Implementation and Analysis of the Apple PSI System [BBMT21]

Alessandro Baccarini
anbaccar@buffalo.edu
University at Buffalo

December 6, 2021

Abstract

In this project, we implement a proof-of-concept version of Apple's private set intersection (PSI) system in Python. We modify the the protocols to use simpler cryptographic primitives, and demonstrate its correctness.

1. Introduction

Earlier this August, Apple unveiled its novel Child Sexual Abuse Material (CSAM) detection system[1] to inhibit the spread of CSAM and aid law enforcement in pursuing criminals. The framework is designed to automatically detect known CSAM images stored in iCloud Photos, and allow Apple to report the offending users to the National Center for Missing and Exploited Children (NCMEC). The system would be present on all US-based iPhone and iPad devices running iOS 15 and iPadOS 15, respectively, with the goal of adding support to macOS and watchOS in the future. The announcement received significant backlash from media, the tech community, and security researchers, stating that this system is effectively a backdoor an authoritarian government can leverage to conduct censorship on any material deemed "inappropriate." However, on September 8, 2021, Apple ultimately decided to delay the rollout of this feature citing feedback from "customers, advocacy groups, and researchers." No projected release date has been provided at the time of writing.

Apple's CSAM detection system consists of two major components:

NeuralHash: a perceptual hashing function for images based on neural networks and produces a "fingerprint" of the input. The algorithm passes the input image into a convolutional neural network to produce an N-dimensional floating point descriptor, which are hashed using Hyperplane LSH (Locality Sensitivity Hashing) to produce an M-bit value as the NeuralHash of the image. Unlike a standard cryptographic hash function (such as MD5 and the SHA family), NeuralHash is insensitive to small perturbations of the input image, such as cropping and pixel

[1] https://www.apple.com/child-safety/

inversion. The publicly available reverse-engineered version of Apple's NeuralHash algorithm has been shown to not be collision resistant [Ygv21]. Apple claims a proprietary server-side algorithm is run to verify the results [CFBC].

Threshold PSI: The Private Set Intersection (PSI) system, proposed in [BBMT21], is constructed as follows: First, the database CSAM hashes is stored on the client's device. The device generates a voucher for a client's input image that encodes a hash of the image and user's decryption key. The voucher is subsequently uploaded to iCloud (the server) to confirm the matching. Threshold secret sharing is leveraged to ensure the user's decryption key cannot be recovered without meeting some predefined "threshold." If this threshold is crossed, Apple can retrieve the decryption key for the flagged user's photos, manually verify that they in fact CSAM images, and report the user to NCMEC.

The goal of this project is implement and verify Apple's protocol using open-source software and standard cryptographic libraries on a small-scale dataset. The paper is organized as follows: we formally introduce the problem and relevant notation in Section 2. Section 3 contains the necessary cryptographic primitives for the complete protocol description in Section 4. We describe and discussion our implementation in Section 5, and conclude in Section 6

2. Threshold PSI with associated data

We provide a brief overview of the authors' warmup problem denoted **threshold PSI with associated data**, or **tPSI-AD**. We refer to Table 1 for all the relevant notation we use.

Suppose the server has a set $X \subseteq \mathcal{U}$ of size n, and the client has an unordered list of m triples:

$$\bar{Y} = \left(\left(y_1, id_1, ad_1 \right), \ldots \left(y_m, id_m, ad_m \right) \right) \in \left(\mathcal{U} \times \mathcal{ID} \times \mathcal{D} \right)^m ,$$

where $y \in \mathcal{U}$ is the hash of an image, a unique identifier $id \in \mathcal{ID}$, and some associated data $ad \in \mathcal{D}$. Identifiers are not assumed to be secret, and are treated as freshly sampled independent random bit strings. We introduce a threshold parameter t known to both the client and server For tPSI-AD, we aim to design a protocol such that at termination, (i) the client only learns $|X|$, (ii) the server only learns $\bar{Y}_{id}$, and

- if $\left| id \left(\bar{Y} \cap X \right) \right| < t$ then the server learns $id \left(\bar{Y} \cap X \right) \subseteq \mathcal{ID}$, or
- if $\left| id \left(\bar{Y} \cap X \right) \right| \geq t$ then the server learns $\bar{Y} \left[id \left(\bar{Y} \cap X \right) \right]_{\{id, ad\}} \subseteq \left(\mathcal{ID} \times \mathcal{D} \right)$.

Namely, if the threshold is not met, the server learns the identifiers in $id \left(\bar{Y} \cap X \right)$, but no associated data. Otherwise, the server learns $\bar{Y} \left[id \left(\bar{Y} \cap X \right) \right]_{\{id, ad\}}$, which is the associated data for all the identifiers in the intersection.

Naturally, the protocol should satisfy the following security goals:

- The server cannot recover the user's matched photos without exceeding the threshold t.
- False positives are impossible.
- The server learns no information is learned about the client's non-matched images.
- The user cannot learn any information about X aside from its size.
- The user cannot identify which of their images were flagged as CSAM by the system.

Symbol	Meaning
$\mathcal{U}$	Universe of all possible image hash values
$X \subseteq \mathcal{U}$	Set of distinct hash values the server has, s.t. $\lvert X \rvert = n$.
$\bar{Y} = ((y_i, id_i, ad_i))$	Triples the client has, s.t. $\lvert \bar{Y} \rvert = m, i \in [1, m]$.
$y \in \mathcal{U}$	Hash value
$id \in \mathcal{ID}$	Unique identifier of a triple
$ad \in \mathcal{D}$	Associated data of a triple
$id(\bar{Y})$	Set of id's of triples in $\bar{Y}$
$id(\bar{Y} \cap X)$	Set of id's of triples in $\bar{Y}$ whose y is also in X
$\bar{Y}_{id} \in \mathcal{ID}^m$	List of all id's in the triples in $\bar{Y}$
$\bar{Y}_{id,ad} \subseteq (\mathcal{ID} \times \mathcal{D})$	Set of id's and ad's in the triples in $\bar{Y}$
$\bar{Y}[T] \subseteq (\mathcal{U} \times \mathcal{ID} \times \mathcal{D})^{\leq m}$	The list of triples in $\bar{Y}$ whose id's are in $T \subseteq \mathcal{ID}$
$\bar{Y}\left[id\left(\bar{Y} \cap X\right)\right]_{\{id,ad\}} \subseteq (\mathcal{ID} \times \mathcal{D})$	Set of associated data for all id's in intersection
$x = d$	Assignment of value d to variable x
$x \leftarrow A(\cdot)$	x is the output of a randomized algorithm A

Table 1: PSI notations.

We also assume both the client and server honestly adhere to the protocol as described. We refer the reader to §4.4 of [BBMT21] for a complete, concrete security description, including proofs of robustness against malicious clients and servers.

An additional security goal of masking the number of matches a client has accumulated until the threshold is exceeded can be achieved through the fuzzy version of tPSI-AD (aptly named ftPSI-AD). This requires construction of the novel *detectable hash function* primitive, which is beyond the scope of this project.

3. Building Blocks

We define the following cryptographic primitives and their respective constructions below:

- (Enc, Dec) is a symmetric encryption scheme with key space $\mathcal{K}'$ and provides IND-CPA security (see [KL14] §3.4.2 for the full definition) and *random key robustness*, which states that if $k \neq k'$ are independent random keys, then $\mathrm{Dec}(\mathrm{Enc}(k, m), k')$ should fail with high probability. AES128-GCM satisfies both requirements.
- $\mathbb{G}_{\mathrm{DH}}$ is a Diffie-Hellman group of prime order q with G as a fixed generator and Decision Diffie-Hellman (DDH) assumption holds. We use Group 14 with a 2048-bit modulus, and $G = 2$.
- $H : \mathcal{U} \to \mathbb{G}_{\mathrm{DH}}$ is a hash function modeled as as random oracle. This is implemented using HMAC with SHA256, and converting the output digest to an integer $\pmod q$.
- $h : \mathcal{U} \to \{1, \ldots, \eta\}$ is a random hash. This is implemented using SHA256 and converting the output digest to an integer $\pmod \eta$.
- $H' : \mathbb{G}_{\mathrm{DH}} \to \mathcal{K}'$ is secure key derivation function; the uniform distribution on $\mathbb{G}_{\mathrm{DH}}$ mapped to an "almost" uniform distribution on $\mathcal{K}'$. This is implemented using HKDF with SHA256 to produce a 128-bit key.

- Shamir secret sharing on an element of $\mathcal{K}'$ to obtain shares in $\mathbb{F}_{\mathsf{Sh}}$ for some field $\mathbb{F}_{\mathsf{Sh}}$ over a prime Sh; $\mathbb{F}_{\mathsf{Sh}}$ must be sufficiently large such that when choosing $t+1$ random elements from $\mathbb{F}_{\mathsf{Sh}}$, the probability of a collision is low.
- A pseudorandom function (PRF) $F : \mathcal{K}'' \times \mathcal{ID} \to \mathbb{F}_{\mathsf{Sh}}$. This is also constructed using HMAC with SHA256, and converting the output digest to an integer $\pmod{\mathsf{Sh}}$.

Diffie-Hellman Random Self Reducability: We introduce a particularly useful property that the protocol relies upon as follows.

Definition 1. Let $\mathbb{G}$ be a group of prime order q with a fixed generator $G \in \mathbb{G}$, and suppose $(L, U, V) \in \mathbb{G}^3$. Then triple (L, U, V) is a **Diffie-Hellman (DH) tuple** if there exists an $\alpha \in \mathbb{F}_q$ such that $L = G^\alpha$ and $V = U^\alpha$.

We work through the arithmetic of a partial random self reduction for DH tuples below.

Definition 2 (DH self-reduction)**.** Given a triple $(L, T, P) \in \mathbb{G}^3$, we

- choose a random $\beta, \gamma \in \mathbb{F}_q$,
- compute $Q = T^\beta \cdot G^\gamma$ and $S = P^\beta \cdot L^\gamma$.
- output (L, Q, S)

The transformation $(L, T, P) \to (L, Q, S)$ has the following properties:

- If (L, T, P) is a DH tuple where $L = G^\alpha$, then Q is a fresh uniformly sampled element in $\mathbb{G}$, and

$$S = P^\beta \cdot L^\gamma = (T^\alpha)^\beta \cdot (G^\alpha)^\gamma = \left(T^\beta \cdot G^\gamma\right)^\alpha = Q^\alpha.$$

- If (L, T, P) is not a DH tuple, then (Q, S) is a fresh uniformly sampled pair in $\mathbb{G}^2$.

4. Threshold PSI-AD using the DH random self reduction

We now walk through every step up the warm-up tPSI-AD protocol outlined in [BBMT21]. The specific version we are implementing occurs in four phases: S-Init, C-Init, C-Gen-Voucher, and S-Process, where S and C refer to the Server and Client, respectively.

Protocol 1: S-Init(X)

```
Remove any duplicates from $X$, and let $n = |X|$.
Construct a hash table $T$ with $\mathsf{HashTable}(X)$
Choose a random nonzero $\alpha \in \mathbb{F}_q$, compute $L = G^\alpha \in \mathbb{G}_{\mathrm{DH}}$
for $i = 1$ to $n'$ do
    if $T[i]$ is non-empty then
        Set $P_i = H(T[i])^\alpha \in \mathbb{G}_{\mathrm{DH}}$, where $T[i] \in X \subseteq \mathcal{U}$, and $H : \mathcal{U} \to \mathbb{G}_{\mathrm{DH}}$.
    else
        Choose a random $P_i \in \mathbb{G}_{\mathrm{DH}}$.
Set pdata = $(L, P_1, \ldots, P_{n'})$.
Procedure HashTable($X$)
    Let $n' \geq n$ be the size of the table, where $n'$ is sufficiently larger than $n$ as to minimize
      collisions.
    Choose a hash function $h : \mathcal{U} \to \{1, \ldots, n'\}$.
    Insert elements of $X$ into $T$, where each cell should have at most one element.
```

At this point, the server would send each client pdata, along with descriptions of each hash function H, H', and h in the form of 128-bit domain separation nonces. For simplicity in our implementation, we assume the existence of fixed, shared keys between the client and server for the hash functions.

Protocol 2: C-Init()

```
Obtain pdata from the server.
Generate $adkey \leftarrow \mathcal{K}'$ for encryption scheme (Enc, Dec).
Generate $fkey \leftarrow \mathcal{K}''$ for the PRF $F : \mathcal{K}'' \times \mathcal{ID} \to \mathbb{F}_{\mathrm{Sh}}$.
Initialize threshold Shamir secret sharing for $adkey$:

    $$f(x) = a_0 + a_1 x + a_2 x + \cdots + a_{t-1} x^{t-1},$$

  where $a_0 = adkey$ is the secret.
```

This completes the setup procedure. In practice, we can assume this procedure would take place when a user registers their new device with Apple and opts-in for iCloud.

Protocol 3: C-Gen-Voucher(y, id, ad)

```
1 Compute $adct \leftarrow \text{Enc}\left(adkey, ad\right)$, and ensure all $adct$ must be the same length.
2 Compute $x = F(fkey, id) \in \mathbb{F}_{\text{Sh}}$.
3 Generate a share $sh = (x, f(x)) \in \mathbb{F}_{\text{Sh}}$ of $adkey$. This guarantees duplicate triples with the
    same $id$ will produce the same $sh$).
4 Choose a random key $rkey \leftarrow \mathcal{K}'$ and compute $rct \leftarrow \text{Enc}\left(rkey, (adct, sh)\right)$.
5 Compute $w = h(y) \in \{1, \ldots, n'\}$.
6 Sample random $\beta, \gamma \in \mathbb{F}_q$, and use $P_w, L$ from pdata to compute:

        $$Q = H(y)^\beta \cdot G^\gamma \text{ and } S = P_w^\beta \cdot L^\gamma,$$

    where if $y = T[w]$, then $P_w = H(y)^\alpha$ and $S = Q^\alpha$.
7 Compute $ct \leftarrow \text{Enc}\left(H'(S), rkey\right)$, where $H' : \mathbb{G}_{\text{DH}} \to \mathcal{K}'$.
8 Send $voucher = (id, Q, ct, rct)$ to the server.
```

The intuition for Step 6 is that the client is applying the DH random self reduction to the triple $(L, H(y), P_w)$. If $y = T[w]$, then $P_w = H(y)^\alpha$ and (Q, S) satisfies $S = Q^\alpha$. Otherwise, (Q, S) are random elements of $\mathbb{G}_{\text{DH}}$.

Protocol 4: S-Process

```
Initialize empty set SHARES and an empty list IDLIST.
foreach $(id, Q, ct, rct)$ received do
    Append $id$ to IDLIST.
    Compute $\hat{S} = Q^\alpha \in \mathbb{G}_{\text{DH}}$.
    Set $rkey = \text{Dec}(H'(\hat{S}), ct)$.
    Set $(adct, sh) = \text{Dec}(rkey, rct)$.
    If either decryptions “fails”, $y$ is a non-match, and ignore the voucher.
    Otherwise, we found a match and add $(id, adct, sh)$ to SHARES.
Let $t'$ denote the number of unique shares in SHARES, and $t'$ should equal the size of
  $id(\bar{Y} \cap X)$.
if $t' < t$ then
    let OUTSET be the set of identifiers in SHARES.
else if $t' \geq t$ then
    Use $t$ shares to reconstruct $adkey \in \mathcal{K}'$.
    Initialize OUTSET $= \{\emptyset\}$.
    foreach triple $(id, adct, sh) \in$ SHARES do
        compute $ad = \text{Dec}(adkey, adct)$.
        If it fails, discard the voucher. Otherwise, add $(id, ad)$ to OUTSET.
Output IDLIST and OUTSET.
```

Since the original protocol relies on slightly different primitives than the version provided in this paper, we argue its correctness below.

Theorem 1 (Correctness)**.** *Suppose the client and serve honestly adhere to Protocols 1, 2, 3, and 4, and the following assumptions hold:*

(i) $H' : \mathbb{G}_{\text{DH}} \to \mathcal{K}'$ *is a secure key derivation function,*
(ii) (Enc, Dec) *is random key robust, and*
(iii) F *is a secure PRF.*

Then the server learns the required tPSI-AD output with high probability.

Proof Sketch. By construction, the output `IDLIST` is equal to $\bar{Y}_{id}$. We first show the server learns the set $id\left(\bar{Y} \cap X\right)$ as required in tPSI-AD. Let $(y, id, ad) \in \bar{Y}$ be a client triple and (id, Q, ct, rct) be its corresponding voucher. We consider two possibilities:

- (Case 1) If $id \notin id\left(\bar{Y} \cap X\right)$, then we know with high probability the element $H(y)^{\alpha} \in \mathbb{G}_{\text{DH}}$ is not equal to the point $P_{h(y)}$ in `pdata`, and therefore $(L, H(Y), P_{h(y)})$ is not a Diffie-Hellman tuple. The client creates the ciphertext ct using a key $H'(Q^{\alpha})$, and the pair (Q, S) are random elements of $\mathbb{G}_{\text{DH}}$ by the Diffie-Hellman random self reduction. The keys $H'(S)$ and $H'(Q^{\alpha})$ are therefore random keys in $\mathcal{K}''$, and thus via random key robustness of (Enc, dec), the server would fail to decrypt ct.
- (Case 2) If $id \in id\left(\bar{Y} \cap X\right)$, then $(L, H(Y), P_{h(y)})$ must be a Diffie-Hellman tuple, and the server would succeed in decrypting ct to learn $rkey$ of the voucher.

Now, suppose $id\left(\bar{Y} \cap X\right)$ contains more than t elements. The server can reconstruct $adkey$ since it has learned all $rkey$'s for every voucher containing a match; it uses $rkey$'s to reveal the pairs $(adct, sh)$, in the process retrieving sufficient shares sh of $adkey$ to perform reconstruction. Once $adkey$ has been recovered, the server decrypts all $adct$'s that correspond to identifiers in $id\left(\bar{Y} \cap X\right)$, hence revealing $\bar{Y}\left[id(\bar{Y} \cap X)\right]_{\{id,ad\}}$ as required. □

Remark 1. One of the major discrepancies with the system is how it handles *duplicate hashes*. The authors consider it a possibility for the existence of triples in $\bar{Y}$ with the same hash y, provided they have different corresponding id's. More specifically, it is valid for a $\bar{Y}$ to contain the triples $t_i = (y_i, id_i, ad_i)$ and $t_j\ (y_i, id_j, ad_i)$ for $i \neq j$. If a client submits a voucher derived from t_j after previously submitting t_i, we are guaranteed to treat t_j as unique match with a valid share of $adkey$, since x is derived from id, namely $x = F(fkey, id)$. This implies a client inputting multiple copies of the same violating image on their device may exceed the threshold t as if every picture was unique. This is particularly notable considering how easy it is to produce collisions using NeuralHash (see [Ath21]). The authors may consider this an acceptably property of the system, but is nonetheless worth noting.

Remark 2. The original protocol uses Cuckoo tables instead of regular hash tables. We experimentally determined that $|T| = n'$ must be at least twice as large as n to provide some collision resistance, namely $n' = 2^{\lceil \log n \rceil + 1}$. Cuckoo tables can achieve perfect collision resistance at a much lower space cost, namely $n' = (1 + \epsilon')n$ constant $\epsilon \geq 0$.

5. Implementation and Discussion

We implement the complete protocol in Python. We opted to forego any network-related functionality in place of a local, single-instance version. We feel examining the protocol itself is more interesting than an online interactive component. The following libraries, functions, and reference code are present in the implementation:

- `cryptography` – contains AES128-GCM, HKDF, and SHA256 implementations.
- `hmac`, `hashlib` – contains HMAC implementation (with alternate SHA256 object).
- `diffiehellman` – contains primes and generators for multiple Diffie-Hellman groups.
- `os.urandom(n)` – generates random n-bit byte string.
- `random.randint(a,b)` – generates a random integer in the range a, b.
- `hash_table.py` – reference hash table implementation from [Lin21], modified to use a the hash function $h : \mathcal{U} \to \{1, \ldots, n\}$.
- `shamir.py` – basic Shamir secret sharing scheme, where `polynom` and `coeff` are from [Gee21] and `recover_secret` (and its sub-procedures) are from [Wik21].
- `nnhash.py` – computes the NeuralHash of an image [Ygv21].

The prime of $\mathbb{F}_{\mathsf{Sh}}$ is the 13th Mersenne prime $2^{521} - 1$; through experimentation we determined that the 12th Mersenne prime $2^{127} - 1$ was too small, and led to incorrect reconstructions of *adkey*.

Correctness testing was performed on each cryptographic primitive, typically by fixing any randomly seeded values and verifying the output. The entire system was subjected to all possible input configurations (outlined below). Once it was fully operational, we re-introduced randomly generated values and confirmed the correctness.

The implementation is packaged with a small set of server data in `images/` and client inputs in `inputs/`. The program can be run with the following optional arguments:

- `-h, --help` – Displays argument information.
- `-d, --dh_num` – Diffie-Hellman group number, default is 14.
- `-t, --thresh` – Shamir secret sharing threshold $t \geq 2$, default is $t = 3$.
- `-i, --image` – Optional input image, must be placed inside `images/` directory.

By default, the program runs an experiment that demonstrates the following scenario with $t = 3$:

1. The client submits a voucher (y_1, id_1, ad_1) with $y_1 \in X$, produces a match, and updates t';
2. The client submits a duplicate voucher (y_1, id_1, ad_1) with $y_1 \in X$, produces a match, but does not update t';
3. The client submits a voucher (y_1, id_1', ad_1') with $y_1 \in X$ (see Remark 1), produces a match, and updates t';
4. The client submits a duplicate voucher (y_2, id_2, ad_2) with $y_2 \notin X$, does not produce a match, and does not update t';
5. The client submits a voucher (y_3, id_3, ad_3) with $y_3 \in X$, produces a match, surpasses the threshold t, reconstructs *adkey*, and recovers the *ad* of all prior matches;

With respect to performance, the most expensive operations are multiplications and exponentiations modulo a large prime q. All other operations (hashing, encryption and decryption, key generation, etc.) are all relatively lightweight. The S-Init procedure needs only to be completed once for subsequent invocations of C-Gen-Voucher and S-Process. It would need to be re-run in the event of adding new hashes to X, along with redistribution of `pdata` to all clients, but we can safely assume this is infrequent.

6. Conclusion

We have implemented a proof-of-concept version of the tPSI-AD system in Python. Our analysis confirms that even with slightly modified cryptographic primitives, we achieve equivalent security as the original protocol. While the system (and the fuzzy variant) are cryptographically sound, its mere existence inspires ethical concerns. An authoritarian government could "request" the scanning for other material (such as pornography, and political, religious, racial, and LGBTQIA+ material) if Apple wanted to continue selling products in their country. The slippery slope argument is befitting.

References

[Ath21] Anish Athalye. NeuralHash Collider. https://github.com/anishathalye/neural-hash-collider, September 2021. original-date: 2021-08-19T00:06:20Z.

[BBMT21] Abhishek Bhowmick, Dan Boneh, Steve Myers, and Kunal Talwar Karl Tarbe. The Apple PSI System. https://www.apple.com/child-safety/pdf/Apple_PSI_System_Security_Protocol_and_Analysis.pdf, 2021.

[CFBC] Joseph Cox, Lorenzo Franceshi-Bicchierai, and Samantha Cole. Apple Defends Its Anti-Child Abuse Imagery Tech After Claims of 'Hash Collisions'. https://www.vice.com/en/article/wx5yzq/apple-defends-its-anti-child-abuse-imagery-tech-after-claims-of-hash-collisions.

[Gee21] GeeksforGeeks. Implementing shamir's secret sharing scheme in python. https://www.geeksforgeeks.org/implementing-shamirs-secret-sharing-scheme-in-python/, 2021.

[KL14] Jonathan Katz and Yehuda Lindell. *Introduction to modern cryptography*. Chapman and Hall/CRC, 2014.

[Lin21] Marcus Lind. How to create a hash table from scratch in python. https://coderbook.com/@marcus/how-to-create-a-hash-table-from-scratch-in-python/, 2021.

[Wik21] Wikipedia. Shamir's secret sharing python example. https://en.wikipedia.org/wiki/Shamir%27s_Secret_Sharing, 2021.

[Ygv21] Asuhariet Ygvar. AppleNeuralHash2ONNX. https://github.com/AsuharietYgvar/AppleNeuralHash2ONNX, September 2021. original-date: 2021-08-15T19:52:47Z.