procd: A privacy-preserving robust implementation to discover contacts in social networks

Fabian Deifuß
University of Wuppertal,
Germany
fabian.deifuss@uni-wuppertal.de

Cornelius Ihle
University of Wuppertal,
Germany
ihle@gipplab.org

Moritz Schubotz
FIZ Karlsruhe,
Germany
moritz.schubotz@fiz-karlsruhe.de

Bela Gipp
University of Wuppertal,
Germany
gipp@uni-wuppertal.de

Abstract

Current instant messengers store the users’ phone book contacts typically unencrypted or hashed on a central server. In case of a server’s corruption, all contacts are either directly available in plaintext or can be unmasked using a simple dictionary attack. To solve this problem, we present procd [pʁoːst] a python implementation for privacy preserving contact discovery. procd is a trustless solution that requires neither plaintext numbers nor hashes of single phone numbers to retrieve contacts. Instead, we transfer hashed combinations of multiple phone numbers, which increases the effort for dictionary attacks to an unfeasible level using today’s hardware.

Keywords: private contact discovery; private set intersection; secure multi-party computation; private information retrieval

1 Introduction

State-of-the-art social networks and messaging services store a social graph of its users to suggest communication options. Having a service provider storing and sharing a social graph is not privacy-preserving and should be avoided whenever possible.

procd [pʁoːst] - Private RObust Contact Discovery is our approach to private contact discovery with increased robustness against brute force attacks and without the need to store a social graph. We instead use a minimal social graph each user already has on its phone, the address book. Thus, the contact data remains distributed and owned by the user

Our goal is to find a way to increase the complexity of a brute force attack to a point where it is computationally infeasible to find an input that hashes to the processed values. For a dictionary attack on contacts, an attacker systematically tries each possible phone number as an input to match and unmask a hashed value to reveal numbers and connections.

2 Related Work

One of the most promising implementations, has the Signal messenger[1]. Signal clients hash their phone number locally before uploading (Marlinspike, 2017) it to the Signal servers. However, even though this is better than uploading and storing everything in plaintext, a typical phone number only consists of about ten digits. Hence, these hashes are vulnerable to dictionary attacks (Bošnjak et al., 2018). Signal is aware of this issue and therefore has to rely on a hardware solution called Software Guard Extension (SGX) from Intel. This, however, moves the trust issue to another party - the hardware manufacturer.

3 Method

For our approach, we aim to meet three criteria:

1. No exchange of plaintext contact information
2. Robustness against brute-force attacks
3. No dependency on single proprietary hardware solutions

We hence, propose an unbalanced private set intersection with increased input complexity.

3.1 Pairwise Hashing

Instead of hashing a single phone number, we form a hash of a pair of numbers. Each hashed pair consists of a user's phone number and one of her contacts.

In pairwise hashing, the server only ever sees the published hashes and does not gather any registered client's information. Even though the input complexity increases drastically, most benefits vanish if an attacker already has information like the (1) relation between the targets or (2) the individual's phone number. Therefore, it is necessary to salt the hash with a privately disclosed secret known only to the two parties trying to communicate.

[1] https://signal.org/blog/private-contact-discovery/

Once a client knows about already registered contacts, a Diffie-Hellman key exchange (Li, 2010) is used to enable authentication and initiate private communication. The public keys can be extracted from our public database. However, we cannot just post each party's public key linked to their phone number. Hence, we apply a way to get a hold of each other's public key without exposing the corresponding phone number, neither in plaintext nor hashed. As mentioned earlier, we suppress false positives by hashing our combinations in two different orders, starting with Bob or starting with Alice. This way it is possible to store one's public key alongside the hash known by both parties.

3.2 Experiment

A REST API serves as an interface to GET an intersection of already registered contacts and POST a user registration to our exemplary service. Figure 1 shows all used system entities and their interfaces.

The REST API processes the user requests and inserts or retrieves information from the public PostgreSQL database, which holds all hashed phone number combinations together with their public keys. The client constructs two dictionaries of number combinations before interacting with the API. These two dictionaries differ in the order of phone numbers, but both can contain a common secret (salt), unique to each contact. The first dictionary (Dict1) is used to publish all contacts to the contact discovery service. The second dictionary (Dict2) is not published but is used to verify any retrieved hash from the service and filter false positives. Additionally to the dictionary, a user's public key is appended to the hash combinations before publishing so that the desired public key can be retrieved and used for an encrypted communication initiation through the messaging service.

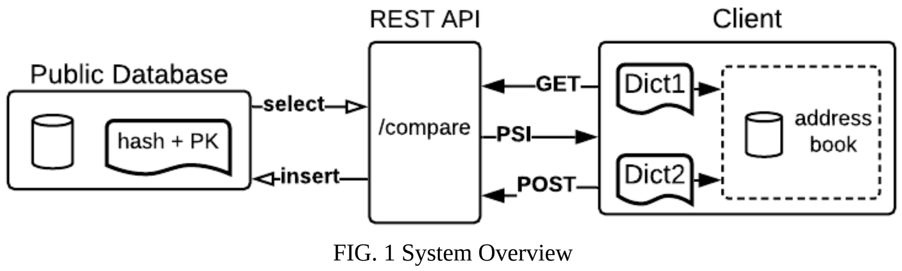

FIG. 1 System Overview

4 Evaluation

In the following paragraphs, we analyze and discuss our architecture with regard to each of the three criteria we aimed for.

4.1 Robustness Against Dictionary Attacks

The workload to compute hashes is horizontally scalable. Thus, the critical metric to evaluate the feasibility of computing a specific dictionary is the price to pay for the necessary computing resources.

On a modern computer (6 cores, 2.8GHz), it takes 0.00063 milliseconds (6.3e-7 seconds) to compute a SHA1 hash. This translates to 1.5e+6 hashes per second. Assuming the desired output hash is computed after half of the possible combinations (input complexity of 8e+20 without salt), it would take approximately 8 million years of computing to get the desired hash. A comparable VM rental on Azure is about 0.30$ per hour. Hence an attacker would need 21 billion USD for a successful dictionary attack.

Table 1 shows the estimated time required to compute the desired hashes for a contact list of 200 entries on modern hardware (single machine) with their estimated costs alongside the different hashing complexities.

TABLE 1 Dictionary Attack Cost Estimation

Input Complexity	Estimated time	Estimated cost of computation
German number hash (1e+6)	1 second	<0.01 USD
Number hash (4e+11)	1.5 days	10 USD
Pairwise German number hash (1e+12)	7.3 days	52 USD
procd German number hash (5e+18)	10.000 years	262 million USD
Pairwise WhatsApp User hash (8e+20)	8 million years	21 billion USD
procd WhatsApp User hash (4e+27)	40 trillion years	105 quadrillion USD

Using our methodology, the upfront cost of resources necessary to compute a specific dictionary in question is incredibly high. Further, increasing the hash’s complexity through a salt is reasonable, as unmasking a single hash would otherwise lead to the exploitation of the corresponding dictionary. Hence, not only a registered phone number but its entire address book would be exposed.

4.2 Comparison

Compared with the contact discovery methods of other state-of-the-art mobile messaging applications, none of the popular applications meets all our criteria.

TABLE 2 Privacy Protection Overview (Kales et al., 2019)

	WhatsApp	**Telegram**	**Signal**	**Threema**	**paired**	**procd**
Processes phone numbers in plaintext	x	x	-	-	-	-
Processes hashes of contacts	-	-	x	x	x	x
Uses Salted hashes to prevent dictionary attacks	-	-	-	-	-	x
Relies on trusted hardware	-	-	x	-	-	-
Cost to unmask a single phone number (self-discovery)	$0	$0	$10	$10	$10	$276
Cost to unmask German numbers(10^6)	$0	$0	$0.01	$0.01	$52	$262M

5 Conclusion

We introduced a new unique method (pairwise phone number hashing) for private contact discovery and increased our robustness against dictionary attacks effectively using a common shared secret. Additionally, we implemented a mechanism to retrieve a public key to initiate communication. With our procd approach, we successfully improved unbalanced private set intersections for the specific use case of contact discovery.

Our experiment shows that a trustless private contact discovery design is possible, and no exchange of plaintext data is needed.

6 References

Bošnjak, L., Sres, J., & Brumen, B. (2018). *Brute-force and dictionary attack on hashed real-world passwords*. https://doi.org/10.23919/MIPRO.2018.8400211

Cimpanu, C. (n.d.). New Platypus attack can steal data from Intel CPUs. *ZDNet*. https://www.zdnet.com/article/new-platypus-attack-can-steal-data-from-intel-cpus/

Evans, Vladimir. Rosulek, M., David. Kolesnikov. (2018). *Pragmatic Introduction to Secure Multi-Party Computation.* NOW PUBLISHERS INC.

Ihle, C., Schubotz, M., Meuschke, N., & Gipp, B. (2020). A First Step Towards Content Protecting Plagiarism Detection. *Proceedings of the ACM/IEEE Joint Conference on Digital Libraries in 2020*, 341–344. https://doi.org/10/ghg7rw

Kales, D., Rechberger, C., Schneider, T., Senker, M., & Weinert, C. (2019). *Mobile Private Contact Discovery at Scale* (No. 517).

Lee, K., Kaiser, B., Mayer, J., & Narayanan, A. (2020). An empirical study of wireless carrier authentication for SIM swaps. *Sixteenth Symposium on Usable Privacy and Security (SOUPS 2020)*, 61–79. https://www.usenix.org/conference/soups2020/presentation/lee

Li, N. (2010). Research on Diffie-Hellman key exchange protocol. *2010 2nd International Conference on Computer Engineering and Technology*, *4*, V4–634–V4–637. https://doi.org/10/bdtfv3

Marlinspike, M. (2017). Technology preview: Private contact discovery for Signal. *Signal Messenger*. https://signal.org/blog/private-contact-discovery/

Yanai, A. (2020). *Private Set Intersection*. https://decentralizedthoughts.github.io/2020-03-29-private-set-intersection-a-soft-introduction/