BlueRing

A Python-Based Encrypted Anonymous Communication Platform

Using the Tox Protocol

By

Kellen Hill

In Partial Fulfillment of a Bachelor's of Science in Computer Science

Department of Computer Science and Engineering Technology

University of Houston-Downtown

Faculty Advisor:

Dr. Shengli Yuan _____

Course Instructor:

Dr. Kenneth Oberhoff _____

Table of Contents

Abstract

The Tox protocol and ToxCore library provide a code base and API for building encrypted communication channels. The software program BlueRing implements this protocol in Python via the PyToxCore codebase. This research includes a quick overview of civilian cryptography, along with a more in-depth look at the modern landscape of encrypted messaging, some of the challenges inherent in encrypted communications, and the approaches taken by the Tox project to overcome these challenges and mitigate risks. In addition, it examines the progress and problems of the development of the BlueRing application.

Acknowledgments

This project would not have been possible without the direction of Dr. Shengli Yuan, who encouraged continued development through a multitudes of adversities, and the continuous endeavors of the developers of the Tox project. Their attention to detail and focus on multi-tiered privacy efforts inspire optimism in a world mired in data mining and marketing optimization. Further thanks are extended to Dr. Kenneth Oberhoff, who oversaw the progress reports and presentation of this project.

A Brief Look at Civilian Cryptography

The history of encryption efforts in computer science is littered with insecure weaknesses, complicated discussions, and numerous trade-offs. In the beginning, DES, developed at IBM, was intentionally weakened by the NSA (which forced a key length much smaller than the internal developers thought secure) (Levy, 2001). A few years afterwards, Whitfield Diffie and Martin Hellman

developed what would be called the Diffie-Hellman key exchange, which would be shown insecure via the Logjam exploit, with accusations that security agencies belonging to both the US and UK had previously discovered this weakness and had opted against disclosing it (Adrian et al, 2015, pg. 5). This pattern would continue well into the 21st century, with PGP creator Phil Zimmermann harassed over his program and later protocol (Goth, 2006). Most recently, the documents released by Edward Snowden outlined massive surveillance efforts by the United States and its "Five Eyes" allies of Australia, Canada, New Zealand, and the United Kingdom (Finlay, 2014).

Security discussions have become more important in the 21st century as news of government or corporate surveillance, data breaches, spyware infections, and persistent privacy threats have grabbed headlines repeatedly. One of the major arenas in which security has seen a resurgence of interest is personal communications. Conversations by law enforcement, security professionals, legislators, and the public have driven an interest in controlling what data should be available to whom, when, and under which circumstances. Concerns regarding both communication content and metadata (who contacted whom, when was the communication made, where was the communication made from or to, and how long did the communication last) mean that many tech companies have begun to implement end-to-end encryption; this method of communicating ensures that a message is encrypted before being sent, stays encrypted for each step in transit, and can only be decrypted by the receiver. One of the implications of this is that the central server facilitating the conversation is unable to read the contents of the messages, but still has access to the metadata for each interaction between users. Even then, the move towards end-to-end encryption has been a slow march, with Signal and WhatsApp providing end-to-end encryption within the last few years (2014 and 2016, respectively) (Nordrum, 2016), while the behemoth in the video-calling space, Skype, did not even begin testing support for the feature until January, 2018 (Newman, 2018).

All of these clients support communications that are encrypted, but they all also include a hefty drawback: the use of these services requires connecting to a central service that facilitates communications, leaving user metadata in the hands of the company running those servers[1], and putting users at risk of information-sharing with advertisers or government agencies, with that same data just waiting for a breach, leak, or subpoena. Researchers have only recently started attempting to determine the possible implications of widespread metadata collection, and data collected today could certainly be used in the future as new techniques are discovered (Schneier, 2014). A major goal of the Tox project was preventing this kind of information leakage (Finlay, 2014). A secondary problem is that each of these applications requires extra data about the user in order to provide service (e.g., the Signal service requires a phone number in order to initially set up the user's account), causing further privacy concerns. The Tox protocol was designed to bypass this pitfall, allowing end-to-end encrypted communications between users using text chat, voice, or video via peer-to-peer methods that don't require or use a centralized server architecture.

Risk Profiles

Most people do not need perfect security for communications; if they did, they certainly wouldn't check their phones in public, send postcards, or use HTTP. Most people tend to think about security when it's most obvious, like using a credit card to buy something from an e-commerce site or logging in to Facebook. In reality, each individual is constantly making risk-analysis decisions regarding their habits and communications, mostly involuntarily. For example: a bus full of people, with a handful having discussions on their phones; a typical (if not occasionally annoying) occurrence.

1 Signal has issued a press release stating their intention to incorporate "sealed sender" into their model (Lund, 2018); if implemented properly, it would encrypt the sender's identifier along with the message (only the receiver would know who sent it). How this will change risk profiles is outside the scope of this research due to the timing of this press release and its proximity to the research deadline.

In that moment, all are deciding, consciously or not, that the communication with the party on the other end of the line is worth the risk of a fellow passenger eavesdropping on their conversation. For a more extreme example, it would not be unheard of for an individual in an abusive relationship to keep their text alert on silent, for fear of incurring the wrath of their abuser; in this case, the metadata (who is receiving the text, and when) matters much more than the contents of that text message. In both of these examples, decisions are being made to balance usability against risk, with considerations to possible threats or risk likelihood. Each user should consider, carefully and intentionally, what matters to them and select a solution that meets their needs for both security and convenience.

Tox sacrifices some usability for security. Two factors here stand out: the the addressing system, and the use of the distributed hash table (DHT, which will be covered in more detail in a later section) for determining who is online. The way in which users check whether or not friends are online can result in delayed updates of a user's status for a few minutes when they first log in, as changes propagate through the DHT (this effect is even noticeable when two clients are being run from the same host). The real-world effects of this are likely minimal, as once two clients are able to find one another through the DHT, a connection between them is made and that connection is used for updates and communication; this is noteworthy, since centralized systems don't have this same pause between status updates. While this trait can be annoying, as no one wants to wait for their friend status to refresh, clients are usually set to ping their friends every sixty seconds. The use of onion-routing (discussed in the Advanced DHT/Onion Routing section) is meant to ensure that even the IP address of a user is concealed from anyone they are not friends with, but it adds significant complexity to the way in which the DHT operates ('Nurupo', 2018). In addition, this behavior also implies that it is not possible to have two Tox clients on a local machine find and communicate with one another unless that

local host is also configured to act as a bootstrap node (runs a specific kind of DHT software to facilitate peer connections, detailed in the DHT section).

In the address system, anonymity has been prioritized over usability. Few people would be excited at the prospect of using a 76-character hexadecimal string as an identifier; this particular hurdle can be bypassed with the use of a name-lookup service, but they are not universally used and incur further privacy trade-offs. One possible consideration is that the use of such services leaks data the same way a traditional centralized model might: anyone that runs the service (or a malicious actor that managed to gain access to logs) can see who is searching for whom, and when. Most importantly, however, is that the use of such name-lookup services must be used carefully, as it opens the user up to possible man-in-the-middle attacks – a user registers their Tox ID as a particular screen name, pretending to be someone else, then connects with their mark and relays all messages from the mark back to the individual they're pretending to be. This is a concern whenever you get contact info from anyone, especially online (Are they who they say they are? How do you know?), but it can be especially troublesome if the user thinks a security-centric application can prevent human exploitation.

The Tox Protocol – Quick Intro

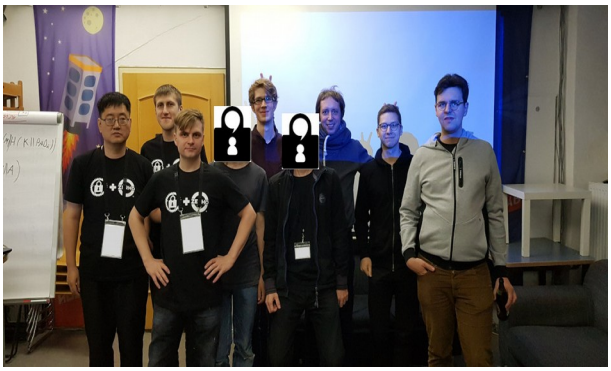
The birth of the Tox project is a direct result of the revelations of Edward Snowden (Tox Project, 2017). The documents Snowden provided outline widespread government surveillance, and it became increasingly obvious that surveillance concerns have justifiably grown. These worries have only been compounded as the weakness of centralized services to political and legal pressure have become apparent (a problem that can be best exemplified by the widely-available Skype guidelines that delineate what information they can and will provide to law enforcement by subpoena). The Tox

project was the result of looking at the chat and videoconferencing systems available at the time, finding them lacking, and attempting to harden them against some of these threats.

The Tox project is a decentralized peer-to-peer messaging and videoconferencing protocol, but that description oversimplifies some of the very clever implementation details that turn the protocol from an interesting side project to a usable communication medium. Like all security-related topics, there is a discussion to be had regarding security versus usability, and which risk factors are addressed by the project contrasted with those that aren't.

The Tox protocol was first implemented in the ToxCore project by user irungentoo, then expanded with the TokTok project by a number of contributors; for the remainder of this document, Tox will be used to describe the protocol, while ToxCore will be used in reference to the code that implements the protocol (whether in the newer TokTok implementation, or the original ToxCore).

Active development is still being done on the project, with a developer conference occurring in November 2018. It is worth a mention here because their posts regarding the conference are a wonderful example of the security conscientiousness of the developers involved: the pictures they posted have some of their faces obscured, and each person has ensured their con badge has been turned over to show only the blank back of it instead of their name or handle.



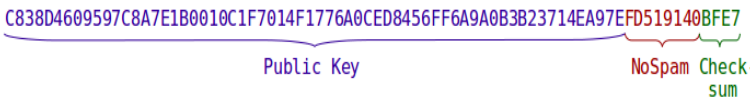
Devs at ToxCon, 2018

Tox Protocol Caveats

As mentioned previously, the Tox protocol and the ToxCore codebase are still in development. Though there are working, usable clients that are (relatively) stable and feature-rich (including full A/V support for videoconferencing), the ToxCore codebase is basically still in the beta-testing stage. The project has a list of features they hope to implement in future versions, and hopes to have further research performed on the security of the protocol or its implementation, et cetera. That being stated, the cryptographic library used by Tox (NaCl) has been subjected to a number of testing methodologies (Bernstein et al, 2012) and is a much more mature base to build upon. A developer summit was held on October 12-14th, 2018 (ToxCon 2018, in Vienna, Austria), but as of publishing this research, no papers or updates regarding topics covered have been made public.

Tox Addressing

Every profile for a Tox client includes a Tox ID; these are 76-digit hexadecimal numbers that uniquely identify the profile. The Tox ID is composed of three parts: a 64-digit public key, an 8-digit "nospam" value, and a 4-digit checksum. While the public key portion of the Tox ID is immutable (changing it requires generating a new profile), the nospam and checksum can change with user interaction.



Length	Contents
32	long term public key
4	nospam
2	checksum

The Fields of a Tox ID (in bytes)

For the Tox protocol, a central service is only used if an individual wants to look up the contact details of a registered user; even then, the presence of a user on that service is opt-in: the user must register their Tox ID to be included in a user directory and this action is completely optional. Email-style addresses can be tied to Tox IDs, allowing for a user to enter an address to retrieve the Tox ID attached to it (the most widely-used such lookup service is hosted at https://toxme.io). A deeper look into these services is available under the “Lookup Services” section.

As far as the namespace is concerned, 32 bytes to the public key portion of the address means that Tox supports a max of 1.16×10^{77} addresses.

Nospam Value and Checksum

The second and third fields of the address are the “nospam” value and the address checksum; a more thorough explanation at how the nospam value is intended to function can be found in the ‘Friends and Messages’ section later in this text, but the information germane here is that it is a random 4-byte value appended to the user’s public key. The public key and the nospam value are then hashed to arrive at a 2-byte checksum for the address. This checksum is used to ensure that data sent to the DHT nodes are valid; if the checksum calculated by the server using the public key and nospam portions of the address does not match the checksum delivered, the data is dropped.

```
static uint16_t checksum(const uint8_t* data, size_t len)
{
    size_t i;
    uint16_t result = 0;
    uint8_t* hash   = (uint8_t*)(&result);
    for (i = 0; i < len; i++)
        hash[i % 2] ^= data[i];

    return result;
}
```

Checksum function used to create checksum of Tox address; data is the public key concatenated with a random nospam value.

DHT

The Tox protocol uses a distributed hash table (DHT) to facilitate connections between friends, and to allow for friend requests to be sent between clients. The Tox DHT is modeled after the DHT designed by BitTorrent (bittorrent.org), used for filesharing. Many of the needs of BitTorrent and Tox are similar, if not identical, and the onboarding documentation provided by the Tox project for developers wishing to understand the DHT explicitly links to BitTorrent's DHT protocol write-up at http://www.bittorrent.org/beps/bep_0005.html. Both of these protocols attempt to reveal as little as possible about their users, both perform UDP hole-punching for NAT networks, and, perhaps most importantly, both have to facilitate peer-to-peer links with clients who have no knowledge of the routing between them. Tox goes a great deal further, generating temporary addresses to prevent un-friended parties from tracking their status via the DHT, and implementing onion routing on top of that.

A quick introduction to the DHT follows, but it is important to note that, just like the topics of onion-routing previously and cryptography to come, a thorough treatment could fill books on its own.

Distance

Before delving into the details of the DHT and connections, a note needs to be made regarding the use of the term "distance" in regards to the DHT and Tox protocol. Much of the existing Tox documentation describes the distance between two nodes; this actually refers to the difference between two Tox ID addresses. The first Tox ID is XORed with the second, and the resulting binary number is interpreted as a big-endian integer which determines the distance between those two Tox IDs. Due to the properties of XOR operations, these comparisons are fast and efficient – and, due to the use of big-endian integers for this, a changed nospam value has no effect on distance.

Connecting to the DHT at Startup

A peer-to-peer network of clients acting in concert to ensure users can find each other in the network sounds like an ideal setup for a network like Tox, but before any other concerns regarding privacy and security are addressed, there is an elephant in the room: how do you connect a new peer to the network, when it doesn't know the addresses of the other peers? Tox takes an interesting approach to this, with users all around the globe running "bootstrap nodes." A bootstrap node is not a standard client; it runs either a process called "tox-bootstrapd" – a Linux/Unix compatible daemon, or one called "DHT_bootstrap", a cross-platform multi-architecture program. These processes are intended to act as a "first step" node for clients to connect to, retrieve information regarding other peers on the network, and then begin operating as nodes for the network.

Bootstrap nodes are intended to be hosts with high availability for both the host and its network connection. Developers of Tox clients usually have their programs pull data from the project's Bootstrap Nodes Status page (https://nodes.tox.chat), or directly hardcode a bootstrap node address and key for smaller projects (this project uses a node located at 198.199.98.108, located in the United States, for instance).

As a general rule, Tox uses port 33445 for communicating with Tox DHT nodes, but some nodes are run on 443 (default for HTTPS) or 3389 (usually used for Microsoft's Remote Desktop Protocol) to prevent port-based blocking.

After Bootstrap Connection

To walk through what happens after the bootstrap connection process is complete, let's assume the user for this client is Alice, and we'll refer to her local computer with the same moniker. We'll also assume she's friends with Bob. The bootstrap node will provide the client with a list (max 32) of peers

present in the network, involving Alice in routing for those nodes. Alice's client, then, will ping a random peer every 20 seconds, along with a ping for each friend in her friends list (in this case, just Bob). If a peer does not respond to any pings in five minutes, it is dropped from the peer list. At this point, Alice is an active, working node in the DHT.

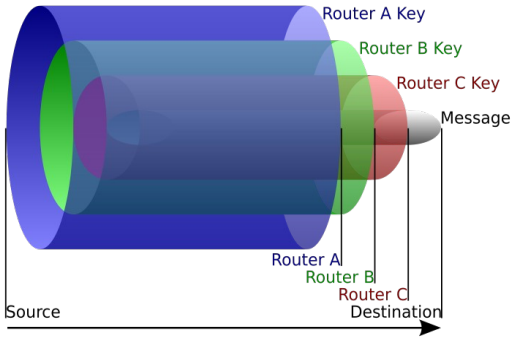
Pinging a friend involves checking to see whether the host currently has an available IP address and port for the friend. If so, the host sends a ping. If not, the host checks the peer list available to the host, and selects the peer with an ID closest to that of their friend. A request ping (encrypted using the public key of the friend, and containing the IP address and port for Alice) is then sent to that host, and the process repeats from there, with each node selecting a peer with an ID closest to that requested. If none can be found within 130 seconds, that request is dropped. When one such request reaches Bob, he is able to decrypt the request using his private key, extract the data contained therein, and then directly send a response to Alice over UDP without needing to traverse the Tox DHT (since he now knows the IP address and port of Alice). Once Alice receives his reply, their respective clients ping each other every 20 seconds with status updates, messages, or other indicators (file transfer requests, video conferencing requests, et cetera).

A similar process is used to determine which nodes a client will "announce" themselves to, giving out their permanent public key. In this situation, once a peer connects to the bootstrap node and pulls a list of peers, they compare their permanent Tox ID to the distance to each peer. The peers with a high distance are contacted using the temporary keypair, acting as a passthrough peer for their requests, but those with a low distance are queried using the permanent Tox ID as if searching for a friend. These low-distance peers should respond with any peers they are connected to that are "closer" to the permanent public key than they are. The original client will then "announce" its presence to the closest

few peers, sharing the permanent public key of the client and accepting friend requests through those peers.

Advanced DHT/Onion Routing

The previous explanation of the DHT is correct, but incomplete. In reality, the vast majority of connections between clients are done using onion routing and temporary key pairs. Alice is actually connecting to multiple nodes using onion routing – node-to-node connections are made, with each packet being encrypted using the public key of the next hop in the "relay," with at least 3 nodes involved in each of these relays. Messages are encrypted with the keys from each hop, with the last key being used first, then the second to last, et cetera. In this manner, each node receives the message, decrypts it, sees only the destination information for only the next hop in the relay, and sends it along.



Onion Routing Packet (Source: Wikipedia)

In addition, in most cases, the communications between nodes are done with temporary key pairs, and a node only announces itself using its permanent public key once it has determined that it has selected the node(s) closest to its public key out of the available peers. Therefore, in addition to bouncing messages between nodes, only a few nodes are going to receive the status information related to the permanent public key attached to our profile; this makes anonymous tracking of users vastly more difficult.

Cryptography

The mantra repeated within the security community has been “Don’t roll your own crypto!” (Cox, 2015). The reasoning behind this is simple: most people are unable to create their own secure cryptographic functions. The myriad mistakes that are made, claims that are later retracted, and vulnerabilities discovered are an industry unto themselves, and to rephrase Schneier’s Law, anybody can build a system they think is secure. The Tox project decided to take a wiser path, and instead implemented the NaCl (pronounced “salt”) networking and cryptography library for its cryptographic functions.

Cryptography Related to the Key Exchange

When a user creates a new Tox ID, one of the first tasks performed is the creation of a new key pair. Each key pair includes a private key and a corresponding public key, and each is 256 bits. ToxCore allows the explicit creation of a new keypair with tox_keypair_new(), although that is really just a wrapper for the NaCl crypto_box_keypair() function, which is implemented using Curve25519 (Bernstein et al, 2012). At this point, we’ll assume each party (by tradition, Alice and Bob again) already have a randomly-selected keypair.

If Alice wishes to contact Bob, she’ll send him a friend request with a message and the IP address and port combination where her client is listening; this friend request is encrypted using Bob’s public key and sent over the DHT. If Bob is online, his client will receive the message, decrypt it using Bob’s private key, and ask Bob whether to accept or reject Alice’s friend request. Bob accepts it, and his client sends an announce message to Alice that includes the IP address and port of Bob’s client and begins a proper key exchange. This results in a Diffie-Hellman-style key exchange, where the key used by Alice is composed with Alice’s private key and Bob’s public key, and Bob’s key is made up of Bob’s

private key and Alice's public key. That shared key is then used for communications between the two (modified with a random nonce) (Bernstein, 2009). It should be noted here that, due to these properties, handling keys for group messaging gets very complicated, very fast (requiring (n(n-1))/2 key exchanges if we are attempting to keep all group messaging participants connected to each other).

Cryptography Related to Messages

At this point, Alice and Bob have agreed on a shared key that will be used for all future communications. The key will be used with the Salsa20 stream cipher and a random nonce to produce the ciphertext of a message (or file, photo, or audio/video stream) using NaCl's crypto_box() function before being sent to the other party. The receiving party will use the same nonce to modify the stream sequence, but then can use the same key to decrypt the message into its plaintext via NaCl's crypto_box_open(). The nonce is required for perfect forward secrecy – if the key is recovered by a third party, previous messages or future messages cannot be decrypted without also knowing the nonce; this also prevents brute force attacks, as breaking one message gives an attacker access to one and only one message.

Non-Repudiation

In many use-cases, a primary concern regarding cryptography is non-repudiation: being able to prove that the individual who sent a message is the only person that could have. Tox uses public-key cryptography, so cryptographically-signing messages to provide such non-repudiation would certainly be technically possible, but the discussion above regarding key exchanges shows that it wasn't baked in to the Tox protocol (NaCl, 2016). The designers' decision to not support this kind of proof-of-authorship is not accidental – it provides upsides to the protocol on both efficiency (since symmetric

encryption is less computationally intensive, and digital signatures are undesirable under the project's goals), and pseudonymity.

Privacy was the main intention behind the development of this protocol, and the developers involved gave the subject the attention it deserved; in this case, "privacy" does not just mean that messages are encrypted while in transit, or that users who are not friends cannot determine one another's IP address, but it means mitigating the possible effects of side-channel attacks or other possible eventualities. Not supporting non-repudiation leaves open a very important last-ditch option: plausible deniability. If screenshots are unearthed or chat logs are available, the fact that multiple parties had access to the key used to create the message builds in to the protocol the ability to cast doubt on such logs. Unfortunately, this distinction between "public-key authentication" and "public key signatures" is unlikely to deter the most powerful or motivated attackers, like nation-states. In those cases, it is probable other methods would be wielded in search of information – whether that be rooting the user's equipment or physical coercion. While a lack of support for non-repudiation is not a perfect solution, it is nonetheless a helpful tool in the pursuit of defense in depth.

Friends and Messages

Adding a friend

A friend request can be sent to any Tox user that the sender knows the Tox ID of. It is encrypted using the public key of the individual it is being sent to. It contains within it the Tox ID of the sender, and allows for a message to be sent with it (i.e., a reminder regarding who the sending individual is, where the two may have met, an organizational identifier, et cetera). Some clients even support in-app lookup of Tox IDs, allowing for users to simply enter the address at which their friend is registered to

automatically retrieve the Tox ID attached to it. In these cases, the user can simply enter a string such as example_friend@toxme.io and hit enter, and the client will retrieve the Tox ID of example_friend and send the friend request to them.

In the case of friends that are currently offline, the Tox client will send a friend request, will not receive a reply (since the friend is offline), and will resend it after a set amount of time. Usually, the request will be sent once per minute for 5 minutes, then will begin increasing the delay between requests until the client is closed. When the client is reopened, if there were un-accepted friend requests that were being sent out, they will resume. Most clients will cease this behavior after an arbitrary amount of time; in such cases, if the client has been sending out friend requests to a specific Tox ID for something like 30 days, it will stop trying to do so (in some clients, this behavior is configurable to adjust the amount of time between initial attempt and giving up).

If the user receives a friend request and denies it, their client will ignore any further friend requests with the same message ID (that is, re-sent copies of the same friend request), but will ask again if another friend request is created (rather than simply re-sent). Some clients will also allow the user to blacklist Tox IDs; in these cases, any friend requests from the offending Tox ID will simply be discarded. This is a more targeted solution than regenerating nospam values, geared more towards stalking and harassment than spam prevention.

If a friend request is accepted, then a key exchange occurs. In addition to the details regarding key exchanges above, it also adds each user to the friends list of the other. Once that has occurred, each one can see the connection state set by the other (“Online”, “Away”, “Busy”, “Inactive”, or “Offline”), the status of the other (a string, set by the user), and retrieve the display name chosen by the user (a security concern outlined under the “Friends List” section that follows.

Preventing Spam Friend Requests

It is an unfortunate truth that, in the modern world, someone will attempt to monetize or abuse any service, venue, or communication medium. With email, spam has represented between 50 and 70% of total email traffic for the last few years (Spam, 2018); each messaging service deals with some form of this, from Facebook messages to Twitter tweets, and major phone carriers have a problem with spam texts and robocalls. With this in mind, what options are there to deal with the problem, since it obviously isn't going away? Most services allow users to report spammers, giving the service provider an opportunity to step in and revoke access for the offender. This requires a central authority to determine what kind of use constitutes abuse, and can often result in offenders creating multiple accounts to continue their activity. Tox, having no such central authority, had to get creative with preventing spam and misuse.

In public key cryptography, one wants to ensure their public key is widely available; the Tox project achieved this with the interesting choice to simply make that information part of a user's Tox ID, thus ensuring that if you have the Tox ID of a user, you also already have their public key. However, had they made the Tox ID of a user simply that user's public key (it is, to reiterate, a combination of the public key of the user plus a 4-byte nospam value and a checksum of these two values), then there would be nothing stopping enterprising individuals from looking up Tox IDs on a lookup service, IRC, or anywhere else they might find them and then proceeding to spam those users with friend requests including messages for whatever pharmaceutical they're advertising or stock they're pumping.

To prevent this, the Tox project added the nospam value and checksum. The purpose of the nospam value is to act as a mutable field to prevent widespread spam. To send a friend request to a user, the full Tox ID must be known; a friend request containing an incomplete Tox ID will be dropped

as an invalid address, while a Tox ID containing the public key of a user but an incorrect nospam value will simply never reach the intended recipient. Should someone start scraping a widely-used name lookup site or begin targeted spam campaigns, a user need only to go into their client and generate a new nospam value. This new nospam value replaces the old one, and a new checksum is calculated. This becomes the user's new Tox ID, which they can re-register with a name lookup service or leave unlisted. Thus, as soon as a user gets a single spam friend request, they can easily mitigate the impact of future abuse.

The true ingenuity of this approach is that while the Tox ID may change, these adjustments don't affect the public key at all – and that is the element used to locate friends on the DHT after a friend request has been accepted. The consequence of this is that once a user has accepted a friend request from someone, they can regenerate their nospam value as many times as necessary, and the friend will still be able to reach that user. Nospam values and the checksum are only used for routing friend requests, not status updates or messages between friends after they've performed a key exchange.

Lookup Services

Not having to rely on a central authority has a number of advantages, but one thing missing from the decentralized model is the ability to provide a search mechanism for peers – if an individual knows someone has a ToxID, how can they find it? In many situations, use cases, or threat models, they simply can't; Tox makes it rather simple to stay isolated from random users, speaking only with those the user has close ties to already. On the other hand, should a user desire a method for people to search for their ToxID, a number of services have been started to allow such searching. In most cases, such

services enable a user to find the ToxID of someone based on an email-style address (or have users register with an existing email address).

As an example, a widely-used such service is called ToxMe (https://toxme.io), and it allows a user to attach their ToxID and a short bio to an address that looks like username@toxme.io; they can give out that address, a great deal easier to remember than the 76-character ToxID, and any individual that wants to contact them need only input that address into ToxMe to retrieve the corresponding ToxID. Some clients take this a step further, allowing automatic fetching of ToxIDs. For these clients, support for at least the most common lookup services are built-in: a user can click “Add Friend”, and instead of pasting in the ToxID of the person they wish to contact, can use the username@service identifier, and the client will automatically query the service for the requisite ToxID attached to the name. These services also provide methods for updating a ToxID, changing profile information, or removing a user’s entry completely.

Friends List

Once a user has accepted a friend request, their Tox client assigns a “friend number” to them – although this is transparent to the user. The client attaches the necessary information about the friend (such as their public key and status message) to that friend number, and any translations between friend number and name, such as displaying the username during a chat, is done at runtime.

Some clients also support the ability for the user to assign permanent names to each friend, as the standard display name of a user is self-selected and fetched whenever status changes are made. This ability to select a permanent identifier for a friend is being pushed to all active projects, as the ability of a user to select their own name being displayed as the only identifier is considered a security risk (Users: Sharing IDs, 2018) – it allows ambiguous username situations, such as a user sending someone a friend request, having it accepted, then later changing their display name to masquerade as someone else (exactly the problem Twitter is currently having with “Verified” accounts changing their names to celebrities to push stock schemes and cryptocurrency scams (Warzel, 2018)).

Code

ToxCore and ToxCoreAV

ToxCore is the package that implements the main Tox capabilities, and ToxCoreAV builds upon that, adding the functions necessary to handle audio and video support – the call and videoconferencing that make Tox the “Skype replacement” that the project aimed for. A deep dive into how ToxCoreAV works is beyond the scope of this research, but it is worthy to note that the encoding technologies used in it (Opus for audio, VP8 for video, with discussions to update to AV1 in the near future) are all unencumbered by patents and royalty-free (they can be distributed without cost, and are unaffected by border restrictions as they do not violate standing patents).

BlueRing

BlueRing is the practical application of the research done regarding the Tox project. It is very simple, largely due to the fact that the ToxCore library handles all the necessary connections on the back end. The file “BlueRing.ini” is necessary (as it contains the options that will be passed to the ToxCore constructor via a super() call and provides the data necessary to set up the bootstrap connection), and the project includes the script “BlueRingConfig.py” that initializes the settings present in BlueRing.ini via a ConfigParser, should BlueRing.ini be corrupt or missing.

```
[BlueRing]
ipv6_enabled = True
udp_enabled = True
proxy_type = 0
proxy_host = None
proxy_port = 0
start_port = 0
end_port = 0
tcp_port = 0
savedata_type = 0
savedata_data = None
bootstrap_node = 198.199.98.108
bootstrap_port = 33445
bootstrap_key = BEF0CFB37AF874BD17B9A8F9FE64C75521DB95A37D33C5BDB00E9CF58659C04F
```

Contents of BlueRing.ini

In BlueRing.ini, we have a number of options, but the most important are those related to the bootstrap process – the information regarding the bootstrap node the program connects to can be seen as one of the US nodes listed at https://nodes.tox.chat; a full-featured client would likely pull JSON values from the same site.

Two of the other options present in BlueRing.ini are savedata_type and savedata_data; these control the loading of a data file by the client. In most clients, this option allows the ToxCore process access to a stored friends list, message history, and other persistent data, but it is important to recognize that these data files should be encrypted. Other clients require a password in order to decrypt the listed data file, as otherwise one runs the risk of saving sensitive information, such as the Tox IDs of friends and message history, in plaintext on disk.

The main program is BlueRingClient.py. In this script, the BlueRing class is defined as a child class of ToxCore, a ConfigParser pulls the options from BlueRing.ini, and functions for bootstrapping into the Tox DHT and ToxCore message handlers are defined. The capabilities of the program are exceptionally basic, meant only as a proof-of-concept due to difficulties encountered during development (chronicled under the Challenges: Project Postmortem section). The program connects to a Tox bootstrap node based in the United States, participates in the Tox DHT, and automatically accepts all friend requests sent to it and performs the key exchanges necessary to connect to friends. It allows the user to input messages on the CLI, but the capabilities to send messages are restricted to the last friend it received a message from.

A number of functions (tox_friend_status_cb(), send_avatar(), send_file(), tox_friend_read_receipt_cb(), and can_accept_file()) are only defined, not implemented. In an interesting and headache-inducing undocumented behavior, if these functions are not at least defined, the application will connect to the bootstrap node, load the list of connected clients, and then the

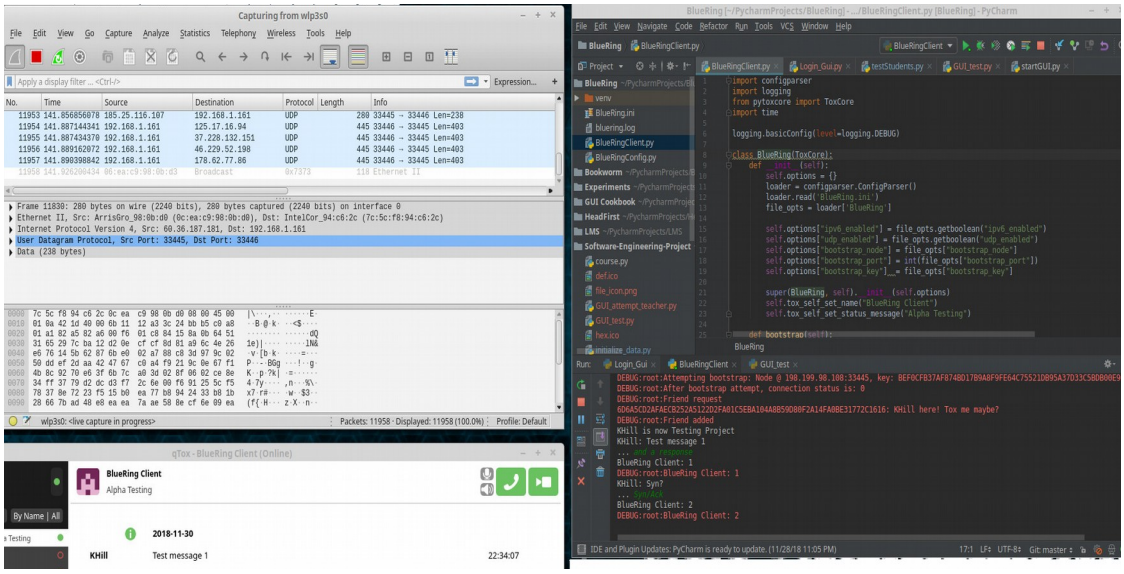
tox_iterate() function that does the heavy lifting within the application will silently hang, refusing to respond to any input, friend requests, or messages.

```
1  import configparser
2  from pytoxcore import ToxCore
3  import logging
4  import time
5
6  logging.basicConfig(level=logging.DEBUG)
7
8  class BlueRing(ToxCore):
9      def __init__(self):
10         self.options = {}
11         loader = configparser.ConfigParser()
12         loader.read('BlueRing.ini')
13         file_opts = loader['BlueRing']
14
15         self.options["ipv6_enabled"] = file_opts.getboolean("ipv6_enabled")
16         self.options["udp_enabled"] = file_opts.getboolean("udp_enabled")
17         self.options["bootstrap_node"] = file_opts["bootstrap_node"]
18         self.options["bootstrap_port"] = int(file_opts["bootstrap_port"])
19         self.options["bootstrap_key"]  = file_opts["bootstrap_key"]
20
21         super(BlueRing, self).__init__(self.options)
22         self.tox_self_set_name("BlueRing Client")
23         self.tox_self_set_status_message("Alpha Testing")
24
25     def bootstrap(self):
26         logging.debug(f'Attempting bootstrap: Node @ {self.options["bootstrap_node"]}'
27                       f':{self.options["bootstrap_port"]}, key:                    ↵
                          {self.options["bootstrap_key"]}')
28         self.tox_bootstrap(self.options["bootstrap_node"],                         ↵
           self.options["bootstrap_port"],
29                            self.options["bootstrap_key"])
30         logging.debug(f'After bootstrap attempt, connection status is:             ↵
           {self.tox_self_get_connection_status()}')
31
32     def run(self):
33         self.bootstrap()
34
35         print(f"Connected, using ToxID {self.tox_self_get_address()}")
36         disconnectionCheck = False
37         iteration_interval = self.tox_iteration_interval()
38
39         while True:
40             if not disconnectionCheck and self.tox_self_get_connection_status() !=  ↵
               ToxCore.TOX_CONNECTION_NONE:
41                 disconnectionCheck = True
42
43             if disconnectionCheck and self.tox_self_get_connection_status() ==      ↵
               ToxCore.TOX_CONNECTION_NONE:
44                 logging.debug(f"Connection interrupted: attempting bootstrap       ↵
                   reconnect")
45                 self.bootstrap()
46                 disconnectionCheck = False
47
48             self.tox_iterate()
49             time.sleep(float(iteration_interval) / 100.0)
50
```

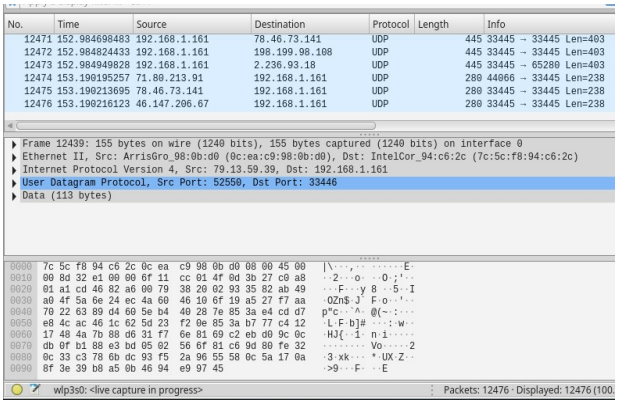
```
51      def tox_friend_message_cb(self, friend_number, message):
52          friend_name = self.tox_friend_get_name(friend_number)
53          print(f"{friend_name}: {message}")
54          string = input("Response: ")
55
56          outbox = self.tox_friend_send_message(friend_number,
            ToxCore.TOX_MESSAGE_TYPE_NORMAL, string)
57          print(f"{self.tox_self_get_name()}: {outbox}")
58          logging.debug(f"{self.tox_self_get_name()}: {outbox}")
59
60      def tox_friend_request_cb(self, public_key, message):
61          logging.debug(f"Friend request\n{public_key}: {message}")
62          self.tox_friend_add_norequest(public_key)
63          logging.debug(f"Friend added")
64
65      # The following functions
66      #         tox_friend_status_message_cb()
67      #         tox_friend_status_cb()
68      #         send_avatar()
69      #         send_file()
70      #         tox_friend_read_receipt_cb()
71      #         can_accept_file()
72      # Are not strictly necessary for the program to run -- but without them, the
        tox_iterate loop will sometimes hang
73
74      def tox_friend_status_message_cb(self, friend_number, message):
75          print(f"{self.tox_friend_get_name(friend_number)} is now {message}")
76
77      def tox_friend_status_cb(self, friend_number, status):
78          pass
79
80      def send_avatar(self, friend_number):
81          pass
82
83      def send_file(self, friend_number, path, name=None):
84          pass
85
86      def tox_friend_read_receipt_cb(self, friend_number, message_id):
87          pass
88
89      def can_accept_file(self, friend_number, file_number, kind, file_size, filename):
90          pass
91
92      def tox_file_recv_cb(self, friend_number, file_number, kind, file_size, filename):
93          pass
94
95
96  bluering = BlueRing()
97  bluering.run()
```

Wireshark Packet Capture

While executing, the program is involved in a large amount of network traffic (small packets, but lots of them). This is due to the peer-to-peer nature of the protocol, and the protocol's reliance on onion routing – almost immediately, the BlueRing client is brought into the DHT and operating as a relay for other clients, helping to hand off packets from the DHT to other peers. The portions of the packet captures that are most interesting are the ports involved and the fact that the UDP payload for all packets are completely unintelligible and random-looking, precisely what is desired for encrypted communications.



Wireshark being used on an open session



Wireshark packet capture detail - encrypted packet payload

Challenges: Project Postmortem

Overview

Challenges with implementation and documentation were anticipated during the planning phase of this project; the scope, breadth, and depth of problems encountered during development certainly were not. Since the Tox project itself is, realistically, still in the beta stages, the documentation is incomplete and occasionally inconsistent. This led to a string of situations in which a particular tool or package appeared to help solve a problem or fix an issue, only to fail at delivering any meaningful progress. There were pitfalls that seemed nearly fatal to the project, but stubbornness is sometimes a virtue. The end product of a working messaging client is a testament to the usefulness of tenacity.

Installation and Dependency Issues

During the initial conception of this project, the PyToxCore Python bindings for the ToxCore library were found, and the PyToxCore module looked like it would perform well for the project. The Github project page included documentation, code examples, and it looked (on the surface) like an ideal base for experimentation. Unfortunately, it did not turn out to work so simply. Multiple attempts over the course of weeks were made, to no avail; cryptic error messages were the only output, usually with the entirely unhelpful “PyToxCore could not be installed: exiting” being as specific as it got. A second Python set of bindings was found, PyTox, but it was unsuitable for a different set of reasons, to be covered shortly. At that point, it was looking like the only available option would be to scrap the idea of a Python client entirely and construct it in C++. Instead, when a complete operating system reinstall was necessary on the system being used for development, installing PyToxCore succeeded; it is suspected that there was a set of dependencies that pip (the Python module installer) was unable to resolve due to conflicting packages on the system, but given how delayed the project was already

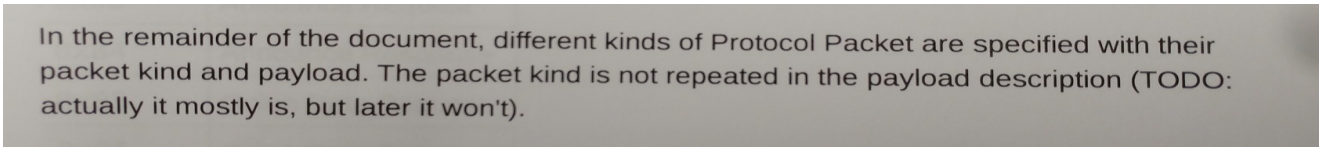
thanks to these problems, no more time was available to try to figure out precisely why it worked in the new environment.

Docker Difficulties

After the failure of PyToxCore to properly install, alternate options were explored. The most promising was a slightly older set of Python bindings known as PyTox. More issues with this package ensued, and it became obvious that similar problems as before were being encountered. This package, though, also included an extra tool: a Docker image. Docker is a container platform that allows developers to bundle up all the components necessary to run a particular application (all the dependencies, packages, helper daemons, everything) into a single file, which is then executed by the Docker process to run – almost akin to a virtual machine. While it eventually succeeded to execute, it also proved to be a dead end (in that while it would run, it wouldn't properly allow a Python 3 instance inside the Docker image to import the PyTox library), it had the costly time-consuming side-effect of forcing a quick introduction to Docker, how it works, and how to use it, into a project already behind schedule.

Documentation Ambiguities and Language Barriers

The documentation for most of the Tox project is clear, concise, and readable. Unfortunately, the remainder of it don't have any of those qualities. Attempting to understand how a function operates might sometimes require looking at the source code for the function, reading the documentation, finding oneself in a cloud of befuddlement as the two don't quite square up together, then realizing that there is a note elsewhere in the documentation stating that the code does not, in fact, behave in the manner described by the protocol specification yet, but is being adjusted.



Protocol Packet Payload Description

An additional hurdle that was encountered was language: the Python bindings for ToxCore that were used for this project include their documentation, descriptions, et cetera in English. In-line code comments and docstrings, though, were foreign – all comments describing the code, the examples, how certain functions were being utilized, and what arguments they took were all in Russian. Unfortunately, this particular trait of the codebase was not caught until after the package had been installed, and a deep dive into example code and the built-in documentation was queried. While not fatal to the project, it was just another problem piled upon the heap whose size had been underestimated by the researcher.

```
Проверка, что файл можно принимать

Аргументы:
    friend_number (int) -- Номер друга
    file_number   (int) -- Номер файла (случайный номер в рамках передачи)
    kind          (int) -- Значение файла (см. TOX_FILE_KIND)
    file_size     (int) -- Размер файла
    filename      (str) -- Имя файла

Результат (bool):
    Флаг разрешения на принятие файла
```

Code Block Comments

Code Which is the Opposite of Pythonic

The example code provided with PyToxCore is an interesting look at the work of another developer. This is not meant to demean the work of Anton Batenev, who appears to be the developer behind the PyToxCore bindings. The bindings themselves work, so they do work for this project, but the example code is bewildering; the example code works, but a bot similar to the one constructed for this project is over 620 lines of code, imports some packages without using some of the central tools made available within those packages, imports other packages without using them at all, and, while the code works, one can’t help but feel as if it shouldn’t.

An excellent example of this is that the example code implementing a configparser within the example code – it includes the following function:

```
@staticmethod
def _bool(value):
    """
    Преобразование строкового значения к булевому

    Аргументы:
        value (str|bool) -- Строковое представление булева значения

    Результат (bool):
        Результат преобразования строкового значения к булеву - [true|yes|t|y|1] => True, иначе False
    """
    if type(value) is bool:
        return value

    value = value.lower().strip()

    if value == "true" or value == "yes" or value == "t" or value == "y" or value == "1":
        return True

    return False
```

_bool function used in example code

This function takes a string as an argument, returns true if that string is “true”, ”yes”, ”t”, “y”, or “1” (case-insensitive), and returns False otherwise. While it does what it claims to, the Python ConfigParser module contains the configparser.getBoolean() function, that does precisely what this function does (but universally, and better), and, most importantly, this example program already imports the configparser module, giving it access to getBoolean() without any extra work. Examples like this are littered throughout the example code, and make it very difficult to follow, nigh-unreadable, and enormous.

Post-Mortem Summary

In Hindsight

The planning stage of this project was wide enough in breadth, but did not properly fathom the depth of what would be required. During the planning phase, steps were taken to ensure that this project was realistic, that the tools necessary to complete it were available, and that the decisions made during project conception were sound. Unfortunately, each of these was performed in a manner that left

blind spots; huge time sinks lurked in the corners of each of these choices, and ultimately hamstrung the development process.

Care was taken to ensure that the Tox project was still being actively developed, so the codebase was not completely abandoned. In fact, those active in the project were organizing the developer convention in Vienna at the time, so it had a core group of developers that actually care about the project and are working to improve it.

Python was chosen as the most suitable language for the project, but for educational purposes. The researcher's own personal experience with Python began with a data mining course, but it was a very brief, shallow interaction. Between semesters after that course, extracurricular projects allowed for a proper introduction to Python and its ease of use, brevity, simplicity, and readability; this made it an ideal candidate for this project. The fact that the researcher was less experienced in it than C++ or Java just meant working harder to catch up, making the educational use of Python for a large project an even more "efficient" learning experience – this assumption turned out to technically be correct (the amount and scope of the necessary learning to get to this point was prodigious, and very effective), but also set circumstances up that required a complete rewrite of the goal for this project and repeated rewrites of the BlueRing client itself. If the opportunity was available again, the choice of Python would still be likely, but more realistic goals regarding the scope and expectations of the project would have been set down in the beginning.

Python bindings for the ToxCore library were found, and it was ensured they were downloaded and complete. Installation was attempted, and failed – but it was assumed to be the result of relative inexperience with Python instead of the red flag that it should have been interpreted as. In hindsight, that installation failing should have caused more apprehension regarding its selection as a cornerstone

of the project, but optimism and the excitement around the topic clouded the importance of that roadblock.

Final Notes

The knowledge gained about how Python works is sure to be useful in the future – knowing how to use Python, the current lingua franca within the security community, in addition to something closer to “bare metal” like C++, is likely to be very useful across a range of situations. In addition, getting to know some of the modules available within Python has already paid dividends, as GUI programming with Tkinter and getting familiar with the ConfigParser module and the functions available in it have both been useful for other courses. Along the same lines, increased fluency in Python has made certain scripting tasks easier in current professional capacities.

A hard lesson was learned in regard to project time management; time should have been front-loaded in the planning phase on preliminary testing of the target libraries. Plans for risk assessment and mitigation did not properly factor in the true time cost of getting a working environment set up, given the problems that were faced in getting PyToxCore to successfully install and function. Realistically, more time should have been invested over the previous summer such that on the first day of the Senior Project course, access to either a completely setup development environment (including working libraries) was available, or the knowledge of the installation complications was known and the scope of the project could be adjusted at the outset instead of chasing dead-end solutions. Out of the teachings of this project, this is the most disappointing and heavy.

Finally, it was eye-opening to get to delve deeply into the Tox protocol. The manner in which the Tox project has considered the privacy implications of every choice surrounding the protocol is awe-inspiring, and some of the clever ways in which they’ve solved certain problems (like generating

temporary key pairs to pull peer lists in order to prevent bootstrap nodes from being able to track connections, or using symmetric-key cryptography to preserve plausible deniability) make it clear that a lot of effort and thought has gone into the project. An element of paranoia pervades the project, treating each node as if it might be malicious and making every effort to protect the data of the users, and diving in to the implementation details and structure steeps readers in that same paranoia. Thoughts have often turned to considerations about how userdata is promiscuously shared with every entity they interact with and the businesses connected to it; what patterns would emerge from the data, and what it could tell a motivated attacker. On the whole, this is probably not entirely healthy, but it is likely that this reflex will serve a definite purpose in the defense of the networks overseen in the future. During the reading for this project, a ticket was found that referenced an issue in which, under very certain circumstances, it would be possible for an individual to ascertain another user's IP address from the DHT knowing only their Tox ID (without being friends with the user); the project responded immediately, treated it as a severe security risk and the ToxCore codebase was patched straightaway. Given the indifference to data leakage shown by companies like Facebook and Yahoo, this contrast was striking – and educational.

Overall, while the most frequent lessons were those in humility, it was also very informative in regards to Python, privacy, and networking; even though writing a full-fledged, feature-complete Tox client is apparently outside of the researcher's current skills, the knowledge of the protocol gained from this effort can aid in assessing the security implications inherent in other communications systems.

Works Cited

A New Kind of Instant Messaging. (n.d.). Retrieved April 04, 2018, from https://tox.chat/faq.html

Adrian, D., Bhargavan, K., Durumeric, Z., Gaudry, P., Green, M., Halderman, J. A., ... Zimmermann, P. (2015). Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice. In *CCS 2015 - Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (Vol. 2015-October, pp. 5-17). Association for Computing Machinery. Retrieved April 06, 2018, from https://weakdh.org/imperfect-forward-secrecy-ccs15.pdf

Bernstein, D. (2009). Cryptography in NaCl. Retrieved November 12, 2018, from https://cr.yp.to/highspeed/naclcrypto-20090310.pdf

Bernstein, D., Lange, T., Schwabe, P. (2012). The Security Impact of a New Cryptographic Library. Retrived November 14, 2018, from https://cr.yp.to/highspeed/coolnacl-20120725.pdf

Beurdouche, B., Bhargavan, K., Protzenko, J., Zinzindohoué J. K. (2017). HACL*: A Verified Modern Cryptographic Library. Retrieved April 06, 2018, from https://eprint.iacr.org/2017/536.pdf

Cox, J. (2015, December 10). Why You Don't Roll Your Own Crypto. Retrieved November 12, 2018, from https://motherboard.vice.com/en_us/article/wnx8nq/why-you-dont-roll-your-own-crypto

Finley, K. (2017, June 03). Out in the Open: Hackers Build a Skype That's Not Controlled by Microsoft. Retrieved March 30, 2018, from https://www.wired.com/2014/09/tox/

Goth, G. (2006, June). VoIP Security Gets More Visible. Retrieved April 06, 2018, from https://www.computer.org/csdl/mags/ic/2006/06/w6008.pdf

Lund, J. (2018, October 29). Technology preview: Sealed sender for Signal. Retrieved October 31, 2018. https://signal.org/blog/sealed-sender/

Levy, D. (2001). *Crypto: How the Code Rebels Beat the Government -- Saving Privacy in the Digital Age*. Penguin Books.

Marvin, R. (2018, January 11). Microsoft Tests End-to-End Encryption in Skype Conversations. Retrieved April 06, 2018, from https://www.pcmag.com/news/358489/microsoft-tests-end-to-end-encryption-in-skype-conversations

NaCl: Networking and Cryptography library. (March 15, 2016). Retrieved November 02, 2018, from https://nacl.cr.yp.to/box.html

Newman, L. H. (2018, January 11). Skype Finally Starts Rolling Out End-to-End Encryption. Retrieved April 04, 2018, from https://www.wired.com/story/skype-end-to-end-encryption-voice-text/

Nordrum, A. (2016, April 22). In Privacy Versus Security, End-to-End Encryption Is Definitely Winning. Retrieved April 04, 2018, from https://spectrum.ieee.org/tech-talk/telecom/security/in-privacy-v-security-endtoend-encryption-is-definitely-winning

'Nurupo'. (2018, April 20). Security vulnerability and new Toxcore release. Retrieved November 11, 2018, from https://blog.tox.chat/2018/04/security-vulnerability-and-new-toxcore-release/

Robinson, R. (2007, June). A Reality Check on Security in VoIP Communications. Retrieved April 06, 2018, from http://sites.ieee.org/denver-com/files/2014/10/Security-in-VoIP-Communications-June-2007.pdf

Schneier, B. (2014, April 15). Metadata = Surveillance. Retrieved November 11, 2018, from https://ieeexplore-ieee-org.ezproxy.uhd.edu/stamp/stamp.jsp?tp=&arnumber=6798571

Spam statistics: Spam e-mail traffic share 2018. (n.d.). Retrieved November 11, 2018, from https://www.statista.com/statistics/420391/spam-email-traffic-share/

Tox Project. (2017, July 30). About – Tox. Retrieved October 30, 2018, from https://tox.chat/about.html

Users: Sharing IDs (2018, March 15). Retrieved November 14, 2018, from https://wiki.tox.chat/users/sharing_ids

Warzel, C. (2018, February 27). Twitter Is Still Allowing Scammers To Hijack Verified Accounts To Take People's Money. Retrieved November 18th, 2018, from https://www.buzzfeednews.com/article/charliewarzel/twitter-allowed-cryptocurrency-scammers-to-hijack-verified