A Cryptographically Protected Phone System

Andres Erbsen Adam Yedidia

October 30, 2014

Introduction

In today's excessively monitored and public world, privacy has become a luxury. Intercepting phone calls, emails, and internet communications is withing the reach of a curious middle-schooler, and the very companies who provide the communications systems usually fund themselves through spying on us. We are seeking to change this: to develop a phone system that cannot be silently intercepted regardless of how the signal is transmitted or who has access to it while maintaining a simple user experience.

To make this happen, we are going to use modern cryptography. The last century, in addition to bringing about the technologies mentioned above for communicating remotely (along with the technologies used to eavesdrop on them) also brought the invention of clever cryptographic schemes for hiding information securely. These schemes go by many names; in our implementation, we will make use of a particular implementation the one called Diffie-Hellman key exchange. The truly incredible thing about Diffie-Hellman key exchange (along with a handful of other schemes which we will not use) is that it enables two interlocutors *with no prior agreements or contact* to communicate securely in the presence of an eavesdropper *who can hear everything they ever say to each other*. Conditioned on the hardness of various problems which are widely believed to in fact be very difficult to solve by the wider academic community, it is in fact provably intractable for the eavesdropper to hear the communications.

In what follows, we will describe a hardware implementation of a telephone system that will be cryptographically protected in a theoretically defensible way; that is, for an attacker to be able to eavesdrop on a call using our telephone system would require either a long-standing conjecture to be false, or would require more vastly more computational power than is available to the world today.

Planned User Experience

Two lab-kits are connected with a couple of wires. When a call is initiated using a button, both hex displays show the same verification code. Anything spoken into one lab-kit's microphone is heard from the headphones of the other and vice versa.

The users are encouraged to read the displayed verification code to each other and check that it is indeed the same for both of them – the protection against sophisticated man-in-the-middle attacks (described later) depends on this check being performed correctly. However, the sole presence of that value is a hurdle for any potential attacker, because they have to decide whether or not to intercept a call before they learn whether the users are going to check the verification code – and assuming the implementation does not have bugs, a verification code mismatch is a definite indication of an ongoing attack.

Transmitting Audio

To transmit the audio across the channel, we will organize the PCM data of the signal into packets, to be sent one at a time across the channel. It is more efficient to perform the cryptographic calculations when the input is in available in larger chunks than a single audio sample. Moreover, each packet requires a header, which will contain a 128-bit cryptographic authenticator and some minimal information. The first field will indicate whether the packet is a data packet or some other specialized signal to the system, such as a command or a preliminary communication. The next will be a sequence number unique to each packet. After that, the length of the following data will be indicated. Finally, the rest of the packet holds the encrypted audio data, followed by the 128-bit cryptographic authenticator (similar to a checksum). The header is included in the calculation of the authenticator and the length of the authenticator is included in the length field.

This design allows us to vary the number of data bits in a packet, so we will be able to tweak the number of bits we send at once because it is not immediately obvious what the value should be; a smaller value will increase the relative overhead of the packet header and make the cryptographic calculations less efficient. A larger packet size, however, will increase the latency of the system, because the system will have to accumulate more bits before it can send a packet. We are likely to want a packet that is as small as possible without further complicating the implementation. Packets will be assembled in BRAM.

Finally, the packets will be encrypted and transmitted to the other lab-kit. We hope that a wire driven by the FPGA will suffice for sending a sufficiently clear signal; if not, we will use a MAX395 chip (designed for this very purpose) instead.

Cryptographic protection

As it is easy for an individual to come up with a cryptosystem that they themselves cannot break but harder to create one that will withstand attacks from others, we will be basing our design on existing well-audited building blocks. As we are not aware of any freely available Verilog implementations of these algorithms, we will be implementing them and verify their correctness by comparing the behavior of our hardware implementations to the existing reference implementations (software).

The overall plan is as follows: in the very beginning of each call, the two communicating lab-kits will perform a Diffie-Hellman handshake to generate a shared secret. That secret will be used to seed a pseudo-random keystream generator and the digital audio signal will be xor-ed with the keystream. This prevents the attacker (who does not know the shared secret or the keystream) from recovering the audio signal. To also prevent the attacker from selectively modifying the audio, each audio packet will be accompanied by a 128-bit authenticator that can only be computed for a message by a party that knows the secret key – the callers, but not the attacker.

Call Initialization: g is a public constant. Participant A chooses random a and reveals $\text{hash}(g^a)$, receives $\text{hash}(g^b)$, reveals g^a, receives g^b, checks that $\text{hash}(g^b)$ is as expected. B acts symmetrically. Finally, both have $s = g^{ab}$.

Encryption: $\text{encrypted}_i = \text{data}_i \oplus \text{keystream}(s, i)$

Decryption: $\text{data}_i = \text{encrypted}_i \oplus \text{keystream}(s, i)$

Packetization: $\text{packet}_i = (\textit{type}, i, \text{length}(\text{encrypted}_i), \text{encrypted}_i)$

Authentication: $\text{authenticator}_i = \text{hash}(s, \text{packet}_i)$

Auth. checking: Accept received packet iff $\text{hash}(s, \text{packet}_i) = \text{authenticator}_i$.

The described mechanisms are sufficient against an attacker that can only see what is sent over the wire but tamper with it; for example, one might cut the wire and connect it to their phone instead and all this would continue without interruption. Furthermore, they might make another call to the intended recipient and connect these calls while maintaining the ability to eavesdrop. To make sure that the two users of ours system are indeed on the same call (and thus not eavesdropped on), we display a value derived from the handshake inputs (g^a and g^b) to both of them – if it matches, the connection hasn't been tampered with. Displaying just g^a and g^b would also work, except that each of them is 255 bits long and comparing them would be tedious. Instead, we display the first 32 bits of $\text{hash}(g^a, g^b)$, where the hash function is such that changing just one of the inputs would chaotically change the output. Note that it is important to have each party to commit to the choice of their g^a (resp. g^b) value before seeing the other party's choice – otherwise a powerful adversary could try about 2^{32} options for their own value until he finds one for which $\text{hash}(g^a, g^b)$ coincides with the verification code for some other conversion of their choice. If they could do that, they could deceive two users into thinking that there is a direct call between them, while actually the adversary is relaying all audio between two calls.

As classical Diffie-Hellman requires hundreds of modular arithmetic operations on multiple-thousand-bit numbers to be secure, we will be using a modern variation where every modular multiplication is replaced with the addition of two points on a carefully chosen elliptic curve. The other relevant properties of elliptic curve addition are the same as for modular multiplication, we will even continue to use the classical notation. Even though one elliptic curve addition uses more than one modular multiplication, smaller numbers can be used without compromising security, and this enables solid performance with relatively simple implementations. Our choice of primitives goes as follows:

Random numbers: Hash of 100ms of microphone input.

Diffie-Hellman function: Curve25519 [3] (uses arithmetic modulo $2^{255} - 19$).

Keystream: ChaCha20 [2] (uses 32-bit addition, xor, and rotation by constant).

Hash: BLAKE[1] (uses 32-bit addition, xor, and rotation by constant).

The last two are relatively straightforward to implement given the specification, as is elliptic curve Diffie-Hellman if an implementation of the underlying modular arithmetic is given. Even though efficiently implementing generic modular arithmetic is difficult, the circuit for special case of calculating modulo a prime that is close to a power of two turns out to be both simpler and more efficient than one might expect based on standard reasoning about multiplication. Nevertheless, we predict that getting all the cryptographic computation right will be the most time-consuming part of the project. To mitigate the inherent risk of something going wrong due to human error, we will test each subcomputation against an independently developed software equivalent.

Security Analysis

It is the nature of practical cryptography that it is very difficult to prove the security of your system. While we do have an informal argument that could likely be extended to a proof, doing this is outside the scope of this class. We will instead enumerate the widely-known attacks that have worked against similar systems, and explain why each one cannot succeed against ours.

Computational Attacks: Computing g^{ab} given only g^a and g^b where g is a point on Curve25519 was conjectured to be intractable in [3] and no noteworthy progress has been made since. Computing m from $m \oplus \text{keystream}_i$ is impossible if keystream_i is unknown, and as with Curve25519, the conjecture is still standing that the ChaCha20 keystream cannot be deduced without knowing the secret key used to generate it.

Man-in-the-Middle Attacks: This type of attack is based off the idea of spoofing the two interlocutors into thinking that they are talking to each other, when in fact, they are both talking to the attacker and the attacker is relaying messages between them. No amount of security on the channel itself can protect against an attack of this kind, since the attacker's ability to hear the communications comes from the fact that two participants in a phone conversation can hear the communications – something that is necessary for the basic functioning of a telephone system. However, we can give the interlocutors the ability to know when there is a man-in-the-middle attacker, by giving everyone access to the public key of a given other person, and also giving them access to the public key of the person they're talking to. If the Alice believes herself to be speaking to Bob, but is actually speaking to Eve, who is executing a man-in-the-middle attack, then she will see both Bob and Eve's public keys displayed on her phone's screen, and this will tell her that she is not speaking to the person she think she is. This is the basic idea; in reality, we would display not the public key itself (which is much too long to display) but a hash thereof, and if Bob and Eve's public keys were different, then the hashes would be virtually certain to also be different (again, this is a conjectured property of the hash function).

Side-channel attacks: Unlike when designing for a cpu, circuit-level programming enables fine-grained control over what computation is performed how and thus enables us to totally avoid large classes of side-channels that have proven disastrous for software implementations. All functions that operate on secret inputs will be implemented in constant time, therefore the overall timing of packets will be independent of the secret key used and cannot leak any information. We could assume that over the duration of one call the device remains in the exclusive possession of the legitimate user and the adversary cannot make any measurements about it. Nevertheless, we will take care not to vary the parts of circuitry that are active based on information we aim to keep secret.

References

[1]Aumasson, J.-P., Henzen, L., Meier, W. and Phan, R.C.-W. 2010. SHA-3 proposal bLAKE. Submission to NIST (Round 3).

[2]Bernstein, D.J. 2008. ChaCha, a variant of salsa20.

[3]Bernstein, D.J. 2006. Curve25519: New diffie-hellman speed records. *Public key cryptography - pKC 2006, 9th international conference on theory and practice of public-key cryptography* (2006), 207–228.