Private Set Intersection (PSI)
Malicious security, and amplifying the success probability

Benny Pinkas, Bar-Ilan University
In this lecture

• Malicious security for PSI
• Amplifying the success probability
• PSI conclusions

(many slides by my coauthors)
Template for PSI based on OPRF (previous hour)

\[ x_1, \ldots, x_n \]

\[ y_1, \ldots, y_n \]

\( K \) is a OPRF key known to Bob

\[ F_K(x_1), \ldots, F_K(x_n) \]

\[ F_K(y_1), \ldots, F_K(y_n) \]

Compresses the two lists

PRF of Bob’s inputs
Implementing the OPRF

• The most efficient OPRF implementations are based on OT extension

• Caveat: Secure only as long as client evaluates the OPRF \textit{at most once}

• E.g., when $F_{a,b}(x) = ax+b$
Solution: Hashing

• Suppose both parties use the same public random hash function $h()$ to hash their $n$ items to $n$ bins.
  • Then obviously if Alice and Bob have the same item, both of them map it to the same bin. PSI can then be independently run for each bin.
  • If Bob has a single item in each bin, he only needs to evaluate the OPRF once.

Problem: many bins will have >1 item mapped to them
Using 2 Hash Functions (cuckoo hashing [PR,KMW])

- $h_1, h_2$: item → bin
- Map $n$ items to $(2 + \epsilon)n$ bins
- Each bin stores at most one item!

- Succeeds with very high probability
- If we also have a stash of size $s$, all items $x$ can be mapped to either $h_1(x), h_2(x)$ or the stash, except with probability $O(n^{-(s+1)})$
The Power of Using 2 Hash Functions (Cuckoo)

- $h_1, h_2$: item $\rightarrow$ bin
- Map $n$ items to $(2 + \epsilon)n$ bins
  - Alice – simple hashing
    - $x \rightarrow h_1(x)$ and $h_2(x)$
  - Bob – Cuckoo hashing
    - $y \rightarrow h_1(y)$, or $h_2(y)$
- Caveat: stash size is $\omega(1)$ (let’s ignore it)

Can compare with single-usage OPRF

Alice

Bob

Caveat: stash size is $\omega(1)$ (let’s ignore it)
Combining cuckoo hashing with PSI

• In each bin, Bob (who uses CH) has one item $y$. Alice has $O(\log n)$ items $x_1, x_2, ...$

• In each bin, they run an OT-based OPRF of a function $F$, so that Bob learns $F(y)$, and Alice can compute $F()$ on any input

• Alice sends to Bob the $F()$ values she learned in all bins
• Bob compares them to the values that he learned
Why isn’t this secure against malicious parties?

It turned out that only the following attack is problematic:

• Suppose that both Alice and Bob have a value \( z \)
  • Alice should put \( z \) in bins \( h_1(z) \) and \( h_2(z) \)
  • Bob uses CH and puts \( z \) in only one of these two bins

• Suppose Alice chooses to put \( z \) only in bin \( h_1(z) \)

• Then \( z \) will be in the PSI output iff Bob chose to put \( z \) in \( h_1(z) \)

• This decision of Bob depends on the other inputs that he has

• → The output of the PSI leaks information about other inputs of Bob
The function $F()$ that is used

- $F()$ can be implemented using oblivious transfer extension
- Specifically, a protocol of Orru, Orsini and Scholl, uses OT-extension to implement $F()$, with the following properties
  - The construction is secure against malicious behavior
  - For each table entry $i$, the receiver learns $F_i(x)$, and the sender can compute $F_i()$

- Important for this lecture: A homomorphic property: $F_i(x) + F_j(y) = F_{i+j}(x \oplus y)$. 
PSI from PaXoS, OKVS, and amplification

Relevant papers:

- **Malicious security for OT extension: “PSI from OT”**: Actively Secure 1-out-of-N OT Extension with Application to Private Set Intersection. Michele Orrù and Emmanuela Orsini and Peter Scholl. (CT-RSA 2017)

- **Efficient malicious PSI: PSI from PaXoS**: Fast, Malicious Private Set Intersection. Benny Pinkas and Mike Rosulek and Ni Trieu and Avishay Yanai (Eurocrypt 2020)

- **Even more efficient malicious PSI; amplification of success probability**: Oblivious Key-Value Stores and Amplification for Private Set Intersection. Gayathri Garimella and Benny Pinkas and Mike Rosulek and Ni Trieu and Avishay Yanai. (Crypto 2021)
A different flavor of cuckoo hashing

• Bob is using CH

• Suppose that \( x \) is mapped to locations \( h_1(x)=i \) and \( h_2(x)=j \).

• Unlike CH, Bob puts there values \( V_i \) and \( V_j \) such that \( V_i \oplus V_j = x \).

• Suppose that this mapping is possible, and this property holds for all items that Bob inserts (this is similar to a garbled Bloom filters).
A different flavor of cuckoo hashing

• In the PSI protocol, Bob runs the OPRF in the bins so that he learns $F_i(V_i)$ and $F_j(V_j)$

• Recall the homomorphic property of the function: $F_i(x) + F_j(y) = F_{i+j}(x \oplus y)$

• Therefore Bob can compute $F_i(V_i) + F_j(V_j) = F_{i+j}(V_i \oplus V_j) = F_{i+j}(x)$

• Alice sends to Bob, for each input $y$ of her, $F_{h_1(y)+h_2(y)}(y)$

• Security: Alice cannot cheat by sending just one of $F_{h_1(y)}(y), F_{h_2(y)}(y)$ (this needs a proof)
OKVS Example – Encoding in PaXoS (simplified*)

key-value \(<a, \text{val}(a)>)

Decode(a) = S[1] \oplus S[5] = \text{val}(a)

How do we encode many such keys such that they decode correctly?
OKVS Example – Encoding in PaXoS (simplified*)

Encode keys: a, b, c, d

Peeling:
- c: slot S[2]
- d: slot S[4]
- b: slot S[3]
- a: slot S[1], S[5]

Solving for ‘S’:
- \( S[1] \oplus S[5] = \text{val}(a) \)
- \( S[4] = S[1] + \text{val}(d) \)

Recursively find slots constrained by just one key

Does encoding always work?
What happens when peeling fails?

• The **2-core** of a graph is the maximum subgraph where each node has degree at least 2
  • Namely, the subgraph containing all cycles, as well as all paths connecting cycles.

• All values (edges) which are **not** in 2-core can be handled via peeling
  • But, peeling does not work on the 2-core


\[ S = \begin{align*}
T &
\begin{array}{c}
S[1] \\
S[2] \\
S[3] \\
S[4] \\
S[5]
\end{array}
\end{align*} \]
What happens when peeling fails?

• THM*: For a CH graph of size $O(n)$, WHP the 2-core of size $O(\log n)$ 😊

• In other words, the encoding the we suggested can handle all but $O(\log n)$ of the items mapped to the CH
  • Handling only $O(\log n)$ items should be efficient
  • But we must hide which items these are
What do we actually need?

• An “Oblivious Key-Value Store” (OKVS)

• Key-Value Store:
  • Encode a set of (key, value) pairs. Querying an encoded key returns the corresponding value.

• Oblivious Key-Value Store (OKVS):
  • Hide the keys!
    • A query for an encoded key k will return the corresponding value
    • A query for a key which is not encoded will return a random value
    • Suppose all encoded values in (key, value) pairs are random
    • These two options will be indistinguishable for those making the queries

• This is a recurring requirement in PSI protocols [CDJ16, KMP+17, PSTY19, PRTY19, KRTW19]...
Oblivious Key-Value Store (OKVS)

\[(K, V)\]

\[(key_1, value_1)\]
\[(key_2, value_2)\]
\[(key_3, value_3)\]
\[\vdots\]
\[(key_n, value_n)\]

\[S\]

\[encode\]

\[decode\]

if values are random; then S hides encoded keys; hence oblivious

\[key^* \notin K \rightarrow \text{"don’t care"}\]
Properties of OKVS

\[
\begin{bmatrix}
K & V = (k_1, v_1), (k_2, v_2), ..., (k_n, v_n)
\end{bmatrix}
\]

\[
S^T = \begin{bmatrix}
\end{bmatrix}
\]

Linear OKVS: if values are in \( F \), use decoding function \( d : K \rightarrow F^m \)

Binary OKVS: special case where \( d(k) = \{0, 1\}^m \)

\[
\begin{bmatrix}
- & d(k_1) & - \\
\vdots & \vdots & \vdots \\
- & d(k_n) & -
\end{bmatrix} \times
\begin{bmatrix}
s_1 \\
s_2 \\
\vdots \\
s_m
\end{bmatrix} =
\begin{bmatrix}
v_1 \\
v_2 \\
\vdots \\
v_n
\end{bmatrix}
\]

OKVS efficiency measures

Size: \( \frac{n}{m} \) (optimal = 1)

Encoding time: solve for ‘\( S \’

Decoding time: matrix mult

A must for the PSI constructions we saw.
OKVS Examples - PaXoS

OKVS efficiency measures for PaXoS

• The memory is linear in n
• Encoding time and decoding time are linear
• But cannot encode all items – failure for those which happen to be in the 2-core
OKVS Examples - Polynomial

\[(K, V) = (k_1, v_1), (k_2, v_2), \ldots, (k_n, v_n)\]

\[\text{encode} \quad S(x) = s_1 + s_2 x^1 + s_3 x^2 + \ldots + s_n x^{n-1}\]

\[
\begin{bmatrix}
1 & k_1 & k_1^2 & k_1^3 & k_1^4 \\
\vdots & \ddots & \ddots & \ddots & \ddots \\
1 & k_6 & k_6^2 & k_6^3 & k_6^4 & k_6^5
\end{bmatrix}
\times
\begin{bmatrix}
s_1 \\
s_2 \\
s_3 \\
s_4 \\
s_5 \\
s_6
\end{bmatrix}
= 
\begin{bmatrix}
v_1 \\
v_2 \\
v_3 \\
v_4 \\
v_5 \\
v_6
\end{bmatrix}
\]

\[S = [s_1 \ s_2 \ s_3 \ \ldots \ s_n]^T\]

OKVS efficiency measures
Linear (optimal) size
Encoding time and Decoding time
= \(O(n \log^2 n)\) field operations (FFT)
Pick a random matrix of size \((n \times m)\) of field elements (row corresponding to key is defined as \(H(\text{key})\))

\[
\begin{bmatrix}
  r_1 & r_2 & r_3 & r_4 & \cdots & r_m \\
  \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\
  s_1 & s_2 & s_3 & s_4 & \cdots & s_m \\
\end{bmatrix}
\times
\begin{bmatrix}
  s_1 \\
  s_2 \\
  s_3 \\
  s_4 \\
  \vdots \\
  s_m \\
\end{bmatrix}
= 
\begin{bmatrix}
  v_1 \\
  v_2 \\
  v_3 \\
  v_4 \\
  \vdots \\
  v_n \\
\end{bmatrix}
\]

**OKVS efficiency measures**

- Size is linear
- Encoding time = \(O(n^3)\)
- Decoding time = \(O(n^2)\)

\[
\Pr[\text{Bad event: random matrix has linearly dependent rows}] < |\mathcal{F}|^{n-m-1}
\]

Binary OKVS: \(d(k) = \{0, 1\}^m\) need \(m \geq n + \lambda - 1\) for error probability \(2^{-\lambda}\)
Handling the 2-core in PaXoS

• Suppose I know in advance that whp $|2\text{-core}| = O(\log n)$
• We can encode these $\log n$ items using a less efficient OKVS, e.g. a random matrix
• Advantage: This requires only $\log n + \lambda$ variables to encode $\log n$ values. Total OKCS size is $O(n) + O(\log n) + \lambda$
• Encoding takes $(\log n + \lambda)^3$ time, but this is fine.
The full solution (read on your own)

• The parties agree on adding $O(\log n) + \lambda$ variables, and a random mapping $H()$ to subsets of these variables.

• The value of each input $x$ is defined as the sum of the values of the two locations to which it is mapped in the CH, and the random subset $H(x)$ of the additional variables to which it is mapped.

• Bob maps his $n$ inputs to a CH of size $(2 + \epsilon)n$

• Bob does peeling and remains with a 2-core of size $O(\log n)$

• Bob sets random values to the nodes in the 2-core, but solves equations with the remaining $O(\log n) + \lambda$ variables, to ensure that the values of items in the 2-core is correct.

• Bob reverses the peeling to set values to nodes, ensuring the right values to all remaining variables.

• Bob uses the OPRF to learn a value from each bin, sums them up, and learns $F(x)$ for all inputs.

Expensive: $O(\log n + \lambda)^3$

Cheap: $O(n)$
What are **concrete** parameters for OKVS?

**Theorem:** If $\Pr[|2\text{-CORE}| \geq 0(\log n)] \leq \epsilon$; $|S| = O(n) + O(\log n) + \lambda$
we can encode successfully with negligible error $\epsilon + 2^{-\lambda}$

PaXoS[PRTY20]: Table size $|S| = 2.4n$ (heuristic), $\Pr[\text{Encoding Fails}] = 2^{-40}$

**The elephant in the room** (for many PSI results): Rigorous analysis to translate the asymptotic theorem to concrete parameters ??
What are concrete parameters for OKVS?

Theorem: If $\Pr[|2\text{-CORE}| \geq 0(\log n)] \leq \epsilon$; $|S| = O(n) + O(\log n) + \lambda$
we can encode successfully with negligible error $\epsilon + 2^{-\lambda}$

[Wal21a] 3-cuckoo hash $\rightarrow |S| = 1.23n$ (empirically extrapolated)

Empirical confidence? How can we claim, with 0.9999 confidence, that Except with probability $2^{-40}$ can we encode using 3-cuckoo hashing “1 M keys into 1.3 M bins with less than 10 keys in 2-CORE”?

Simulation is very resource intensive!!

What if the application needs failure probability $2^{-80}$?
Using probabilistic constructions for PSI

• Hashing is a probabilistic process
  • Sometimes it fails. In systems this results in higher overhead (not a big deal).

• For PSI, a hashing failure results in either
  • Inaccurate output (based on a subset of the original input set), or
  • Information leakage

• For some applications this does not matter much
  • ML?
  • CSAM detection (false negatives are fine)
Using probabilistic constructions for PSI

• For a theoretical analysis, we want a negligible failure probability (smaller than any polynomial function)

• For a concrete analysis we want the failure probability to be, e.g., $< 2^{-40}$

• Typically, cuckoo hashing constructions have a very sharp threshold 😊
  • E.g., cuckoo hashing with 3 hash functions succeeds when $|\text{Table}| > 1.23n$

• But there is no tight analysis of the failure probability 😞
  • E.g., if the table is of size $1.3n$, what’s the probability of failure?

• Solutions?
  Heuristics; experiments (costly); amplification of success probability.
Using probabilistic constructions for PSI

Things to note:

• Typically, cuckoo hashing constructions have a very sharp threshold
  • So, in practice, by using a slightly larger hash table, hashing should work well

• The failure probability is a function of the input size
  • For small inputs, failure probability might be too large 😞
  • E.g., a failure probability of $O(n^{-3})$ (what constants?) might not be sufficiently small when $n=1,000$
New approach: amplification

We can very efficiently verify statements about large failure probabilities: E.g., that with 0.9999 confidence, it holds that 3-cuckoo hashing can encode

“1k keys into 1.3k bins/slots with less than 10 keys in 2-CORE”

with failure probability $< 2^{-15}$

Main idea

Compose empirically verified “smaller” OKVS instances into “larger” OKVS provably amplifying the correctness guarantee from $2^{-15}$ to say, $2^{-40}$
Star architecture

• 4 OKVS instances, each large enough to encode $n/3$ items, with failure probability $p$

• A hash function $H()$ which maps items to one of $T_1,T_2,T_3$. 

![Star architecture diagram]

- $T_0$: An additional OKVS of size $(n/3)$
- $T_1$: OKVS size $(n/3)$
- $T_2$: OKVS size $(n/3)$
- $T_3$: OKVS size $(n/3)$
Star architecture - decoding

• Given a key $k$, read its values from tables $T_0$ and $T_{H(k)}$ and return the XOR of these results: $\text{Decode}(k) = \text{Decode}(T_0, k) \oplus \text{Decode}(T_{H(k)}, k)$
Star architecture - encoding
(The success of encoding into a table is a function of the keys, not the values)

- If encoding succeeds for all of $T_1, T_2, T_3$, then
  - Fill random values in $T_0$.
  - Insert values to $T_1, T_2, T_3$, such that decoding succeeds: for all $k$, insert to $T_{H(k)}$ the value $\text{Decode}(T_0, k) \text{ XOR value}(k)$

An additional OKVS of size $n/3$

```
T_0
  An additional OKVS of size(n/3)
    T_1
      OKVS size(n/3)
    T_2
      OKVS size(n/3)
    T_3
      OKVS size(n/3)
```
Star architecture - encoding

• If encoding succeeds for $T_1, T_2$ but not for $T_3$, then
  • Fill random values in $T_3$.
  • Insert values to $T_0$, such that decoding succeeds for items mapped to $T_3$: for $k$ mapped to $T_3$, insert to $T_0$ the value $\text{Decode}(T_3, k) \oplus \text{value}(k)$
  • Insert values to $T_1, T_2$, such that decoding succeeds: for all $k$ mapped to $T_1, T_2$, insert to $T_{H(k)}$ the value $\text{Decode}(T_0, k) \oplus \text{value}(k)$

An additional OKVS of size(n/3)

Same if encoding fails for $T_1$ or $T_2$
Star architecture – bad event

• If encoding fails for two tables, then the process fails

• This only happens with probability $\approx \binom{4}{2} \cdot p^2$

• Performance:
  • Size: $1.33 \times$ optimal OKVS
  • To set the parameters, need to verify a failure probability of $p$ (easier)
  • Obtain smaller failure probability $p^2$

• Can generalize to $q$ bins
Concrete parameters for OKVS

Encode $n=10^6$ key-value pairs:

- $Pr[\text{encode fails}] = 2^{-45.05}$
- Encoding time = 2.915 s
- Decoding time = 1.625 seconds

OKVS size($n$) = 161x8622 = 1.388$n$

Size = 8622 slots

$q = 160$ bins, $n=6250$
Further Improvements?

• Can design recursive constructions
• For practical deployments, a single-level construction seems sufficient

• Open question: build a polynomial-size OKVS with a negligible failure probability (polynomial-size amplification of a small OKVS which has a polynomial failure probability?)
Applications of OKVS

Amplified 3H-GCT can replace any random encoding task

Polynomials
✓ Sparse OT extension $\rightarrow$ PSI [PRTY19]
✓ Oblivious Programmable PRF
  ✓ Circuit-PSI [PSTY19, GMRSS21]
  ✓ Private Set Union [KRTW19]
  ✓ Multi-party PSI [KMPRT17]
    new efficient malicious-secure n-PSI

PaXoS
✓ OT-PaXoS PSI [PRTY20]
  ✓ fastest semi-honest 2PSI
  ✓ fastest malicious 2-PSI
  ✓ empirically verified
  ✓ generalize to admit linear OKVS
    new vOLE-OKVS PSI
✓ vOLE-PaXoS PSI [RS21]
Experimental Results

Takeaways while using this to compute PSI on million items:

3H-GCT, 3H-GCT (star-amp) : 1.61x, 1.43x less communication than PaXoS-PSI

malicious : fastest run-time, ~2x faster than PaXoS-PSI
            : faster than [PSTY19] semi-honest PSI
What should we consider when choosing a PSI solution?
Simplicity

• Most cryptographic papers optimize performance, but if you want to use PSI, you would also desire a solution that it is
  • simple to understand and to explain (to your managers)
  • simple to implement

• DH based constructions are much simpler than the constructions based on OT extension + hashing
Using probabilistic constructions for PSI

• What is the concrete failure probability?
• Sometimes a heuristic analysis is fine
• For some applications hashing failures do not matter much
  • ML ?
  • CSAM detection (false negatives are fine)
• The failure probability is a function of the input size
  • For small inputs, failure probability might be too large 😞
What input size should we plan for?

The cost-per-item of PSI for small sets is higher than for larger sets:

- OT extension / VOLE run a preprocessing step using public key operations
  - This is costly if we do only a few hundred OTs
- The hashing failure probability is smaller for larger input sets
  - For smaller sets, obtaining a failure probability of $2^{-40}$ is costly

→ For smaller input sizes, DH might be better than OT-based PSI
What input size should we plan for?

- For smaller input sets, a recent DH-based protocol of Rosulek-Triue (CCS 2021) is best (also has malicious security)

(graph by Mark Rosulek)
How to measure performance?

• What is more important, computation or communication costs?

• Google [IKN+19]: “For the offline “batch computing” scenarios we consider, communication costs are far more important than computation. ... It is much less expensive to add CPUs to a shared network than to expand network capacity.”
How to measure performance?

Apple’s recent CSAM detection system:

• Each photo uploaded from a device is accompanied by a PSI message.
• The additional message size is negligible. Computation (=battery usage) is far more important.
SpOT PSI (Crypto 2019 [PRTY])

![Diagram showing communication versus running time with points labeled as KKRT, ours (fast), ours (low comm), DH-PSI (K-283), and DH-PSI (25519). The communication is in MB on the y-axis and running time is in seconds on the x-axis. There are points labeled with network bps: 10G, 100M, 10M. The equation $n = 2^{20}$ is also shown.]
Security: Semi-honest vs. Malicious

• For PSI, the performance gap between semi-honest and malicious security is very small 😊

• OT-based protocols: $[PRTY20,GPRTY21]$ have the best performance, and almost the same overhead for malicious and semi-honest security

• DH-based protocols: for small sets, the malicious protocol of $[RT21]$ is only 10%-20% slower than the best semi-honest PSI protocol
What should we use?

**DH-based protocols**
- Best performance for small inputs
- Easy to implement and explain
- Can be modified to compute intersection size

**OT-based protocols**
- Much more efficient for larger inputs
- More complicated
- Harder to modify

**PSI + generic MPC protocols**
- Can compute arbitrary functions
- Slower than OT-based
- More complicated
A different model: Outsourced PSI

- “MPC as a service”
- Many users share their data between servers, which a run the MPC.
- A different trust assumption (!) but can be very efficient!