2010 Ninth IEEE International Symposium on Network Computing and Applications

GROK: A Practical System for Securing Group Communications

Joseph A. Cooley, Roger I. Khazan, Benjamin W. Fuller, Galen E. Pickard

MIT Lincoln Laboratory
244 Wood St.
Lexington, MA 02420 USA

{cooley,rkh,bfuller,gpickard}@ll.mit.edu

***Abstract*—We have designed and implemented a general-purpose cryptographic building block, called GROK, for securing communication among groups of entities in networks composed of high-latency, low-bandwidth, intermittently connected links. During the process, we solved a number of non-trivial system problems. This paper describes these problems and our solutions, and motivates and justifies these solutions from three viewpoints: usability, efficiency, and security.**

The solutions described in this paper have been tempered by securing a widely-used group-oriented application, group text chat. We implemented a prototype extension to a popular text chat client called Pidgin and evaluated it in a real-world scenario. Based on our experiences, these solutions are useful to designers of group-oriented systems specifically, and secure systems in general.

I. Introduction

Group-oriented communication is an important class of networked applications. Examples include collaborative applications, such as text and voice chat, and content distribution applications, such as video broadcasts. Securing communication in these types of applications can be done with a key that is shared among legitimate group members. Numerous schemes have been proposed for distributing keys to group members and efficiently replacing these keys with new ones when a group's membership changes [1]. The designs of these group keying schemes typically focus on the algorithms and their optimizations for addressing perceived bottlenecks, such as communication and computation.

In order for such group keying schemes to be used in real systems, a number of practical details; such as APIs, identity and key management, communications, and storage, need to be addressed in a system's design and implementation. How these details are addressed has a non-trivial effect on the system, and can, in practice, make or break the system's usability, efficiency, and security [2], [3]. Consequently, addressing these details can constitute the core of creating a secure system.

We recently designed and implemented a general-purpose cryptographic building block, called GROK, for securing communication among groups of members. We had to solve a number of non-trivial system problems. This paper describes these problems and our solutions, and motivates and justifies these solutions from three viewpoints: usability, efficiency, and security.

This paper does not describe in detail the specific group keying schemes used by GROK; such descriptions are subject of a separate paper [4]. The contributions of this paper are the solutions to the non-trivial problem of creating a secure system in general, and a group-oriented system in particular. Specifically, this paper demonstrates:

- A simple and generic API for securing group keys and group-oriented messages, as well as an interface for implementing new group keying algorithms;
- A practical identity management scheme that decouples application-level identities from their cryptographic counterparts;
- A comprehensive key management approach that covers the entire life-cycle of cryptographic materials;
- A general approach of relying on the application's communication channels for message transmission, relieving the application developer from the need to implement or integrate with a specialized, security-related communications subsystem;
- A number of communication optimizations, such as caching of long-term information on disk, using short-hand references for long identifiers, and compactly representing membership; and
- A set of configuration defaults to provide out of the box security and a practical approach for configuration interoperability.

The solutions described in this paper have been tempered by securing a widely-used application, group text chat. Doing so helped make details concrete, such as the best way to structure the rekey API or identify bottlenecks. Most recently, the general-purpose nature of these solutions has been put to the test by a group of developers using GROK with a different application: securing video broadcast in a mobile, wireless network. Based on their experiences, the solutions described in this paper are more generally useful to designers of secure systems.

This work is sponsored by the Department of Defense under Air Force contract FA8721-05-C-0002. Opinions, interpretations, conclusions, and recommendations are those of the authors and are not necessarily endorsed by the United States Government.

We continue our discussion with an overview of the GROK system, its design principles, and the API in Section II. Section III discusses how GROK implements identity, and Section IV examines how it manages keying material, including rekey algorithms, communications, and storage. Section V reviews how GROK synchronizes cryptographic material among group members. Section VI discusses the GROK code base and its application to securing group text chat. Section VII discusses related work, and Section VIII concludes.

II. System Overview

We designed a general-purpose building block to provide communications security among groups of users, i.e., *group security*. GROK's architecture, shown in Figure 1, consists of an application plugin and a standalone component. The components interact with one another as shown in Figure 2, to implement confidentiality, integrity, and authenticity on each message destined to the group.

All of a user's applications can interact with the same network accessible, standalone GROK process. Using the GROK API shown in Listing 1, applications can effectively reuse cryptographic material corresponding to identical user groups and access GROK even when it resides on remote machines. The standalone process isolates GROK's address space from direct access by applications, making it more secure. Reuse is safe because applications never have direct access to the material and because GROK uses key derivations to reuse secrets for different purposes.

GROK relies on applications' communication channels and a state synchronization framework called Authenticated Statement Exchange (ASE) [5] to synchronize cryptographic material among processes. It also relies on the application to manage group membership. Doing so allows it to remain a general-purpose cryptographic building block, unbound to a particular communications subsystem or layer in the network protocol stack. When necessary, however, GROK can communicate directly with certificate authorities (CA) to verify CA-signed data using online certificate status protocol (OCSP) [6].

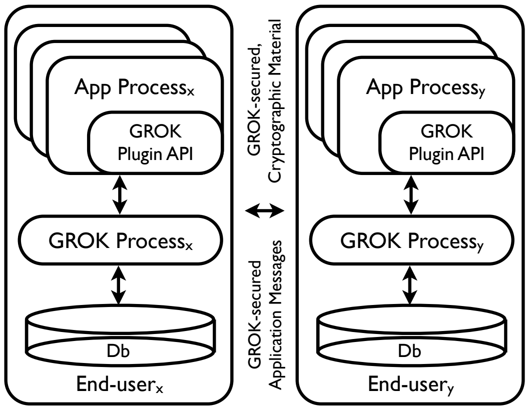


Fig. 1. The GROK architecture. An application uses the GROK plugin API to interact with the GROK process, which performs cryptographic operations to secure data. Each end-user runs a standalone, network-accessible GROK process, and each process has a database to store configuration information and accumulated cryptographic material. The material collects over time during peer interactions and includes keys used to protect group communications, application and cryptographic identities, and any material received from a central authority.

A. Design Principles

The target operating environment of the system depends on the application it secures. Consequently, we rely on GROK's configurability to adapt it for different applications. In addition, we tailored its design to include the properties of usability, efficiency, and security. We outline these properties in the context of GROK.

1) Usability: In part, the security provided by GROK relies on its usability from the viewpoints of administrators, end-users, and developers. An unusable system, by definition, provides little or no security to these parties. Administrators won't setup and deploy it, users disable inconvenient security, and developers won't use it. So, we consider each perspective in turn.

System administrators configure applications to implement policy and meet the needs of their user base and operating environment. Usable systems provide administrators with options that allow them to achieve these goals. Administrators can tweak GROK's default configuration to achieve many desired effects. For example, policy might dictate that all secured messages use sender-authenticity, or that it should function in a stateless mode, where every message relies on

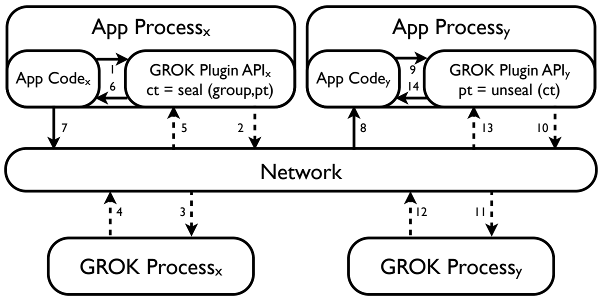


Fig. 2. A representative, GROK-secured interaction between *App Process$_x$* and *App Process$_y$*. The interaction begins when *App Code$_x$* calls *GROK Plugin API$_x$* to encrypt and integrity-protect, i.e., *seal*, a plaintext message (1). Afterwards, GROK passes plaintext through a secured channel (2) to *GROK Process$_x$* to generate ciphertext (3). When the application and GROK processes are colocated on the same machine, these communications occur over a loopback interface. *GROK Process$_x$* returns ciphertext through a secured channel (4) to the application (5), and *GROK Plugin API$_x$* returns ciphertext to *App Code$_x$* (6) for any further processing and distribution to peers (7). On receipt (8), *App Code$_y$* calls *GROK Plugin API$_y$* (9) to decrypt and validate, i.e., *unseal*, ciphertext. GROK passes it through a secured channel (10) to *GROK Process$_y$* to generate plaintext (11). *GROK Process$_y$* returns plaintext through a secured channel (12) to the application (13). Finally, *GROK Plugin API$_y$* passes plaintext messages back to *App Code$_y$* (14).

a new key for message security. Policy might require GROK to escrow keying material after it expires, or to periodically review certificate revocation lists for expired and revoked keying material.

Applications often rely on human-meaningful identities, like “alice,” to represent users. GROK instead must rely on cryptographic identities like public/private key pairs to provide protections. To reconcile these identities and realize usable security, GROK interfaces with the application using a human-meaningful, application identifier, such as “alice,” and maintains an internal mapping between the two. The user remains isolated from the underlying details. Figure 3 depicts the relationship of human-meaningful and cryptographic identities, and Section III describes how GROK handles identities in general.

To the developer, an API expresses usability in the form of simplicity and extensibility. GROK implements secure group communications among sets of users via a core four-function API, shown in Listing 1. Just as with plaintext messages, the application is responsible for communicating ciphertext messages after they’re generated. Additional housekeeping exists to initialize, configure, and dispose of programmatic resources; these are not shown here. The simplicity of the core API and GROK’s reliance on the application to communicate ciphertext messages enables developers to secure existing applications with little effort.

```
  group_add (group, user)
  group_remove (group, user)
  ciphertext = seal (group, plaintext)
  plaintext = unseal (db, ciphertext)
```

Listing 1. GROK API. Add a user to a group using *group_add*. The application manages membership and updates the secure group accordingly. Remove a user from a group using *group_remove*. Generate ciphertext from plaintext for a specific group using *seal*. Internally, *group* maintains a database pointer and a pointer to the configured rekey mode. The application is responsible for communicating the sealed message. Given the user’s db of existing keying material, use *unseal* to generate plaintext from ciphertext. The group may not be known during unseal and thus, is not passed as an argument.

Rekey modes implement the means by which GROK disseminates new keying material to groups of users. Operational needs, or the existence of new cryptography, might motivate creation of a new mode. A developer can extend GROK’s suite of rekey modes by implementing the interface presented in Listing 2.

```
wrapped = wrap (group, key)
unwrapped = unwrap_next (iterator,
                         ciphertext)
```

Listing 2. Rekey interface. A rekey-mode developer implements *wrap* to seal a key to a group according to their algorithm. The developer implements *unwrap* to return the next unwrapped key from the ciphertext using an iterator idiom. Each message can contain multiple wrappings of a key, and the idiom provides a natural mechanism for accessing them. In addition, the iterator maintains a database pointer to allow the rekey mode to perform database lookups.

GROK leaves any supporting functionality beyond these APIs, e.g., a graphical presentation of security, for the application developer to implement. Our secured text chat application (Figure 4) presents a good example of a graphical presentation by using unobtrusive icons to differentiate between unsecured and secured messages.

2) Efficiency: GROK defines efficiency according to the performance requirements of the applications it secures. Its design accounts for efficiency in message size, key transport modes, and other configurable options. These details can be tuned to reduce its impact on application performance.

In general, the details adapt GROK’s communications efficiency and the extent to which it relies on stored cryptographic material. With today’s technology, a greater reliance on stored material is, at a minimum, cost effective. In our testing environments [7], storage is three to six orders of magnitude less expensive per byte than communications. Communications links such as Iridium and Inmarsat [8], can cost greater than \$10 per minute for data rates of 2.4-128 Kbps, and disk storage costs less than \$100 per terabyte.

3) Security: Many applications implement trade-offs between usability, efficiency, and security. We designed GROK to support trade-offs using configuration options, and include a set of default values, e.g., Advanced Encryption Standard (AES) 256 cipher block chaining cipher mode [9] and Secure Hash Algorithm (SHA) 256 [10], to help ensure it is used correctly and securely out of the box.

The threat model considered by application users conditions the configuration. Here we outline the threat environment considered in GROK’s design and implementation.

- **End-user workstation** GROK runs in its own process, and we assume non-root processes do not have access to its address space or coredump files.
- **Persistent storage** GROK uses password protection to prevent unauthorized access to the user’s persistently stored, cryptographic material. We assume nobody other than the password owner can access the material.
- **Network** Any user has read/write access to the network. We assume users can intercept packets on the network, modify them, store them, and inject them at a later time. Users can also duplicate, drop, and fabricate packets.
- **Current group members** Users are “honest but curious.” They do not actively seek to thwart GROK by sharing symmetric keys, unencrypted material, or when an application requires membership secrecy, membership information.

III. Identification Management

GROK splits identity into the three components shown in Figure 3 to minimize the impact of security on application usability. Application users and developers rely on normal application identities called *uid*s while interacting with the application and interfacing with GROK. The string “alice” represents a sample *uid* or User ID. GROK relies on Cryptographic Identities called *cid*s when securing messages, and it translates between *uid*s and *cid*s using User-Cryptographic Bindings called *ucb*s. The split simultaneously allows GROK

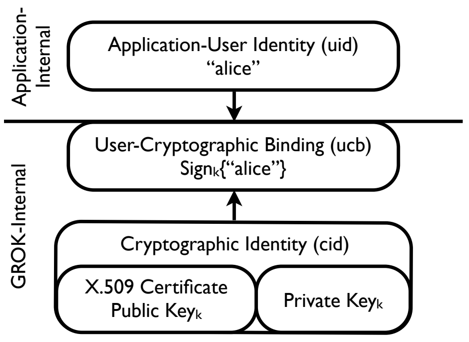


Fig. 3. Identification architecture. GROK represents each user with a public/private key pair. It creates a unique name for the pair by hashing the public key to form the *cid*. The name equips GROK with a unique, communications-efficient label for the underlying value, which it uses when referencing the pair to remote GROK processes. GROK creates a *ucb* by signing the user's *uid* with the private key associated with the *cid*, and stores each identity component (including the signature) in its database. Storing the signature allows GROK to cryptographically verify the *ucb* at any time in the future.

and the application to use identifiers suitable for their individual needs, decouples their maintenance, and permits security based on useful, application-specific identifiers.

GROK synchronizes identifiers and bindings among peers using the ASE framework, described in Section V, and application communication channels. It uses this information to address each secured message to a well-defined, unambiguous set of *cid*s according to an application-specified set of *uid*s, i.e., a *set membership*.[1] GROK also stores this membership in its database.

The identification architecture also allows message recipients to authenticate and check the integrity of sealed messages. To create sender-authentic messages, GROK digitally signs encrypted messages with private keys bound to *uids*. In general, these private keys are only used for signing data.

A. Lifetime Management

Information typically has an expiration time, after which its value diminishes or expires. We call the period before expiration a *lifetime*. Different data can possess different lifetimes. Moreover, in systems where remote end-points synchronize and cache data with differing lifetimes, classifying different data accordingly can improve communications efficiency—systems can synchronize information having a long lifetime less frequently than that having a short one.

We incorporate this concept into GROK to improve efficiency in constrained communications environments. *uid* and *cid* identity components can change independently from one another, and independent of material unrelated to identity. A user might refresh a cryptographic identity every 1-2 years, yet never change their *uid*. GROK synchronizes new values using ASE and then updates related set memberships stored in the database.

B. Identity Reuse

GROK supports *ucb* reuse across applications to minimize the amount of cryptographic- and/or binding-related material exchanged by GROK and to support *uid* reuse across applications. For example, text chat could reuse identities exchanged in email.

GROK maintains a (ucb, aid) pair for each *ucb*, where *aid* is an Application Identifier, to support reuse and help manage continuity among applications. An application name like "Pidgin" represents a typical *aid*. Each GROK instance maintains these pairs locally in its database.

C. Shorthand Identifiers

To reduce protocol message size and decrease data structure lookup time, GROK represents public keys and secret, symmetric keys with shorthand, public identifiers. Shorthand identifiers reference longer counterparts within local data structures and derive from a prefix of their hashed, longer counterparts. As an example, the prefix of a *cid* called a Member Identifier (*mid*) represents a user's cryptographic identity.

IV. Key Management

Managing secrets comprises the bulk of GROK's functionality and includes their generation, derivation, storage, configuration agreement, synchronization among set members, usage, maintenance, auditing, and destruction when no longer needed.

The secrets, called *set keys*, protect communications among *set* members. We define a *set key* as a piece of cryptographic material (a key) which is shared by a specific set of users, and is known to no one else. We define a *cryptographic set*, or *set* for short, as a tuple that includes a static membership and a key, and differentiate it from the term *group* in that a group's membership changes over time.

Next, we discuss GROK's approach to key management in more detail.

A. Handling Keys

GROK can generate symmetric keys using a variety of techniques. For example, it can use its underlying cryptographic library and random number generator (RNG), or local cryptographic hardware. By default, GROK relies on its underlying cryptographic library to generate keying material, and the library uses the operating system's random number generator.

GROK implements key diversity to reduce an adversary's ability to compromise any specific key and to reduce the fallout from any specific key compromise. Functionally separate keys remain cryptographically separate from one another.

A set identifier, included unencrypted in each sealed message, notifies message recipients which key was used to seal the message, so the recipient can more efficiently find the correct value to unseal the message.

[1]This does not preclude hierarchical arrangements of users where, for example, one *uid* might correspond to a user who redistributes messages to users unknown to the original sender.

To promote communications efficiency, GROK stores synchronized cryptographic material in a database. The database provides standard atomic, consistent, isolated, and durable (ACID) properties, which help ensure a consistent store despite possible hardware and software failures. It also provides a natural data structure for retrieving material. For example, a database can readily store set keys, or *ucb*-based translations between *uid* and *cid*.

Key derivations based on a user password encrypt and protect integrity of the database file and each individual secret it stores. The database stores no plaintext secrets and its protection scheme allows a user to change his/her password without re-encrypting the entire database file or re-wrapping all stored secrets.

GROK enables users to share keying material with one another in the form of rekey messages. Each rekey message relies on a rekey mode to wrap a set key to set members in such a way that it can be unwrapped only by members of the set and no one else. When set membership changes to a previously unseen set, the system creates a new set key, prepares a rekey message, and returns the message to the application for distribution. The new, wrapped set key corresponds to the new set membership.

GROK's design includes an interface to create, parse, and otherwise manage rekey messages. To support a new mode, the developer simply implements the interface, as shown in Listing 2, and appends the mode-specific database schema to the GROK schema. Each rekey mode manages its own configuration and state, but can access and use configuration from GROK and existing rekey modes.

GROK currently implements two novel rekey modes [4]. In each, the system constructs a rekey message according to a desired target membership.

In the first mode, called Dynamic Set Key Encryption (DSKE), GROK generates a new secret for the target set and wraps it to each set member using existing set keys. The result of a greedy set covering over existing sets defines which existing set keys (including pairwise) to use as wrapping keys. Only members that share a set key with the sender are included in the target set.

A second rekey mode, Boneh-Gentry-Waters set key encryption (BSKE) [4], relies on a cryptographic primitive defined in [11], to create a key for a new target set. Each user has a public identifier, a public key vector, and a private key. The Boneh-Gentry-Waters algorithm, in combination with this material, generates a new set key. BSKE mode provides a cryptographically secure binding of a set key to a set membership.

To support the reuse of keys generated by others, a rekey message carries encrypted membership information. A hash of the sorted list of member public keys forms the representation. Message recipients compute a similar compact value based on their current notion of set membership. If the two values match, the recipient associates the membership with the set key and uses the secret to secure future group communications. For a given set, this technique allows members to reuse keys generated by others instead of distributing a separate rekey message for each set. If policy allows this type of reuse, GROK can function more efficiently.

GROK incorporates protections in sealed messages to prevent surreptitious message modifications and sender impersonation. Principal protections exist in the form of sender- and set-authenticity, and a successful validation in either mode functions as a positive integrity check on the message. In sender-authentic mode, a sender digitally signs the message and the receiver verifies authenticity using the sender's public key. In set-authentic mode, a sender computes an HMAC over the message using an integrity key derived from a set key.

Careful key maintenance helps minimize the effect of key compromise by reducing the period of time or volume of data protected by a given key. GROK's design supports many standard key maintenance tasks, including versioning, rollover, revocation, expiration, and auditing.

In versioning, the system changes a key version by updating the set key version number in future messages. In rollover, the system replaces expired keying material with fresh material. The database is purged of expired cryptographic material and any related sets. Similarly, material is purged after revocation of an identity or key compromise. When audit trails are maintained, the system first wraps expired material with an auditor's public key, and stores the wrapped value in the database. It then purges expired material.

Using long-term secrets to protect messages allows an adversary to unencrypt past messages in the future if long-term secrets become compromised and if, in the past, the adversary captured messages secured with them. GROK prevents this scenario by implementing perfect forward secrecy [12]. No long-term secrets encrypt secured communications.

Finally, GROK is not responsible for determining set membership—only for enforcing it. GROK provides authorization by managing who can decrypt rekey messages, and by extension, future normal messages. Authorization rests on the fact that GROK seals each application message to a specific membership set using a key known only to that membership. The membership is determined by the application using the *group_add* and *group_remove* API from Listing 1. If a rekey message excludes a particular member, the excluded member does not have access to the rekey message, or any related, future messages.

B. Configuration Agreement

Before a cryptographic protocol can begin securing communications, protocol endpoints need to agree on which algorithms and configuration settings will underlie their security. To achieve usable, efficient security in the targeted environments, GROK forgoes multi-phase, cryptographic parameter and configuration agreement. Instead, each message includes a global configuration identifier defined as the prefix of a hash of the configuration values, e.g., $cfgid = Prefix(SHA256(cipherID, ...))$. Recipients ignore messages that contain identifiers they do not understand, and to follow an accepted practice [13], users do not determine others' configurations for securing data. Policy ultimately dictates which values fulfill the requirements of a given application, and finally, supporting new configurations equates to recognizing new identifiers.

C. Protection Mechanisms

If GROK doesn't adequately protect its cryptographic artifacts, no amount of clever algorithms will matter. Accordingly, GROK's design and implementation include many mechanisms to protect secrets, whether stored on disk, in memory, or transmitted in rekey messages. Protections exist in the form of cryptographic and software mechanisms.

Even with perfect protocols, software errors can undermine security. Therefore, we use several mechanisms to improve code safety. User input checks help protect the system from malicious inputs. Assertion checks at function calls alert of fatal, invalid conditions. Error handling within each function relies on factored cleanup code at the end of each function to unroll state modifications on error. As one example, the unroll protections help prevent adversaries from filling a user's memory with cryptographic material when unseal operations fail.

We took care to minimize the volume of plaintext secrets residing in memory at any given time. The internal data structures use hashed secret values as references instead of plaintext secret values. GROK securely cleanses memory after allocating secret data and before freeing it. Key diversity reduces the type of secret protected by any given database key-wrapping secret. Buffers of secret data are locked in memory and therefore, never paged to disk. Finally, reference counted plaintext values minimize memory-resident copies.

In the normal case, applications do not directly access secret material. The API enables developers to manipulate group membership and seal and unseal messages without passing plaintext secrets.

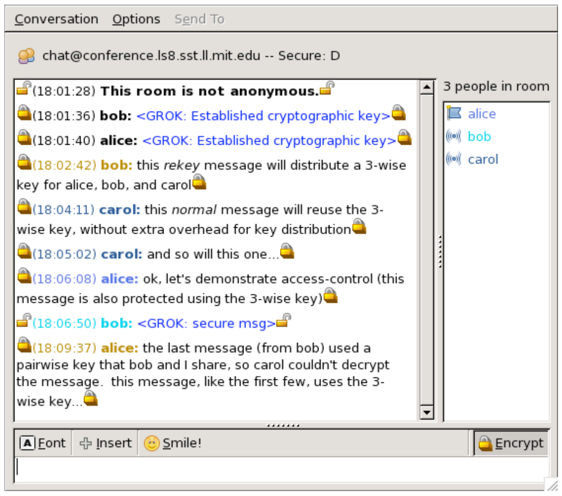
Rekey developers will access plaintext secrets as they wrap and unwrap set keys, and some applications will require similar access. Such applications might rely on GROK for key transport and storage, and directly use secrets within their system. To support them without jeopardizing other applications relying on the same set key, the API includes a function to export a set key derivation.

GROK runs in its own single-threaded process. Each application plugin (implemented using the GROK plugin API shown in Listing 1) transparently communicates with the GROK process through a TLS-secured channel.

As mentioned earlier, the database implements ACID properties. In addition, the entire database file and each stored secret remain encrypted on disk under the protection of the user's password.

V. Bootstrapping and Synchronization

A GROK subsystem, called Authenticated Statement Exchange (ASE), provides a mechanism for efficiently synchronizing state among set members [5]. GROK relies on ASE to



chat@conference.ls8.sst.ll.mit.edu -- Secure: D

(18:01:28) **This room is not anonymous.**

(18:01:36) **bob:** <GROK: Established cryptographic key>

(18:01:40) **alice:** <GROK: Established cryptographic key>

(18:02:42) **bob:** this *rekey* message will distribute a 3-wise key for alice, bob, and carol

(18:04:11) **carol:** this *normal* message will reuse the 3-wise key, without extra overhead for key distribution

(18:05:02) **carol:** and so will this one...

(18:06:08) **alice:** ok, let's demonstrate access-control (this message is also protected using the 3-wise key)

(18:06:50) **bob:** <GROK: secure msg>

(18:09:37) **alice:** the last message (from bob) used a pairwise key that bob and I share, so carol couldn't decrypt the message. this message, like the first few, uses the 3-wise key...

Fig. 4. An example Pidgin [14] window for a user *carol* in a secure chat with *alice* and *bob*. First, *bob* joins the room, which induces *bob* and *carol* to form pairwise keys. Next, *alice* joins and forms a pairwise key with *carol* (and *bob*, but you do not see that in *carol's* window). Then, *bob* transmits a secured rekey message containing a three-member set key corresponding to the membership $\{alice, bob, carol\}$—the UI does not display the cryptographic portion of the message. Next, *carol* and *alice* use the key to protect messages to the three of them. *bob* then excludes *carol* in a message to *alice*, and finally, *alice* answers using the three-member set key.

bootstrap secure communications by exchanging identities and other cryptographic material when members first encounter one another. For example, ASE can exchange signed DH pairs to form authenticated pairwise keys among set members. ASE also helps ensure that each set member receives rekey messages.

VI. Evaluation

To vet the design and implementation of GROK, we implemented a Pidgin [14] chat plugin that secures both instant messages and group chat. The C plugin included approximately 600 lines of crypto-related code for securing groups and 600 lines for bootstrapping and synchronization cryptographic material using ASE. Figure 4 depicts a chat session among users *alice*, *bob*, and *carol*.

In our test configuration, Pidgin clients connect to an Openfire server [15] and communicate using the XMPP protocol [16]. Using the GROK plugin, chat users communicate with one another in a secure way. Only the users chosen by the sender of the message, the current room members by default, can read messages. Neither servers nor network observers have access to plaintext messages. Users can selectively add and remove other users from the current secure group, independently of existing room membership. By default, as users join and leave, the plugin triggers GROK to change the set key used to secure messages. Past members cannot read messages sent to the current room and future members cannot read past messages sent to the room.

As Figure 4 shows, the only visible signs of security

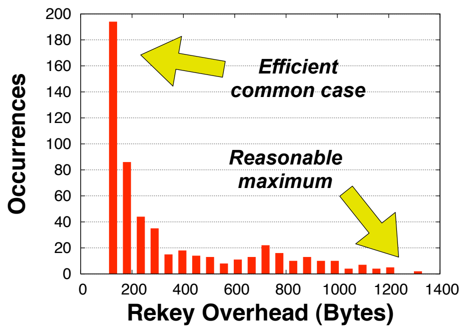


Fig. 5. Overhead of 554 total DSKE rekey messages during the CAPSTONE II exercise. In the common case, overhead was small (only two wrapped keys—one for the old group, and one for the new member), and in the worst case, it was less than 1400 bytes.

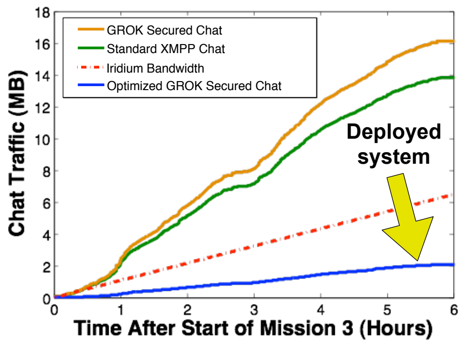


Fig. 6. System performance. During mission 3, the data volume generated by standard XMPP text chat, with or without GROK security, exceeds Iridium capacity—Iridium serves as a primary communications link for many military scenarios, which motivates our comparison with Iridium. The GROK Secure Chat line shows that rekey overhead and standard cryptographic message padding adds a small amount of overhead to Standard XMPP Chat. Our deployed text chat system shown as Optimized GROK Secured Chat in the figure, compressed XMPP messages and used less than Iridium's 2.4 Kbps link capacity.

in the UI are icons and occasional status messages. Lock icons highlight secured messages and unlock icons highlight unsecured messages.

A button, shown in the bottom right of Figure 4, allows users to disable encryption. It promotes the use of GROK because it allows secured users to briefly chat with unsecured users without disabling decryption or altogether disabling GROK.

We tested our secure chat plugin during a military exercise called CAPSTONE II [7]. The setup contained multiple geographically distributed sites, interconnected with intermittent, high-latency, low-bandwidth links. Each site had users that connected to a local server, and servers connected to one another. In each of six separate, six hour missions, approximately 30 users exercised the system from terrestrial and airborne platforms. During each mission, users primarily joined and left one main chat room and typed 2500 messages to one another. GROK exchanged cryptographic material as necessary and sealed each message to the current room membership. Figure 5 shows the nature and scope of rekey overhead generated as a result of membership changes. We used the DSKE rekey mode, and in the common case, the overhead was minimal—no more than two key wraps. According to users, GROK-secured text-chat functioned well using links with bandwidths of 2.4 Kbps and round-trip time delays of 3.5 seconds. Figure 6 supports this conclusion—it used only a fraction of the total link capacity available to users. See [17] for more details about the nature of the communication links. That work presents information about the same poor quality links, but in the context of a different exercise.

A. Securing Broadcast Video

A group of developers recently applied GROK to secure video broadcast from one node to many in a mobile, wireless network using less than 450 lines of crypto-related code. The new application tested GROK's general-purpose nature, and GROK's ability to operate with a different network model. Unlike the bi-directional, unicast links used in text chat, this network included one uni-directional, broadcast link. The new employment of GROK has been successful so far—they were able to secure video and handle membership changes with a minimal impact on video quality.

B. Software

The GROK libraries consist of ∼30K lines of C and ∼10K lines of C++ that has been tested on Linux and OS X. The code provides an API to secure set communications and another to implement new rekey modes. Each rekey mode exists as its own library. Approximately 560 lines of C implement greedy set-covering used in DSKE, and approximately 3800 lines of C implement BSKE. GROK relies on OpenSSL [18] to implement cryptography, PBC [19] for pairing-based cryptography in BSKE rekey mode, GMP [20] for bit operations and greedy set-covering, and Sqlite [21] for database management.

VII. Related Work

A handful of key management applications and protocols have similarities to aspects of GROK. Apple's Keychain [22] and Microsoft's CryptoAPI [23] provide a managed keystore to securely maintain application passwords and public/private key pairs. The TLS [24] protocol secures point-to-point client-server interactions. IPsec [25] secures IP datagrams, typically in a point-to-point setting, and an extension exists for securing multicast traffic [26]. S/MIME [27] secures email to an arbitrary number of users. A host of algorithms and architectures exist to secure group communications [28], [29], [30], [31], [32]. Multi-party off-the-record (mpOTR) messaging provides secure group communications [33].

VIII. Conclusions

We describe practical solutions to a number of problems in creating a general-purpose building block for securing group communications. These solutions were developed as part of the GROK system and were tempered by securing a widely-used text chat application. According to users during a multi-mission, military exercise, GROK performed well, and performance measurements support their conclusion.

To help make the system usable and inform users about the secured nature of their text chat messages, we implemented subtle interface cues. Interestingly, we implemented an interface option to disable encryption, which made the system more secure because it allowed users to easily re-enable security after disabling it temporarily. Before the option, some users disabled the GROK plugin altogether and left message security turned off.

GROK allowed users to chat securely over the Iridium link, which serves as an important communications channel in military scenarios. With a bandwidth of 2.4 Kbps and an average round-trip latency of 3.5 s, Iridium functions as a lower bound on link performance. Chatting securely and successfully in this environment was due in part to the behavior of DSKE rekey mode: most rekey events included only two wrapped keys in the message header. Because GROK performs well on Iridium, it can function effectively elsewhere.

In the future, we plan to develop new rekeying modes, apply GROK to securing IP multicast, use the Cryptographic Message Syntax [34] as a message packing format, and implement more design features. In our future work, we will continue addressing practical system details and focusing on usability, efficiency, and security, as these are critical for useful systems.

IX. Acknowledgements

We thank Stephen Okano, Sean O'Melia, Adam Petcher, Dan Utin, and the Paul Revere team for helping to improve GROK, and we thank our sponsors who provided many practical motivations for the problem of secure group communications and broad support for our solution.

References

[1] S. Rafaeli and D. Hutchison, "A Survey of Key Management for Secure Group Communication," *ACM Computing Surveys*, vol. 35, no. 3, pp. 309–329, 2003.

[2] R. J. Anderson, *Security Engineering: A Guide to Building Dependable Distributed Systems*. Wiley Publishing, 2008.

[3] B. Schneier, *Applied Cryptography (2nd ed.): Protocols, Algorithms, and Source Code in C*. New York, NY, USA: John Wiley & Sons, Inc., 1995.

[4] R. Figueiredo, "Securing group communication in dynamic, disadvantaged networks : implementation of an elliptic-curve pairing-based cryptography library," Master's thesis, Massachusetts Institute of Technology, Dept. of Electrical Engineering and Computer Science, 2006. [Online]. Available: http://hdl.handle.net/1721.1/41602

[5] B. W. Fuller, R. I. Khazan, J. A. Cooley, G. E. Pickard, and D. Utin, "ASE: Authenticated Statement Exchange," *9th IEEE International Symposium on Network Computing and Applications*, 2010.

[6] M. Myers, R. Ankney, A. Malpani, S. Galperin, and C. Adams, "X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP," RFC 2560 (Proposed Standard), Jun. 1999. [Online]. Available: http://www.ietf.org/rfc/rfc2560.txt

[7] "MILSTAR." [Online]. Available: http://www.lockheedmartin.com/products/Milstar/index.html

[8] "EMS SATCOM." [Online]. Available: http://www.emssatcom.com

[9] *FIPS PUB 197: Advanced Encryption Standard (AES)*. Gaithersburg, MD, USA: National Institute for Standards and Technology, November 2001. [Online]. Available: http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf

[10] National Institute of Standards and Technology, *FIPS PUB 180-3: Secure Hash Standard*. Gaithersburg, MD, USA: National Institute for Standards and Technology, April 2008. [Online]. Available: http://csrc.nist.gov/publications/fips/fips180-3/fips180-3_final.pdf

[11] D. Boneh, C. Gentry, and B. Waters, "Collusion Resistant Broadcast Encryption With Short Ciphertexts and Private Keys," in *Crypto*, 2005, pp. 258–275. [Online]. Available: http://www.truststc.org/pubs/605.html

[12] W. Diffie, P. C. V. Oorschot, and M. J. Wiener, "Authentication and Authenticated Key Exchanges," 1992.

[13] C. Kaufman, R. Perlman, and M. Speciner, *Network Security: PRIVATE Communication in a PUBLIC World*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2002.

[14] "Pidgin Chat Client." [Online]. Available: http://www.pidgin.im/

[15] "Openfire." [Online]. Available: http://www.igniterealtime.org/projects/openfire/index.jsp

[16] P. Saint-Andre, "Extensible Messaging and Presence Protocol (XMPP): Core," RFC 3920 (Proposed Standard), Oct. 2004. [Online]. Available: http://www.ietf.org/rfc/rfc3920.txt

[17] J. Cooley, O. Huang, L. Veytser, and S. McGarry, "Mobile Airborne Networking Experience with Paul Revere," in *Military Communications Conference, 2005. MILCOM 2005. IEEE*, 2005, pp. 2882–2888 Vol. 5.

[18] "The OpenSSL Project." [Online]. Available: http://www.openssl.org/

[19] B. Lynn, "PBC: The Pairing-Based Cryptography Library." [Online]. Available: http://crypto.stanford.edu/pbc/

[20] "GMP: The GNU Multiprecision Arithmetic Library." [Online]. Available: http://gmplib.org/

[21] D. R. Hipp, "Sqlite." [Online]. Available: http://www.sqlite.org/

[22] "Apple Keychain Services Programming Guide," March 12, 2009. [Online]. Available: http://developer.apple.com/documentation/Security/Conceptual/keychainServConcepts/keychainServConcepts.pdf

[23] "Microsoft Developer Network Cryptographic Functions." [Online]. Available: http://msdn.microsoft.com/en-us/library/aa380252(VS.85).aspx

[24] T. Dierks and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2," RFC 5246 (Proposed Standard), Aug. 2008. [Online]. Available: http://www.ietf.org/rfc/rfc5246.txt

[25] S. Kent and K. Seo, "Security Architecture for the Internet Protocol," RFC 4301 (Proposed Standard), Dec. 2005. [Online]. Available: http://www.ietf.org/rfc/rfc4301.txt

[26] B. Weis, G. Gross, and D. Ignjatic, "Multicast Extensions to the Security Architecture for the Internet Protocol," RFC 5374 (Proposed Standard), Nov. 2008. [Online]. Available: http://www.ietf.org/rfc/rfc5374.txt

[27] B. Ramsdell, "Secure/Multipurpose Internet Mail Extensions (S/MIME) Version 3.1 Message Specification," RFC 3851 (Proposed Standard), Jul. 2004. [Online]. Available: http://www.ietf.org/rfc/rfc3851.txt

[28] D. Wallner, E. Harder, and R. Agee, "Key Management for Multicast: Issues and Architectures," RFC 2627 (Informational), Jun. 1999. [Online]. Available: http://www.ietf.org/rfc/rfc2627.txt

[29] T. Hardjono and B. Weis, "The Multicast Group Security Architecture," RFC 3740 (Informational), Mar. 2004. [Online]. Available: http://www.ietf.org/rfc/rfc3740.txt

[30] H. Harney, U. Meth, A. Colegrove, and G. Gross, "GSAKMP: Group Secure Association Key Management Protocol," RFC 4535 (Proposed Standard), Jun. 2006. [Online]. Available: http://www.ietf.org/rfc/rfc4535.txt

[31] M. Baugher, R. Canetti, L. Dondeti, and F. Lindholm, "Multicast Security (MSEC) Group Key Management Architecture," RFC 4046 (Informational), Apr. 2005. [Online]. Available: http://www.ietf.org/rfc/rfc4046.txt

[32] A. T. Sherman and D. A. McGrew, "Key Establishment in Large Dynamic Groups Using One-Way Function Trees," *IEEE Transactions on Software Engineering*, vol. 29, no. 5, pp. 444–458, 2003.

[33] I. Goldberg, B. Ustaoğlu, M. D. Van Gundy, and H. Chen, "Multi-party off-the-record messaging," in *CCS '09: Proceedings of the 16th ACM conference on Computer and communications security*. New York, NY, USA: ACM, 2009, pp. 358–368.

[34] R. Housley, "Cryptographic Message Syntax (CMS)," RFC 3852 (Proposed Standard), Jul. 2004, updated by RFCs 4853, 5083. [Online]. Available: http://www.ietf.org/rfc/rfc3852.txt