CS 595: Cryptography
Final Project

Tim Wylie

December 7, 2009

Project Overview

I have implemented a basic covert multi-party communication instant messaging program. The users can communicate with any other user, but communication is a two-way channel only. The program makes use of the concept of covert two-party computation[1] and steganographic[4] methods of communication if the SFE (secure function evaluation of the computation) has returned true for both parties.

The implementation itself took quite a while, and therefore there is a section covering the basic usage of the application. There are also sections describing issues with the implementation and future developments if the application were to be continued to be worked on for daily use.

Cryptographic Backend

As in standard two-party computation, Alice and Bob have secret inputs x_A and x_B and they desire to compute a function $f(x_A, x_B)$, the difference is that neither Alice nor Bob know if the other person wants to compute the function, and they won't find out unless both choose to run the computation and the outcome is true. The function f in my program is simply

$$f(x_A, x_B) = \begin{cases} 1, & \text{if } x_A = x_B \\ 0, & \text{if } x_A \neq x_B \end{cases}$$

This function could be easily extended to calculate some group public key given two private keys or something similar that is secure other than just comparing the group name input.

This is a binding commitment scheme as well since once a "check" is issued on the client's side, it cannot be changed. There is, however, the ability to check multiple groups against the same user. Ideally, there should be a guarantee that the user is actually a memeber of that group via some key scheme as mentioned above. This might seem to imply that if you check a user against all groups, you gain some knowledge about that user if only one group check returns true. This is not the case though. Alice may check all of her groups against Bob, but Bob may only check one of his against Alice. Also, a communication is only allowed and the client notified the result was true if the client has his group set to that. So if Bob checks group 1,2,3 against Alice, and Alice checks group 1,2,3 against Bob the result would be True for all three groups, but if Bob has his group set to 1 and Alice has hers on 3, they will still not be able to communicate because they have essentially changed x_A and x_B after a submission to $f(x_A, x_B)$. Basically, the program checks that the output of $f(x_A, x_B)$ is contingent on x_A and x_B.

This sort of contingency was not mentioned in the paper[1], but is a necessity when dealing with two-way covert computation where several computations are attempted at once. In this example, should Alice change her group to 1, then they would be notified and the computation would have been a success.

The idea of using steganography is well known for secretly sending data, and has been proven that it can be done securely[4]. The encryption algorithm used in this application is ARC2, and then the protocol described in RFC1751[5] is used in order to translate the bytes into readable ASCII characters before they're hidden within the image. The reason for this translation is because the python steganography library that I used only accepts ASCII characters to be encoded. The key is simply the group name which is not the most secure, but given that you can only communicate should both parties be members of the same group covertly, it is sufficient for this application.

Application

Overview

In order to run the application, the server must first be running.
$python covert_server.py
Then as many clients as desired can be started.
$python covert_client.py

The user connects by starting the "Connect" dialog under the "File" menu. Although there is the option for changing the port, it is hardcoded in the server, so it shouldn't be changed. The available users are listed at the bottom of the server(this issue is mentioned in the "Issues" section). They are user1, user2, user3, and user4 with passwords a, b, c, and d respectively. The avatar is optional, but if chosen will show up in the sidebar along with the user names.

In order to send a message the client must select another user in the side bar and then a message or emoticon can be sent. GUI emoticons must be added via the "Send" menu.

By default there are no groups. A client must add a group via the "Add Group" dialog. The new group is not selected by default though and the client has to change to it under the "Group" menu. In order to be covert with another user, the client should have his new group selected, highlight another user, then click "Check User" under the "Group" menu. If the other user similarly checks that group the status bar will tell you that you are covert with them. If the client is covert with another user and sends an emoticon, an "Embed" dialog will pop up where he can type into the text box and click "OK". That message will then be encrypted, translated, and hidden within the emoticon. The other user will receive the emoticon and a message box will pop up with the secret message.

Technologies

The application is written in python, and other technologies were chosen based on this. The Twisted[6] network framework was used for the network communication. This allows for single-threaded high-volume asynchronous networking. Within the Twisted framework, Perspective Broker was used for communication and authentication. This allows for the client to call remote functions on the server as if they were local. This gets rid of the need for defining a networking protocol and parsing incoming strings. The authentication is also secure and hidden during this process.

The steganography library used was Stepic[8]. It is written in python

and can be downloaded and imported into the project without installing. The library is very easy to use, but has one drawback: there is no native encryption or passwords used. Anyone with the image can call the decode function and get the results. This means that an adversary could simply try to decode every image until he found one that returned without an error. This is why separate encryption algorithms were also used. If someone does decode the image, the will not be able to decode the message.

The python libarary that was used for the encryption was the Python Cryptography Toolkit[7]. Specifically, the ARC2 encryption algorithm was used. This was chosen because it allows variable length keys and was rather easy to include. As mentioned above the RFC1751 protocol was also used from PyCrypto in order to translate the bits into ASCII so that Stepic could encode the text.

Dependencies

- Python interpreter
- Twisted Framework - http://www.twistedmatrix.com
- The Python Cryptography Toolkit - http://www.dlitz.net/software/pycrypto/
- Stepic Library - http://domnit.org/stepic/doc/
- pyGTK - This is the cross-platform GTK GUI toolkit with python bindings.

Issues

Several shortcuts were taken since this was merely a proof-of-concept application. None of them hinder the functionality, but before any sort of extreme use these should be addressed.

The clients that log in must be authenticated, however, since I did not want to tie the application to a database an in-memory user list is used. This means that any new users must be added to the server before the server is started. The users are declared at the bottom of the server script.

Emoticons are sent by themselves rather than inline with other text. This has to do with parsing out images within a text buffer in order to display them. You have to search through a text buffer and find certain bytes, then copy the image data out and insert it as a pixbuf rather than text. Due simply to the amount of time already put into the application, this issue was simply not worth solving.

When emoticons are sent, they no longer have transparency, but instead have a white background. The steganography library Stepic makes no distinction between images with or without transparency, so when it encodes a message within an image with transparency the output image is completely scrambled and looks like noise. Since we're hiding data this is unacceptable. I kept the transparent ones within the menu since they look nice, and ideally a library that can handle them should be used, but I did not find one with python bindings and did not want to waste anymore time finding one.

As mentioned above, Stepic returns the text if it finds any. Ideally a library would be used that would only return text given some encryption and passphrase used for the data hiding. This way an adversary would have no way of knowing whether or not information was hidden at all.

The selected user becomes unhighlighted occasionally. This is because every 20 seconds the client checks to get the user list. When the list is reloaded, the application does not save which user was selected. This was again not fixed due to time constraints.

Occasionally, when you check a user for a group that has previously been covert with the checker, the user automatically gains covert status with the other user. This is simply because sometimes when a user logs out and logs back in later, the records were not correctly deleted on the server. This persistence shouldn't happen, but occasionally does.

Hitting "Enter" doesn't send the message. It should, but I did not implement that or other common shortcuts.

The password text field is not obscured. This is simply an option I didn't see when I was building the dialog. This should be an easy fix with an understanding of gtk development.

Future Developments

There are a few developments that could have also been issues, but I chose to put them here because they were intentional limited features. It is assumed that all items in the "Issues" section should also be addressed in future developments.

The encryption scheme should be selectable upon group creation. This would not have been that hard to implement, but I was overrun with the amount of stuff that needed to be implemented. The "Add Group" dialog should have a combo box that lets the client select which type of encryption scheme to use and lets him specify the key rather than just using the group name. This would also mean that the inputs x_A, x_B each consist of the group name, the encryption used, and the passphrase.

Public Key Encryption could also be easily added by encoding your public key within your avatar. Then for each covert message received, after it has been decoded from the image, the other user's avatar can be decoded to get their public key. This allows for a PK encryption scheme to be used without it being explicitly used from the perspective of other users. In fact, the PK can be constantly changing because a user can update their avatar and it will propogate seamlessly to all other users.

Making the application P2P would make it more relevant to being covert. This would not be that difficult overall, but again because of time, a simple client/server model was much faster to build. Having a P2P client would make it easy for a client to create his own function f(which is part of the next point).

Using another function for the covert computation would be interesting. Maybe something like the millionaire problem, where Alice only wants to be covert with those that have less money than her, but Bob only wants to be covert with users wealthier than him so he can be "in" with a better crowd. The paper mentions a dating application where the users "like" each other and similarly, in this application they then could send secret love messages between themselves if they both "like" each other. Maybe a situation where I want to get support like AA, but I don't want all of my friends to know that I need help. I can check for other users who are looking for a "secret" support group.

Usually, instant messaging application use text emoticons and replace them with the graphic rather than actually sending the graphic. This should probably work the same to avoid all the extra overhead. Maybe the application can simply allow sending picture files like over IM applications, but also give the user the embedded text option. This would be much less suspicious and more practical. Plus, if an adversary is watching the line, they could see the same emoticon several times with different minor bits. They would then know that the two users are using steganography in order to secretly communicate. If emoticons were used normally, but pictures were just being sent, an adversary would not have another version to compare the image to.

The server is very simplistic in handling messages and users. This was purposeful so that translating the program to a P2P structure would not be that difficult in the future. If the client/server model is going to continue to be used without a P2P structure though, a more advanced framework should be developed.

There is no ability to save conversations or to re-read a covert message by selecting an emoticon. This ability would be a nice feature, but in a secret conversation, it may also be good that it is not possible.

Conclusion

I believe that this software does a good job of providing a facility to do covert two-party computation while giving a useful application for the technology when combined with some other cryptographic tools. If the improvements were made that were suggested above, the application could be incredibly useful in a single package. Similar concepts could be employed other ways using other separate applications. If another application existed for the covert computation, the users could then log on to any instant messenger, use a separate tool to embed messages in an image, and send it to each other through the other service.

My software shows how these separate tools can be combined in a useful way that give it unique properties and applications the tools would not have had otherwise.

References

1. von Ahn, L., Hopper, N., and Langford, J. 2005. Covert two-party computation. In Proceedings of the Thirty-Seventh Annual ACM Symposium on theory of Computing (Baltimore, MD, USA, May 22 - 24, 2005). STOC '05. ACM, New York, NY, 513-522.
DOI= http://doi.acm.org/10.1145/1060590.1060668

2. Anguraj S., Balamurugan D. 2009. Implementation of Audio Steganography in Real-Time Protocol(RTP) and Hypothesis of RTP Features. 1st National Conference on Intelligent Electrical Systems (NCIES09), 24-25 April 2009, Maha College of Engineering, Salem, India.

3. Bayer, P., Widenfors, H. 2002. Information Hiding - Steganographic Content in Streaming Media. Masters Thesis (Blekinge Institue of Technology, Sweden).

4. Hopper, N., von Ahn, L., and Langford, J. 2009. Provably Secure Steganography. IEEE Trans. Comput. 58, 5 (May. 2009), 662-676.
DOI= http://dx.doi.org/10.1109/TC.2008.199

5. McDonald 1994. Human-Readable 128-bit Keys. RFC1751 (Dec. 1994)

6. Twisted Framework - http://www.twistedmatrix.com

7. The Python Cryptography Toolkit - http://www.dlitz.net/software/pycrypto/

8. Stepic Library - http://domnit.org/stepic/doc/

9. messagebox - http://danmarner.blogspot.com/2008/05/creating-message-box-with-pygtk-and.html