. Maurer, "General Secure Multi-party Computation from Any Linear Secret-Sharing Scheme," *Proc. 19th Int'l Conf. EuroCrypt*, 2000, pp. 316–334.
2. C. Gentry, "Fully Homomorphic Encryption Using Ideal Lattices," *Proc. 41st Ann. ACM Symp. Theory of Computing* (STOC 09), 2009, pp. 169–178.
3. Z. Brakerski and V. Vaikuntanathan, "Fully Homomorphic Encryption from Ring-LWE and Security for Key Dependent Messages," *Proc. 31st Ann. Conf. Advances in Cryptology*, 2011, pp. 505–524.
4. J. Hoffstein, J. Pipher, and J. Silverman, "NTRU: A Ring-Based Public-Key Cryptosystem," *Proc. 3rd Int'l Symp. Algorithmic Number Theory*, LNCS 1423, Springer, 1998, pp. 267–288.
5. D. Bogdanov et al., "High-Performance Secure Multi-party Computation for Data Mining Applications," *Int'l J. Information Security*, vol. 11, no. 6, 2012, pp. 403–418.
6. *Pulse Code Modulation (PCM) of Voice Frequencies*, Int'l Telecommunication Union, ITU-T Recommendation G.711, 1993.
7. J. Launchbury et al., "Application-Scale Secure Multiparty Computation," *Proc. 23rd Ann. European Symp. Programming*, LNCS 8410, 2014, pp. 8–26.
8. S. Fischer, F. Huch, and T. Wilke, "A Play on Regular Expressions: Functional Pearl," *ACM SIGPLAN Notices*, vol. 45, no. 9, 2010, pp. 357–368.

David W. Archer is a research lead at Galois. His research interests include information provenance and trustworthiness, and information assurance. Archer received a PhD in computer science from Portland State University. Contact him at dwa@galois.com.

Kurt Rohloff is an associate professor of computer science at the New Jersey Institute of Technology. His research interests include homomorphic encryption, large-scale distributed computing, and secure computation. Rohloff received a PhD in electrical engineering and computer science from the University of Michigan. Contact him at kurt.rohloff@njit.edu.