Hardened Dildo.io: A Cryptographically Secure, Usable Matchmaking Service

Rajeev Parvathala, Jack Serrino, Douglas Chen, Siddharth Seethepalli
{rparvat,jserrino,dpchen,sidds}@mit.edu

May 12, 2016

Abstract

The website *Dildo.io* [1] has been used by many MIT students over the last few months as a matchmaking service restricted to students at MIT. However, such a service may have a key security flaw: all the information is centralized, and some central server(s) have access to everyone's preferences. We wanted to set up a cryptographically secure matchmaking service in which no central server has access to anyone's preferences, and we wanted to make it difficult for any client or the central server to learn about preferences. In addition, many solutions to secure matchmaking involve incredibly complex protocols, so we set out to make our system as easy to use as possible.

1 Motivation

In the 21st century, online matchmaking services have become extremely popular in America. From Tinder to Match.com to OkCupid, dozens of services have popped up. However, in each of these services, a centrally controlled server stores information about the matches of its users. Data breaches, like the one that hit AshleyMadison in mid 2015[9], can reveal everyone's private match and preference information to external parties. Thus, there is certainly place in the world for cryptographically secure matchmaking.

The fundamental motivation for our project was to create a system in which no central server has access to information about individuals' preferences. We modeled our service after *Dildo.io* (shown in Figure 1), a popular service started by an East Campus resident in 2016. It is an intra-MIT matchmaking service; in particular, it restricts the set of users to all undergraduate students at MIT and has approximately one thousand users [5]. To mimic this sort of user base, we constructed our system to heavily use MIT's Athena, and Athena's Shared File System known as AFS; because all users who have accounts on Athena can access certain shared resources through AFS, we used this as a base in order to ensure that public keys were accessible and data storage was persistent.

Fundamentally, our goals for inter-user security are as follows:

A Screenshot from Dildo.io

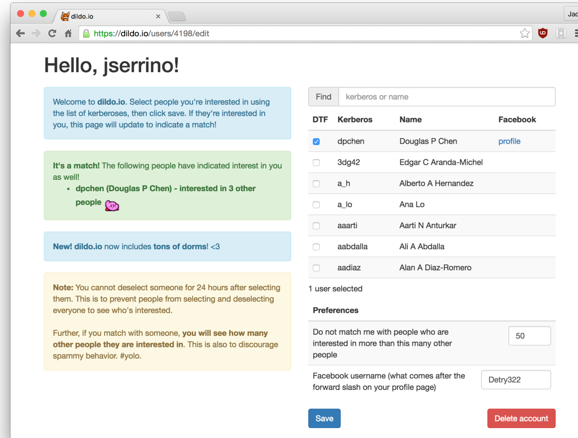

Figure 1: In *Dildo.io*, users can easily search and add preferences for other users. Matches are shown on the left, and current preferences are shown as checks on the right. Although has many features, every user's preferences is available to the server operator.

- Users are able to set preferences for other users, and the service will inform each user if there is a match in preferences.
- Users learn the minimum amount about each other's preferences required to execute the protocol.
- Users should be uniquely capable of presenting match preferences for themselves. The central server or any other party should not be able to forge a "yes" message for a given user.
- If there is any wrongdoer in the system, a user who was subject to malicious actions should be able to prove the wrongdoing, and use this proof to ban the wrongdoer
- There should be no case in which only one user becomes aware of a match or non-match.

In a peer to peer system with no authority controlling message flow between users, all possible protocols have the property that for two users participating in a potential match, one user learns the result before the other and must be trusted to continue execution. This is in direct violation of our security goals, so our protocol uses a server who acts as a sort of escrow for the participating users.

Beyond our security concerns, our secondary goal is to maximize usability and provide an optimal user experience. To achieve this, we limit the round trips between users to one. Each user should be able to get the result of a potential match from the server as soon as both users have submitted preferences.

2 Threat Model

For the initial design of our protocol, we assume an honest-but-curious server, i.e. the server will execute our designed protocol correctly, but may try to learn as much information as possible in the process. Thus the server is not totally malicious: it will not drop messages, refuse to pass along messages, or try to forge messages.

We do not assume anything about the users. Users may be totally malicious: they can send arbitrary messages to other users, attempt to act on behalf of other users/the server, and conduct man-in-the-middle (MITM) attacks. However, we do assume that malicious users do not collude with the server.

Any other interested parties may do any kinds of attacks possible, and may collude with either (1) any subset of the users, OR (2) the central server. For all possible adversaries, we assume probabilistic polynomial computation time bounds since a computationally unbounded adversary would render our public key encryption schemes insecure. Furthermore, even if any other parties collude with users AND the server, they can only learn about preferences and matches submitted by and directed to the users they are colluding with.

We also present an extension of our protocol to defend from malicious servers that incorrectly execute the protocol and/or attempt to forge messages. However, we assume servers still do not collude with clients.

3 Design

3.1 Paillier Encryption Scheme

The Paillier cryptosystem [6] is an additively homomorphic asymmetric encryption algorithm. In particular, for any two plaintexts a and b, $\text{Dec}(\text{Enc}(a)+\text{Enc}(b)) = a + b \mod n$ for n.

For key generation, the Paillier cryptosystem requires two large, random primes p, q. Let $n = pq$ and $\lambda = \text{lcm}(p-1, q-1)$. Then let $g \in \mathbb{Z}^*_{n^2}$, such that $\mu = (L(g^\lambda \bmod n^2)^{-1}) \bmod n$ exists. Then, the key pair is as follows:

- Public key: (n, g)
- Private key: (p, q, μ)

Encryption of a ciphertext is done using the public key, and decryption is achieved using the private key. Paillier encryption is secure against chosen plaintext attack (IND-CPA), although this is not too useful in our protocol since we already add randomness to our plaintexts for other purposes.

We use Paillier encryption to allow the server to perform operations without learning anything about the inputs. By submitting ciphertexts representing preferences, users can allow the central server to perform computation on the ciphertext and return results to involved users, each of which will be able to decrypt the result and learn about the existence of a match.

3.2 Protocol Setup

In our protocol, the central server and all users know a large pre-generated prime k, which is used later in the protocol. The central server and all users each have public/private key RSA pairs for encrypting messages. We assume that the public keys are always available on a trusted shared file system, i.e. AFS in our case. The protocol can also use a certificate authority or other public-key infrastructure. All clients and the server also share a global security parameter λ. This security parameter is used for the length of various large primes, such as in the RSA and Paillier public keys.

3.3 Key Generation

The first step of the protocol is key generation for future matching attempts. These keys are generated when a user joins the *Hardened Dildo.io* system. Each new user A downloads a list of every registered user from the central server, and fetches the registered users' public keys from AFS. For every registered user B, the new user generates three large primes (each λ bits long) $pq = n$ and c for each other user. The primes p and q are used to generate a public, private Paillier keypair $(P, S)_{AB}$. A then encrypts $(S, c)_{AB}$ with B's public key P_B to get $\text{Enc}_{P_B}((S, c)_{AB})$. Finally, A sends $\text{Enc}_{P_B}((S, c)_{AB}), P_{AB}$ to the server. With P_{AB} the server can perform computations on encrypted data for A and B, but only A and B can decrypt the result.

When user B reconnects to the system, they download a list of new users and the corresponding $\text{Enc}_{P_B}((S, c)_{AB})$ values shared between the new users and B from the server. B performs $\text{Dec}_{S_B}(\text{Enc}_{P_B}((S, c)_{AB}))$ to get $(S, c)_{AB}$ and checks that p, q and c are λ bits long, prime and distinct from each other and k. If the values do not meet those conditions, B rejects them and continues as if the new user had not joined. Figure 2 shows a this sign-up process.

3.4 Sending Preferences

In *Dildo.io* users indicate their preferences with a "yes" or "no" for other users. In order to hide from the server which users are sending preferences to which other users, users always submit preferences for all users simultaneously. If a user has not explicitly specified a preference for another user, the *Hardened Dildo.io* client automatically sends a "no" for that user. A matching result of "yes" indicates both users submitted "yes." Users may change their matching preferences by using their client to resubmit all of their preferences simultaneously.

The Sign-up Process

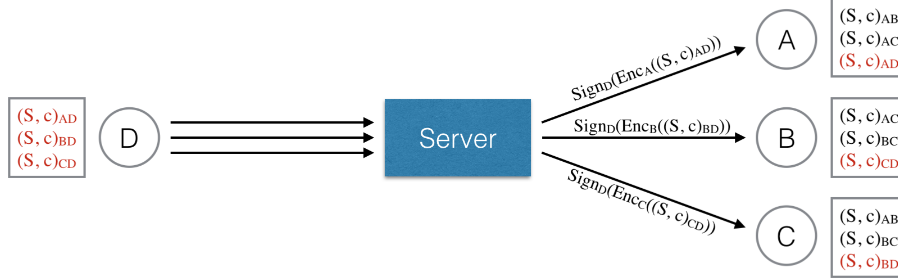

Figure 2: When User D signs up in the protocol, he sends Paillier keypairs for every other user already in the system. New keypairs are shown in red. While new users incur a large start-up cost, incremental work done by existing users is minimal.

Suppose user A wishes to submit preferences. First, for each user B, A retrieves $(S, c)_{AB}$ from the Key Generation portion of the protocol and extracts c. To submit a "yes", user A picks a random integer s (different for each preference submission) in the range $[0, n)$ and computes the decision $d_{AB} = c + sp \mod n$. To submit a "no", user A computes a random $d_{AB} \not\equiv c \mod p$ as his decision instead. User A then encrypts d_{AB} with the Paillier public key n previously shared between A and B and sends $\text{Enc}_{P_{AB}}(d_{AB})$ to the server. To hide which preferences are updated, ciphertexts are sent corresponding to each user. When user B wants to submit a preference for user A, she does the same thing except with different random values. We denote her decision as $d_{BA} = c + sp$ for "yes" and a random $d_{BA} \not\equiv c \mod p$ for "no."

3.5 Matching

Once both users A, B in a pair have submitted their encrypted preferences $\text{Enc}_{P_{AB}}(d_{AB})$ and $\text{Enc}_{P_{AB}}(d_{BA})$ to the server, the server performs matching by computing a random encrypted linear combination of d_A and d_B. In particular, the server computes $m = (r \cdot d_{AB} + (k - r) \cdot d_{BA} \mod n)_{P_{AB}}$ for a random r uniformly selected in the range $[0, n)$. To compute this value efficiently, the server computes $2^x d_{AB} \forall x : 1 \leq 2^x \leq r$ through repeated addition and adds the required values. A similar sequence of steps is taken for d_{BA}. The server then delivers m to A and B if they are online, or enqueues them for later delivery if they are not. These queues are not emptied even if users change their preferences. This discourages a user from sending a "yes" preference and then immediately changing it to a "no" preference in order to determine other users' preferences for them since the other users will be notified of the match and subsequent un-match.

When users A or B receives their encrypted match result m, they decrypt

it to find $t = rd_{AB} + (k - r)d_{BA} \mod n$. If A and B both submitted a "yes" preference for each other, t will be equal to kc modulo p since $r(c + s_A p) + (k - r)(c + s_B p) \equiv rc + (k - r)c \equiv kc \mod p$. Hence, to determine whether there was a match A and B each check whether $t \equiv kc \mod p$. Otherwise, with very high probability t will appear to be a pseudorandom number modulo n, depending on r, s_A and s_B, indicating that there was no match. Furthermore, if A submitted a "no" preference, A will not be able to compute B's preference d_B with very high probability. You can see a timeline of submitted preferences and results in Figure 3.

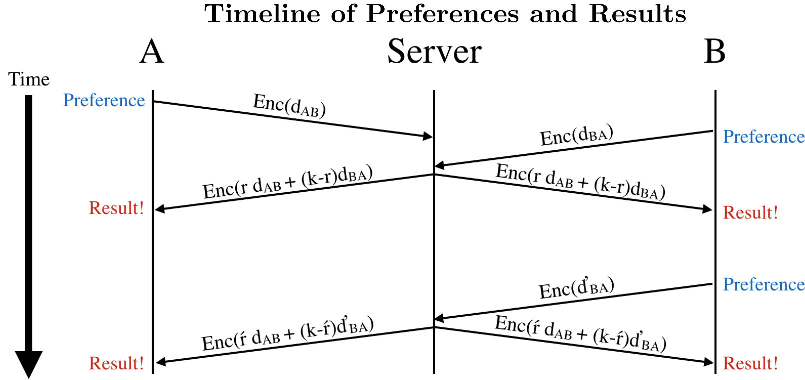

Figure 3: Preferences for a user can be submitted at any time. Once the server has both users' preferences, it computes and sends match results back. From then on, if a user sends another preference, a result can be recomputed easily. If User B changes his preference before User A comes back online, User A will still be notified of both B's preferences.

3.6 Proof of Preference Hiding

We will first show that as long as $d_{AB} \not\equiv d_{BA} \mod p$ and $d_{AB} \not\equiv d_{BA} \mod q$, user A cannot compute user B's preference. We will then show that with very high probability, user A cannot discover user B's preference if user A submits a "no" preference.

We will first consider solving $t = rd_{AB} + (k - r)d_{BA}$ for r modulo p. We have:

$$r \equiv (d_{AB} - d_{BA})^{-1}(t - k \cdot d_{BA})$$

Since p is prime, the inverse $(d_{AB} - d_{BA})^{-1}$ exists if and only if $d_{AB} \not\equiv d_{BA} \mod p$. But by assumption $d_{AB} \not\equiv d_{BA} \mod p$, so the inverse exists and a unique value of $r \mod p$ exists corresponding to the pair (d_{AB}, d_{BA}). Furthermore, changing d_{AB} (but making sure the assumption still holds) always

changes $r \mod p$ as well, since multiplicative inverses are unique. By symmetry, changing d_{BA} also changes $r \mod p$ if we instead think about solving for $k - r \mod p$ rather than $r \mod p$ directly.

An identical argument holds modulo q, since q is also prime and a divisor of n and we have assumed $d_{AB} \not\equiv d_{BA} \mod q$ as well. Then by the Chinese remainder theorem, there exists a $r \mod n$ for each pair (d_{AB}, d_{BA}). Furthermore, for each distinct $d_{BA} \not\equiv d_{AB} \mod p$ the value for r is distinct. Then since r is chosen uniformly at random, the posterior probability distribution for d_{BA} given t and d_{AB} (excluding values equal to d_{AB} modulo p, q) is uniform as well. Hence user A cannot compute user B's preference.

We will now show that if user A submitted a "no" preference, d_{AB} and d_{BA} are not equal modulo p, q with very high probability.

Consider the d_{AB} and d_{BA} modulo p. Suppose user B submitted a "yes" preference. Then d_{AB} and d_{BA} are guaranteed to be different, since otherwise $d_{AB} \equiv c \mod p$, contradicting the assumption that user A submitted a "no." If user B submitted a "no" preference, then $d_{BA} \mod p$ is uniformly random modulo p. In particular, $d_{BA} \mod p$ is unknown to A. The best A can do is guess, with a $1/p \approx 2^{-\lambda}$ probability of correctly guessing a $d_{AB} \equiv d_{BA} \mod p$.

Now consider d_{AB} and d_{BA} modulo q. In the case that user B submits "yes", $d_{BA} = c + s_B p$ is uniformly random modulo q since $\gcd(p, q) = 1$ and s_B is uniformly random over $[0, n)$. If user B submits "no", then d_{BA} is also uniformly random modulo q. In both cases A does not know what $d_{BA} \mod q$ is. Again, user A can only guess with success probability $1/q = 2^{-\lambda}$.

By the union bound, the probability that A can select $d_{AB} \equiv d_{BA}$ modulo p or q without submitting "yes" is no greater than $2 \cdot 2^{-\lambda}$. This is a negligibly small probability. Therefore user B's preference is hidden from user A with very high probability.

4 Extension: Malicious Server

So far when describing the protocol, we have assumed that the server is honest-but-curious. However, this may not always be the case - the server could become malicious through compromise by an adversary or Byzantine failure. We can continue to protect users through a few simple modifications to our protocol. We are primarily concerned about the case in which the server attempts to forge a match when one or more of the participants has sent a "no" preference, as this could have unfortunate consequences for the users. In our original protocol the server can do this fairly easily by guessing if a user submitted a "yes" preference, and using that as the input for both users in the protocol. The case in which the server attempts to forge the absence of a match is not very interesting as the server can always prevent users from matching through a denial of service.

Since users have a secure medium for public key exchange (AFS in our implementation), user message authentication can be accomplished through a simple signature scheme. Message signatures can also be used to prove that a server is behaving badly. This proof can be posted to a public message board or

distribution service to expose the malicious server, and should generally deter adversaries from operating malicious servers. These signatures allow a single client to prove that a server is malicious, but also prevent malicious clients from framing an honest server. For signing, each user and the server has a separate set of RSA key pairs also published on AFS, which we will denote with (P'_A, S'_A) for a user A.

4.1 Key Generation Modifications

Since the purpose of unknown $c \neq 0$ is to prevent server forgeries, this is no longer necessary for the modified protocol. Instead for each other client B, A sends $(\text{Enc}_{P_B}(S_{AB}), \text{Sign}_{S'_A}(\text{Enc}_{P_B}(S_{AB})), P_{AB}, \text{Sign}_{S'_A}(P_{AB}))$ to the server. After B connects, the server sends $(\text{Enc}_{P_B}(S_{AB}), \text{Sign}_{S'_A}(\text{Enc}_{P_B}(S_{AB}))$ to B, who then verifies that the signature is valid before proceeding.

4.2 Sending Preferences Modifications

The preference creation portion of the protocol operates the same as before with the change $c = 0$. However, when A is sending the server his preference for B, he sends $(\text{Enc}_{P_{AB}}(d_{AB}), I_{AB})$ where $I_{AB} = \text{Sign}_{S'_A}(\text{Enc}_{P_{AB}}(d_{AB}))$.

4.3 Matching Modifications

The matching protocol remains identical except that the server sends A the value (m, O_{AB}) where $O_{AB} = \text{Sign}_{S'_{Server}}(m||I_{AB}))$, proving that the server acknowledges the input from A and the result m. User A verifies that the signature presented by the server is valid before proceeding with the rest of the protocol to determine the match result.

4.4 Detecting Malicious Servers

Now suppose user A matches with user B despite submitting a "no" preference, indicating malicious behavior by the server. Then user A can post the shared Paillier private key S_{AB}, his encrypted preference $\text{Enc}_{P_{AB}}(d_{AB})$, the input signature I_{AB}, the output m and the output signature O_{AB} to a public location. Anyone can verify that the server incorrectly output a "yes" by verifying O_{AB} and then decrypting m to find a value equal to $kc \equiv 0 \mod p$. They can also verify that user A correctly submitted a "no" by verifying the input signature I_{AB} against the encrypted preference and checking that A's preference is not $c \equiv 0 \mod p$.

4.5 Protecting Honest Servers

Under this scheme, two malicious clients cannot frame an honest server because the will not be able to generate valid O_{AB}. Since O_{AB} is dependent on both the output m and the input signature I_{AB} (which depends on the input d_{AB}),

O_{AB} shows to any observer that the server computed the correct output from the inputs.

5 Evaluation

To evaluate our protocol, we revisit our design goals.

5.1 Security goals

- Users are able to set preferences for other users, and the service will inform each user if there is a match in preferences

  By executing the protocol, we fulfill this goal.

- Users learn the minimum amount about each other's preferences required to execute the protocol

  With high probability, user A only learns user B's preference for A, if A submitted "yes" for B. Note that this is a property of the matchmaking problem and it is impossible to do better.

- Users should be uniquely capable of presenting match preferences for themselves. The central server or any other party should not be able to forge a "yes" message for a given user

  With the extension mentioned above, the server is prevented from forging a match for two users and preferences are signed with each user's private key, preventing forgery.

- If there is any wrongdoer in the system, a user who was subject to malicious actions should be able to prove the wrongdoing, and use this proof to ban the wrongdoer.

  The extension above allows the users to determine if the server is commiting harmful malicious actions and present a proof to other users.

- There should be no case in which only one user becomes aware of a match or non-match.

  With an honest-but-curious server, the server sends results to both parties simultaneously (or on next login). With a malcious server, we cannot ensure that both parties receive the match result, but we believe this is not possible in general as the distributing party can always legitimately crash during distribution.

5.2 Usability Goals

For a system in which users are only online a very small fraction of the time, a protocol requiring many round trips between users would significantly limit convenience, as two users log in many times over several days before obtaining

the result of a match. We considered this trade-off when opting for our protocol over a multi-party computation scheme requiring multiple round trips that might provide better security guarantees. Since our scheme generates a result immediately after both users have submitted preferences, it performs optimally in this regard.

From a scalablity perspective, our system produces load quadratic in the number of users on the central server and linear in the number of users on each user. Clearly, this imposes strict limitations in terms of the potential scale of the service. However, *Dildo.io* is only available to the student body at MIT, and a *Hardened Dildo.io* server is easily able to serve the needs of a few thousand users. To allow for scaling to larger groups, the protocol will have to be modified, potentially by reducing the number of users to whom each user sends messages.

6 Implementation

On any MIT Athena workstation, one can run the following commands to try our proof of concept.

```
$ add dildo
$ dildo
```

The proof of concept is written in Python, and does not include the Extension from section 4. The source code for our system is available on GitHub[2], and a screenshot of the program is visible in Figure 4.

The User Interface of Hardened Dildo.io

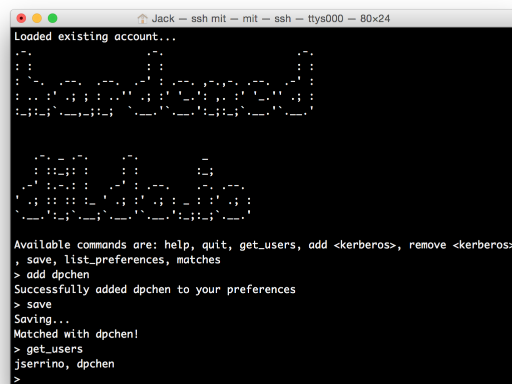

Figure 4: Text-based commands are entered into *Hardened Dildo.io* to `add` and `save` preferences, as well as see matches. Protocol details are entirely hidden from the user, providing better user experience.

Upon first start, it generates a public and private key pair, and makes the public key available on AFS. Then, as a small feature, the user is asked whether or not she wants to encrypt her private local data on AFS. Although this is somewhat outside the scope of our threat model, this prevents an attacker from viewing the user's preferences should the user's Athena account be compromised.

In normal operation, the experience of the system is smooth - one can `add` and `remove` preferences easily via commands, and they only become public upon a `save` command, or program exit. Protocol details are hidden from the user - the downloading/verification of Paillier keys, sending of preferences, and execution of matches are all done upon the execution of a `save`. In addition, the program continually checks for new matches in the background, notifying the user as they occur. Should someone match and unmatch the user in a short period of time, the user is notified - otherwise, unmatches are generally silent.

We believe this proof of concept highlights the usability of our system. Because of the few round-trips of our protocol, users are generally notified immediately of new matches. In addition, since most of the initialization work is done at user registration, regular use of the system incurs small additional cost.

7 Related Work

Several protocols have been developed in the past for cryptographic matchmaking. These protocols focus on determining whether two parties share a secret (called a *wish*), and jointly notifying and authenticating them if they do. However, the parties remain anonymous before they match. For example, a company and a job-seeker might to match based on some shared criteria, which would be encoded into the wish. However, the job-seeker wishes to remain anonymous until the match has happened, to avoid angering his current employer. The first protocol to do this was Baldwin and Gramlich's protocol [3]; however, this protocol is vulnerable to simple man-in-the-middle attacks [11] and requires a fully trusted matchmaker. Meadows published another protocol [4] that only requires a trusted third party during the initialization step when users join, and mutually authenticates two users if their secret credentials the same. More recently, Shin and Gilgor published another protocol [7] that improves on the previous protocols in several respects, including user anonymity and resistance to offline dictionary attacks against wishes.

We could have used matchmaking protocols to implement *Hardened Dildo.io*, using wishes consisting of pairs of user identifiers to encode "yes" preferences and empty wishes to encode "no." However, these protocols (especially Shin and Gilgor's) are very complex because they solve a more general problem than *Hardened Dildo.io*'s simple "yes" and "no" preference matching. They require numerous messages and round trips, which would significantly decrease the usability of our system. Furthermore, anonymity in these protocols is not useful to us because preferences already include the user's identity.

Secure multiparty computation [10] is also related to cryptographic matchmaking. It is a much more general primitive, allowing two or more parties to

compute a function. Each party contributes an input to the function, and all parties receive the output; however, no party learns about other party's input. It is also possible to do secure two-party computation covertly, ie. without notifying the other party that the computation is happening at all unless they are participating [8]. It would be easy to implement *Hardened Dildo.io* using these protocols by computing the logical AND of two users' preferences. However, these protocols also require numerous round trips, and are even more general and difficult to implement compared to cryptographic matchmaking protocols.

8 Conclusion

Our system, *Hardened Dildo.io*, allows users to engage in cryptographically secure matchmaking, providing protection from the regular data breaches that plague similar existing services. In doing so, we take care to maintain similar levels of usability, since users are often willing to trade security for convenience. Our protocol is also highly implementable compared to the complex protocols developed so far and can be integrated into a web service with slight modifications. After it is thoroughly vetted, if it is found to be secure, we hope that our design will be adopted and improved upon by future developers of matchmaking services.

9 Acknowledgments

We would like to thank our TA Kevin King for helping us formulate and refine many aspects of this project. We would also like to thank Professor Ron Rivest and the rest of the 6.857 staff for running a very interesting and practical class. We'd also like to thank Max Justicz for creating the original *Dildo.io*.

References

[1] http://dildo.io.

[2] https://github.com/Detry322/hardened-dildo.

[3] Baldwin, R. W., and Gramlich, W. C. Cryptographic protocol for trustable match making. IEEE, p. 92.

[4] Meadows, C. A more efficient cryptographic matchmaking protocol for use in the absence of a continuously available third party. In *Security and Privacy, 1986 IEEE Symposium on* (1986), IEEE, pp. 134–134.

[5] Navarre, W. 20% of students have used dildo.io. http://thetech.com/2016/04/21/site-to-help-students-find-sexual-partners.

[6] Paillier, P. Public-key cryptosystems based on composite degree residuosity classes. In *Advances in cryptology–EUROCRYPT99* (1999), Springer, pp. 223–238.

[7] Shin, J. S., and Gligor, V. D. A new privacy-enhanced matchmaking protocol. *IEICE Transactions on Communications 96*, 8 (2013), 2049–2059.

[8] Von Ahn, L., Hopper, N., and Langford, J. Covert two-party computation. In *Proceedings of the thirty-seventh annual ACM symposium on Theory of computing* (2005), ACM, pp. 513–522.

[9] Weldon, D. Ashley madison breach shows hackers may be getting personal. `http://www.cio.com/article/2987830/online-security/ashley-madison-breach-shows-hackers-may-be-getting-personal.html`.

[10] Yao, A. C. Protocols for secure computations. In *Foundations of Computer Science, 1982. SFCS'08. 23rd Annual Symposium on* (1982), IEEE, pp. 160–164.

[11] Zhang, K., and Needham, R. A private matchmaking protocol, 2001.