On the performance of certain Private Set Intersection protocols*

Emiliano De Cristofaro[1] and Gene Tsudik[2]

[1] PARC [2] UC Irvine

April 7, 2012

Abstract

Private Set Intersection (PSI) is a useful cryptographic primitive that allows two parties (client and server) to interact based on their respective (private) input sets, in such a way that client obtains nothing other than the set intersection, while server learns nothing beyond client set size. This paper considers one PSI construct from [DT10] and reports on its optimized implementation and performance evaluation. Several key implementation choices that significantly impact real-life performance are identified and a comprehensive experimental analysis (including micro-benchmarking, with various input sizes) is presented. Finally, it is shown that our optimized implementation of this RSA-OPRF-based PSI protocol markedly outperforms the one presented in [HEK12].

1 Introduction

Private Set Intersection (PSI) is a primitive that allow two parties (client and server), to interact on their respective input sets, such that client only obtains the intersection of the two sets, whereas, server learns nothing beyond the size of client input set. PSI is appealing in many real-world settings: common application examples include national security/law enforcement [DT10], Intelligence Community systems [DJL+10], healthcare and genomic applications [BBD+11], collaborative botnet detection techniques [NMH+10], location sharing [NTL+11] as well as cheating prevention in online gaming [BLHB11]. Motivated by practical relevance of the problem, the research community has considered PSI quite extensively and devised a number of techniques that vary in costs, security assumptions and adversarial models, e.g., [FNP04, KS05, HL08, JL09, DSMRY09, DT10, HN10, JL10, DKT10, ADT11]. (Notable PSI protocols are reviewed in Appendix A.)

In this paper, we focus on a specific RSA-OPRF-based PSI protocol from [DT10] that currently offers the most efficient operation. It achieves linear computational and communication complexity and improves overall efficiency (over prior work) by reducing the total cost of underlying cryptographic operations. Although [DT10] actually presents two PSI protocols, this paper focuses on the second – RSA-OPRF-based, in Figure 4 of [DT10] – which is the more efficient of the two. Hereafter, it is referred to as DT10-v4.

Objectives: We discuss our implementation of DT10-v4 and experimentally assess its performance. Our goal is twofold: (1) Identify implementation choices that impact overall protocol performance, and (2) Provide a comprehensive performance evaluation.

Organization: Next section overviews DT10-v4. Then, Section 3 and Section 4 describe, respectively, its implementation and performance evaluation. Finally, performance analysis of our optimized implementation is contrasted with that in [HEK12].

*A shorter version of this report, titled *Experimenting with Fast Private Set Intersection*, appears in the 5th Intl. Conference on Trust&Trustworthy Computing (TRUST 2012) [DT12]. This report include work-in-progress and may be occasionally updated.

2 The DT10-v4 PSI Protocol

We now review the PSI protocol presented in Figure 4 in [DT10], from here on denoted as DT10-v4. First, we introduce some notation, present actual construction, and, finally, discuss settings where (server-side) precomputation is possible/recommended.

2.1 Notation

Notation used in the rest of this paper is reflected in Table 1 below:

$a \leftarrow A$	variable a is chosen uniformly at random from set A
τ, τ'	security parameters
p, q	safe primes
$N = pq, e, d$	RSA modulus, public and private exponents
$H(\cdot)$	full-domain hash function $H : \{0,1\}^* \rightarrow \mathbb{Z}_N^*$
$H'(\cdot)$	cryptographic hash function $H' : \{0,1\}^{\tau_1} \rightarrow \{0,1\}^{\tau'}$
$\mathcal{C}, \mathcal{S}$	client's and server's sets, respectively
v, w	sizes of $\mathcal{C}$ and $\mathcal{S}$, respectively
$i \in [1, v],\ j \in [1, w]$	indices of elements of $\mathcal{C}$ and $\mathcal{S}$, respectively
c_i, s_j	i-th and j-th elements of $\mathcal{C}$ and $\mathcal{S}$, respectively
hc_i, hs_j	$H(c_i)$ and $H(s_j)$, respectively

Table 1: Notation

2.2 Protocol Specification

Figure 1 shows the operation of DT10-v4 below.

Client, on input $\mathcal{C} = \{c_1, \ldots, c_v\}$		**Server**, on input $p, q, d, \mathcal{S} = \{s_1, \ldots, s_w\}$
$\forall_i = 1, \ldots, v$:		$\forall_j = 1, \ldots, w$:
(1) $r_i \leftarrow \mathbb{Z}_N$		(1) $ks_j = (hs_j)^d \bmod N$
(2) $\mu_i = hc_i \cdot {r_i}^e \bmod N$		(2) $ts_j = H'(ks_j)$
(3)	$\{\mu_1, \ldots, \mu_v\} \rightarrow$	(3)
		$\forall_i = 1, \ldots, v$:
		(4) $\mu'_i = (\mu_i)^d \bmod N$
(4)	$\leftarrow \{\mu'_1, \ldots, \mu'_v\}$ / $\{ts_1, \ldots, ts_w\}$	
$\forall_i = 1, \ldots, v$:		
(5) $kc_i = \mu'_i / r_i \bmod N$		
(6) $tc_i = H'(kc_i)$		
(7) If $\exists_j$ s.t. $tc_i = ts_j$ **output** $c_i \in \mathcal{C} \cap \mathcal{S}$		

Figure 1: DT10-v4 executes on common input: $N, e, H(\cdot), H'(\cdot)$.

Correctness: If $c_i \in \mathcal{C} \cap \mathcal{S}$, then $\exists_j$ s.t.: $kc_i = \mu'_i / r_i = (hc_i \cdot {r_i}^e)^d / r_i = {hs_j}^d = ks_j \Longrightarrow tc_i = ts_j$.

Security: DT10-v4 is proven secure in the presence of semi-honest adversaries, under the One-More-RSA assumption [BNPS03] in the Random Oracle Model (ROM) – see [DT10] for details. The proof in Appendix B of [DT10] actually achieves one-side (adaptive) simulation in the ideal-world/real-world

paradigm.[1] Thus, security of DT10-v4 may actually hold in the presence of a malicious client and a semi-honest server. Further, security against a malicious server also seems easy to obtain: RSA signatures have the desirable property of verifiability, thus, client can easily verify server's adherence to the protocol with respect to the computation of $\mu_i' = (\mu_i)^d \bmod N$. Also, client's message to server (i.e., the first round) does not depend on any information from latter, which, in fact, produces no output. However, server would need to prove that its RSA parameters are generated correctly, and it could do so using, for example, techniques from [CM99] or [HMRT11]. Nonetheless, we leave as part of future work formal proofs for malicious security of DT10-v4.

Communication Complexity: DT10-v4 communication complexity amounts to $2v$ group elements and w hash outputs. Specifically, in the first round, client sends v elements in $\mathbb{Z}_N$, whereas, in the second, server transfers v elements in $\mathbb{Z}_N$ and w outputs of $H'(\cdot)$. For 80-bit security, SHA-1, which has 160-bit outputs, may suffice.

Computational Complexity: We note that server workload can be dramatically reduced if exponentiations $(\cdot)^d \bmod N$ are optimized using the ***Chinese Remainder Theorem*** (CRT)[2], since server knows factorization of N. Specifically, DT10-v4's computational complexity is as follows. *Server* computes: w full-domain hashes; $2w + 2v$ modular exponentiations with $(|N|/2)$-bit exponents and $(|N|/2)$-bit moduli (using CRT); w invocations of $H'(\cdot)$. *Client* computes: v full-domain hashes; v exponentiations with $|e|$-bit exponent and $|N|$-bit modulus (in practice, one can select $e = 3$); v modular inverses of $|N|$-bit integers modulo $|N|$ bits; $2v$ modular multiplications of $|N|$-bit integers modulo $|N|$ bits; v invocations of $H'(\cdot)$. Thus, on server side, computational complexity is dominated by $O(w + v)$ CRT exponentiations, whereas, client's computation is dominated by $O(v)$ modular multiplications and inverses. Since client does not perform *any* expensive cryptographic operation (i.e., no modular exponentiations), DT10-v4 is particularly suited for scenarios where *client runs on a resource-poor device*, e.g., a smart-phone.

2.3 Precomputation

One beneficial feature of DT10-v4, as well as some other PSI techniques in [HL08], [JL09], [JL10], is that server computation over its own input does not depend on any client input. Therefore:

1. Server does not need to wait for client to perform its w exponentiations to compute $ks_j = H(hs_j)^d \bmod N$ (for $j = 1, \ldots, w$). These operations can be done as soon as server set is available. In the absolute worst case, server can perform these operations in parallel with receiving client's first message.

2. Results of server computation over its own set can be re-used in multiple protocol instances. Thus, unless server's set changes frequently, the overhead is negligible.

In light of the above, [DT10] suggests to divide the protocol into two phases: *off-line* and *on-line*. This way, computational complexity of the latter is dominated by $O(v)$ CRT exponentiations, while *off-line* phase overhead amounts to $O(w)$ CRT exponentiations. This makes DT10-v4 particularly appealing for scenarios where server input set is not *"very dynamic"*.

3 Implementing DT10-v4

This section presents our implementation of DT10-v4 PSI construction from [DT10]. We discuss some design choices that may affect overall performance, present our prototype implementation, and discuss

[1] Specifically, the proof constructs of an (adaptive) ideal world simulator SIM_c from a malicious real-world client C^*, and shows that the views of C^* in the real game with the real-world server and in the interaction with SIM_c are indistinguishable.

[2] See items 14.71 and 14.75 in [MVOV97] for more details on CRT-based exponentiation.

additional techniques to optimize performance.

3.1 Important Design Choices

We now identify and discuss some factors that significantly affect overall performance of DT10-v4 implementation. We begin with straightforward issues and then turn to some less trivial strategies. (*Note: for the sake of generality, we assume below that server does NOT perform precomputation.*)

1. ***Small RSA public exponent:*** Recall from Section 2.2, that the only modular exponentiations performed by client are those in step (2), specifically, raising random values r_i-s to the e-th power $(\mathrm{mod}\ N)$. Therefore, the choice of RSA public exponent e directly influences client run-time. Common choices of e are: $3, 17$, and 2^{16}+1 = 65537. The cryptography research community has often raised concerns related to possible attacks when using $e = 3$ for RSA encryption [Bon98, FKJM+06]. However, although further careful consideration is needed, such concerns do not seem to apply in this setting, since r_i-s are generated anew, at random.

2. ***Chinese Remainder Theorem:*** On server side, the most computation-intensive operations are exponentiations $(\cdot)^d \bmod N$ – in steps (1) and (4). As discussed in Section 2.2, these can be optimized using (CRT). Specifically, it is well known that using CRT can make exponentiations 4 times faster.

3. ***Pipelining:*** While we describe DT10-v4 as a sequence of steps, pipelining can be used to maximize overall efficiency by minimizing wait times. A good start is to implement computation and communication in separate *threads*, such that independent operations can be performed in parallel. (Note that this does not presume that underlying hardware has multiple cores). Specifically:

   a) Server can compute $ts_j = H'((hs_j)^d \bmod N)$, $j = 1, \ldots, w$ (i.e., steps (1)-(2)), as soon as (s_j's) are available, i.e., even before starting interaction with client, or, in the worst case, as soon as client starts transmitting. This is as simple as implementing server's steps (1)-(2) in a dedicated thread.

   b) Server does not need to wait for $\mu_{i+1}, \ldots, \mu_v$ to arrive in order to compute $\mu'_i = (\mu_i)^d \bmod N$. To minimize waiting, we simply need to implement exponentiations in a separate thread drawing input from a shared buffer, where the thread listening on the channel pushes received values.

   c) Similarly, client can compute ${r_i}^{-1}$ (needed to compute $\mu_i / r_i \bmod N$) in step (5) in parallel with steps (2)-(4).

   d) Finally, client does not need to wait for $\mu'_{i+1}, \ldots, \mu_v$ to arrive to compute $tc_i = H'(\mu'_i / r_i \bmod N)$, i.e., steps (5)-(6).

4. ***Threading in Multi-Core Settings:*** Structuring the code in multiple threads allows us to further improve overall performance. For example, on server side, we can create two threads for step (1) and step (4), respectively. Thus, if multiple cores are available (or the computing architecture using aggressive pipelining), these operations are performed in parallel, thus, lowering overall run-time. Once again, we note that parallel thread execution is transparent to application developers and normally incurs no extra costs.

5. ***Fast Cryptographic Library:*** The choice of the cryptographic library is a crucial factor affecting overall performance. Efficiency of modular exponentiations varies widely across cryptographic libraris. For example, Table 2 shows modular exponentiations measured on a 64-bit desktop with an Intel Xeon CPU E31225 at 3.10GHz (running Ubuntu 11.10), using increasingly large exponents and moduli.

	1024-bit	2048-bit	3072-bit
C/GMP	0.60ms	4.44ms	14.08ms
C/OpenSSL	0.81ms	6.12ms	20.89ms
Java (v1.6.0_23)	3.33ms	24.47ms	76.91ms
Ratio Java/GMP	5.55	5.51	5.46

Table 2: Benchmarking of modular exponentiations with increasingly large moduli.

3.2 Prototype Implementation

We now report on the design of a prototype implementation for the DT10-v4 PSI protocol. It is implemented in C, using the GMP library for large integer arithmetic and OpenSSL for key generation and hash function implementation. Motivated by the design choices above, we use multi-threading and the GMP library for operations on large integers, e.g., modular exponentiations, multiplications, and inverses.

Remark: The reported pseudo-code does not describe by any means a working implementation. Our goal is to outline an intuition of our prototype design following the discussion in Section 3.1. Nonetheless, it mirrors the software used in our experiments, presented in Section 4, which can be obtained upon request from the authors.

Pseudo-code: Figure 12 and Figure 13 overview DT10-v4 PSI server software. The client side is only sketched, in Figure 14, and detailed subroutines are not outlined to ease presentation. Figure 15 describes some of the auxiliary functions used by our prototype. (Figures appear at the end of the report).

3.3 Additional Performance-optimizing techniques

Besides design choices discussed in Section 3.1 above – all of which can be easily adopted – there are some less obvious aspects that can help us further optimize implementation of DT10-v4. Although we discuss them below, we defer their implementation to the next version of the prototype, since these optimizations appeal to specific settings. Whereas, this paper focuses on the general PSI scenario.

1. ***Bottleneck identification:*** In settings where the PSI protocol is executed over the Internet and communication takes place over slow links, communication overhead is likely to become the bottleneck. For instance, consider a scenario, where server runs on an Intel Xeon CPU at 3.10GHz. Using GMP, it takes, on average, $0.15ms$ to perform $(\cdot)^d \bmod N$ exponentiations, with 1024-bit moduli, using CRT. Therefore, one can estimate the link speed at which the bottleneck becomes transmission of the $\{\mu_i\}_{i=1}^{v}$ and $\{ts_j\}_{j=1}^{w}$ values, respectively. Specifically, if network speed is lower than: $\frac{|\mu'_i|}{\text{time}} = \frac{1024 \text{ bits}}{0.15\text{ms}} = 7.31\text{Mbps}$, it takes longer to transmit μ'_i than to compute it. Whereas, if network speed is lower than: $\frac{|ts_j|}{\text{time}} = \frac{160 \text{ bits}}{0.15\text{ms}} = 1.14\text{Mbps}$ then it takes longer to send ts_j than to compute it. These estimates could be useful for further protocol optimizations; see below.

2. ***Exploiting parallelism:*** Many modern desktops and laptops have multiple cores. Thus, if the bottleneck is computation of $(\cdot)^d$ exponentiations, the server-side thread in charge of receiving $\{\mu_i\}_{i=1}^{v}$ will push them into the buffer faster than the thread computing $\{(\mu_i)^d\}_{i=1}^{v}$ can pull them. With multiple cores, exponentiations of multiple values in the buffer could be done in parallel. Similarly, computation of $\{(hs_j)^d\}_{i=j}^{w}$ does not depend on any other information; thus, it could be parallelized too.

3. ***Minimizing transmission:*** If the bottleneck is transmission time, then we can optimize software by, for example, using UDP instead of TCP, or choosing socket options geared for transmission of many tiny packets.

4 Performance Evaluation

We now present a detailed performance evaluation of our DT10-v4 implementation.

Experimental Setup. Experiments are performed on the following testbed: PSI server ran on a Linux computer, equipped with an Intel Xeon E31225 CPU (running at 3.10GHz). PSI client ran on a Mid-2011 13-inch Apple Macbook Air, with an Intel Core i5 (running at 1.7GHz). Server and client are connected through a 100Mbps Ethernet LAN. The code was written in C using the GMP library for modular arithmetic operations and OpenSSL for other cryptographic operations (such as, random numbers and key generation, hash function invocations). Finally, note that we used 1024-bit, 2048-bit, or 3072-bit RSA moduli and SHA-1 to instantiate the $H'(\cdot)$ function.

4.1 Protocol Total Running Time

In Figures 2 and 3, we report total run-times for DT10-v4 protocol running on, respectively, small (100 to 1000) and medium (1000 to 10000) sets, using 1024-bit moduli.[3] Next, Figures 4, 5, 6, and 7, respectively, report total run-time for small and medium sets, using 2048-bit and 3072-bit moduli, respectively.

Time is measured as the difference between system time read when the protocol starts and time read when the protocol ends. Specifically, we consider the protocol as *started* whenever client initiates protocol execution (i.e., it opens a connection on server's listening socket), whereas, the protocol ends whenever client outputs the intersection (if any). In other words, *we do not perform precomputation* and, on a conservative stance, we do not allow the server to start computation of $\{(hs_j)^d \bmod N\}_{j=1}^{w}$ (its step (1)) until client establishes a connection on the listening socket.

The only cryptographic operation performed ahead of time (thus, not included in run-time) is RSA key generation, since server executes it only once for all possible clients and all executions. Finally, protocol execution time does not count time spent by server waiting for an incoming connection, since the listening socket is created only once, for all possible clients and all protocol executions.

4.2 Micro-benchmarking

We now analyze performance of specific operations performed by client and server during DT10-v4 protocol execution. We start with ***Client***. In Figure 8 (resp., Figure 10), we measure the time spent by the process executing DT10-v4 client, using 1024-bit (resp., 2048-bit) moduli, for the following operations:

1. Label 'Receive' corresponds to the time spent to wait/receive the $\{\mu'_i\}_{i=1}^{v}$ and $\{ts_j\}_{j=1}^{w}$ values from server.
2. Label 'Cli-1' corresponds to the time needed to compute $\{\mu_i = hc_i \cdot {r_i}^e \bmod N\}_{i=1}^{v}$.
3. Label 'Inverse' corresponds to the time to compute $\{r_i^{-1} \bmod N\}_{i=1}^{v}$.
4. Label 'Cli-2' corresponds to the time needed to compute $\{tc_i = H'(\mu'_i / r_i \bmod N)\}_{i=1}^{v}$.

Next, we look at ***Server***. In Figure 9, (resp., Figure 11)), we measure the time spent by the process executing DT10-v4 server, using 1024-bit (resp., 2048-bit) moduli, for the following operations:

1. Label 'Receive' corresponds to the time spent to wait/receive the $\{\mu_i\}_{i=1}^{v}$ values from client.
2. Label 'BlindSig' corresponds to the time to compute $\{\mu'_i = (\mu_i)^d \bmod N\}_{i=1}^{v}$.

[3] We also ran experiments with even large sets (in the order of hundreds of thousands). We do not include them here as they simply grow linearly for increasing set sizes, thus, one can obtain an estimation of them, for essentially any input size, by looking at Figures 2–7.

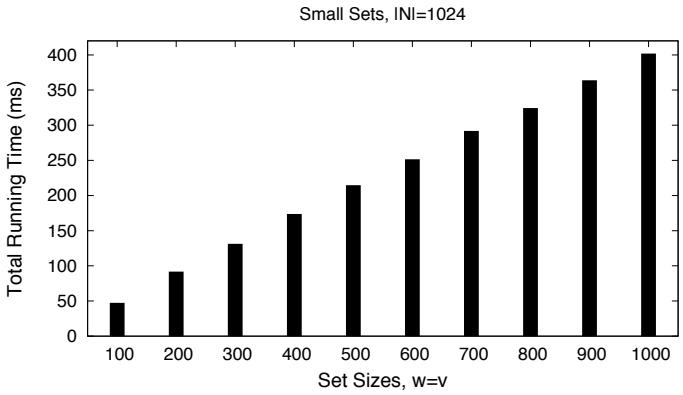

Figure 2: DT10-v4 total run-time for small sets (100 to 1000 items), using 1024-bit moduli.

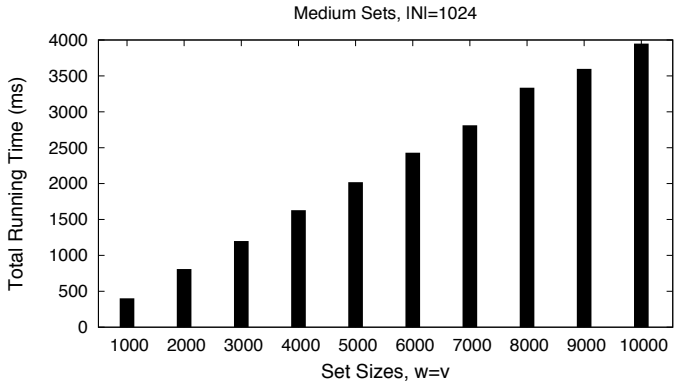

Figure 3: DT10-v4 total run-time for medium sets (1000 to 10000 items), using 1024-bit moduli.

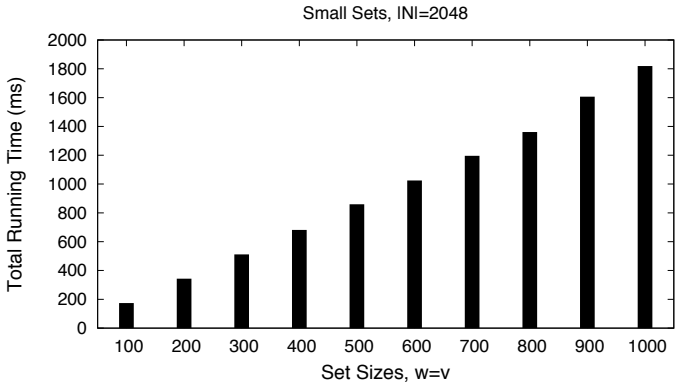

Figure 4: DT10-v4 total run-time for small sets (100 to 1000 items), using 2048-bit moduli.

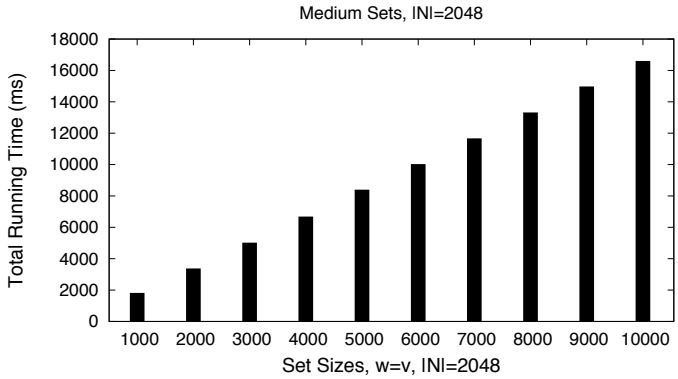

Figure 5: DT10-v4 total run-time for medium sets (1000 to 10000 items), using 2048-bit moduli.

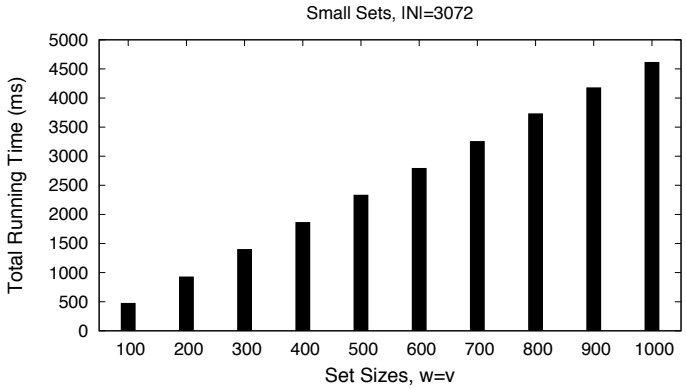

Figure 6: DT10-v4 total run-time for small sets (100 to 1000 items), using 3072-bit moduli.

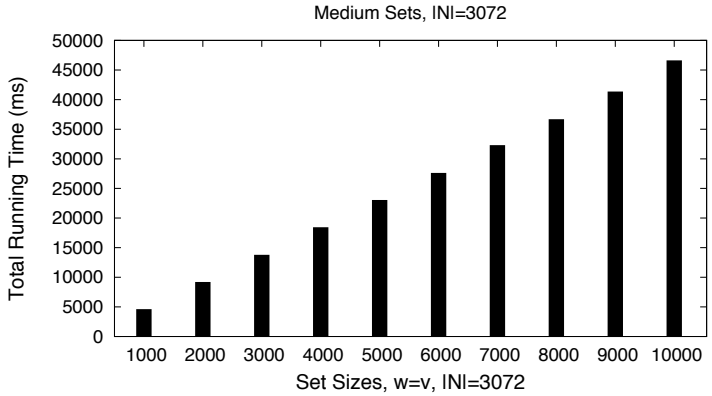

Figure 7: DT10-v4 total run-time for medium sets (1000 to 10000 items), using 3072-bit moduli.

3. Label 'Sig' corresponds to the time to compute $\{ts_j = H'((hs_j)^d \bmod N)\}_{j=1}^{w}$.

It is interesting to observe that, using 1024-bit moduli, client actually spends less time to receive all the values from server than vice versa, despite the former actually needs to receive more. This is a good opportunity to see multi-threading *in action*: client's thread responsible to send the $\{\mu_i\}$ values has to wait for them to be available, thus, causing some waiting time to server's thread that receives them. In other words, by looking at the micro-benchmarking one can identify different "bottlenecks" in the different settings.

5 Comparison to [HEK12]

In this section, we focus on the performance evaluation of the DT10-v4 PSI protocol presented in [HEK12].

The work in [HEK12] presents a few novel Private Set Intersection constructions based on garbled circuits [Yao82]: the main intuition is that, by leveraging the *Oblivious Transfer* (OT) extension [IKNP03], the complexity of such protocols is essentially tied to a number of OTs (thus, public-key operations) equal to the security parameter k. In fact, OT extension achieves an unlimited number of OTs at the cost of (essentially) k OTs. Therefore, for very large security parameters, the number of public-key operations with this technique may grow more gracefully than with custom protocols. Finally, [HEK12] compares the efficiency of newly proposed constructions to an implementation of "custom" PSI protocols from [DT10].

Note that we do not examine the proposals and the experimental methodology of [HEK12]. Rather, we observe that the implementation of DT10-v4 presented in this paper achieves a remarkable speed up

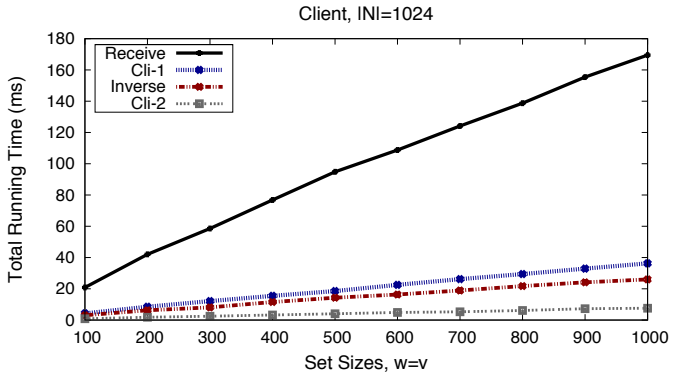

Figure 8: Micro-benchmarking client's operations in DT10-v4 (small sets), using 1024-bit moduli.

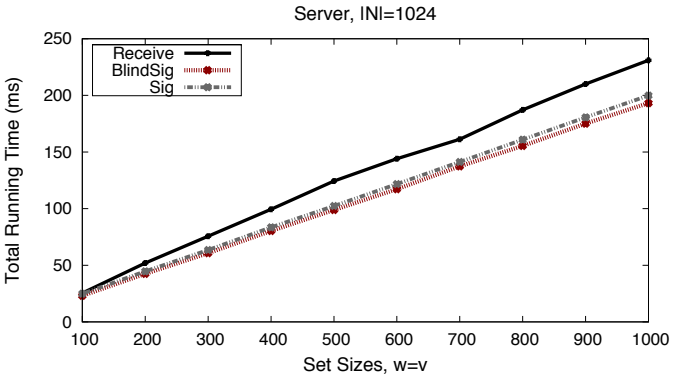

Figure 9: Micro-benchmarking server's operations in DT10-v4 (small sets), using 1024-bit moduli.

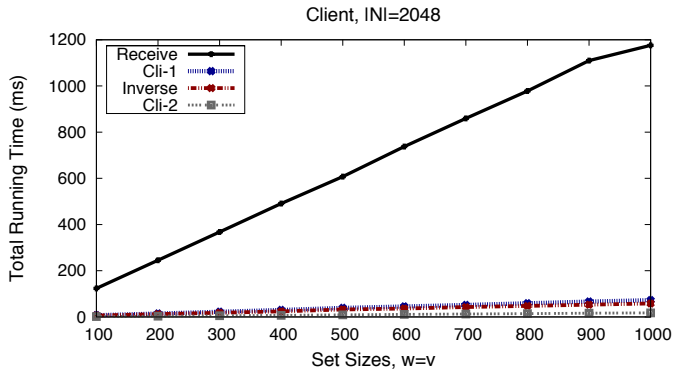

Figure 10: Micro-benchmarking client's operations in DT10-v4 (small sets), using 2048-bit moduli.

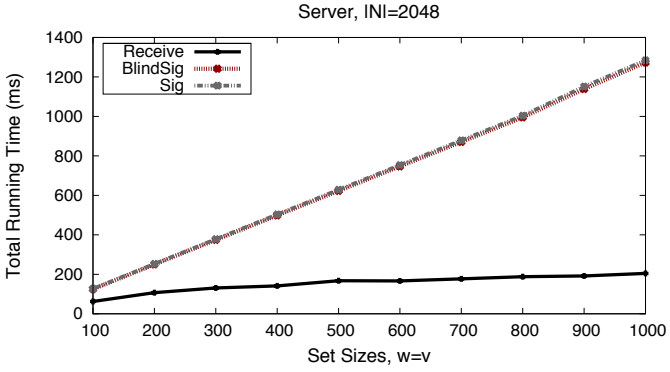

Figure 11: Micro-benchmarking server's operations in DT10-v4 (small sets), using 2048-bit moduli.

compared to performance results presented in [HEK12] for same protocols. Finally, we highlight some open questions regarding comparison between techniques in [HEK12] and those in [DT10].

5.1 Performance Comparison

We start by noticing that the run-time of the PSI protocol in [DT10] is reported to be around 10 seconds in a setting where $|\mathcal{S}| = |\mathcal{C}| = 1024$, the security parameter is 80-bit (thus, RSA moduli are 1024-bit), no precomputation is allowed at server, and communication between server and client is over a 100Mbps LAN. It is not clear whether this measure is the sum of server and client execution time or represents the time for the protocol to complete. On a conservative stance, we assume the former. On a comparable hardware,[4] and using the parameters discussed above, our measure for DT10-v4 protocol never exceeded 1 seconds (and DT10-v4 is actually not reported as the fastest protocol – see Section 5.2). Similarly, evaluation in [HEK12] reports 62 seconds (resp., 126 seconds) using 2048-bit (resp., 3072-bit) moduli; whereas, our implementation of DT10-v4 never exceeds 2 seconds (resp., 5 seconds).

In Table 3 below, we summarize running times for DT10-v4 as per our implementation, and compare to those for garbled-circuit based techniques presented in [HEK12] and reported in Fig. 11 of [HEK12]. We argue that our implementation of DT10-v4 markedly outperforms PSI protocols based on garbled circuits, in all the three security-parameter settings that we consider (and that are realistic today), in stark contrast to what has been claimed in [HEK12].

	80-bit	112-bit	128-bit
DT10-v4 as per our implementation	$< 1s$	$< 2s$	$< 5s$
Best Custom-protocol PSI as per [HEK12]'s experiments	10.9s	62.4s	126s
Garbled-circuit based PSI in [HEK12]	51.5s	57.1s	61.5s

Table 3: Summary of PSI running times (with $|\mathcal{S}| = |\mathcal{C}| = 1024$).

5.2 The choice of protocols from [DT10]

Authors of [HEK12] argue that the protocol in Figure 3 of [DT10], based on the One-More-DH assumption, is more efficient than that in Figure 4 (based on the One-More-RSA assumption and denoted as DT10-v4) in scenarios where server-side precomputation is not possible. Our analysis below shows that this is wrong.

In the following, aiming at 80-bit security, we use: a 1024-bit RSA modulus N, an RSA public exponent $e = 3$, CRT-optimized exponentiations, a 1024-bit prime p, a 160-bit prime q, and SHA-1 hash function. Also recall that $w = |\mathcal{S}|$ and $v = |\mathcal{C}|$. We also use m to denote a modular multiplication of 1024-bit integers. Consequently, we say that exponentiations modulo 1024 bits require, on average, $O(1.5 \cdot |exp|) \cdot \mathsf{m}$, where $|exp|$ denotes exponent size. Modular exponentiations with 512-bit moduli count for approximately $O(1.5 \cdot |exp|) \cdot \mathsf{m}/4$. As we discussed earlier in the paper, the computational complexity of protocol in Figure 4 in [DT10] (DT10-v4) is clearly determined by $2w + 2v$ exponentiations with 512-bit exponents and moduli, thus, $(2w+2v)(1.5 \cdot 512)\mathsf{m}/4$, i.e., $(384w+384v) \cdot \mathsf{m}$. Whereas, the computational complexity of protocol in Figure 3 of [DT10] comes down to $w + 3v$ exponentiations with 160-bit exponents and 1024-bit moduli, thus, $(w + 3v) \cdot (1.5 \cdot 160) \cdot \mathsf{m}$, i.e., $(240w + 720v) \cdot \mathsf{m}$.

If one allows precomputation, then protocol in Figure 4 (DT10-v4) is straightforwardly more efficient than the Figure 3 counterpart, since online complexity goes down to $(384v) \cdot \mathsf{m}$. But if one does not allow

[4] In [HEK12] both server and client run on 3GHz CPU, whereas, in our experiments, server runs on a 3.1GHz CPU and client on a 1.7GHz CPU.

precomputation (as in [HEK12]), then it would seem that Figure 3 protocol would outperform DT10-v4 for settings where approximately $\frac{v}{w} < \frac{4}{10}$ — a setting that is anyway never tested in [HEK12], which always assumes $w = v$. Nonetheless, when precomputation is not possible, then the analysis of Figure 3's complexity should actually account for $w + v$ additional exponentiations needed to evaluate the $H(\cdot)$ function, which is of the *hash-into-the-group* kind, i.e., $H(x) = x^{(\frac{p-1}{q})} \bmod p$, thus, protocol in Figure 3 appears to be always slower than Figure 4 (i.e., DT10-v4.) Therefore, the protocol in [DT10]'s Figure 4 is always more computational efficient than the one in Figure 3.

5.3 Evaluation Criteria

Once again, note that it is out of the scope of this paper to provide a definite explanation as to why our implementation of DT10-v4 achieves run-times several times lower than those reported by [HEK12] (see section 7 thereof). Similarly, we do not analyze the validity of the conclusions drawn by the authors of [HEK12] regarding whether or not DT10-v4 PSI protocol is more efficient than garbled circuits-based constructions in all settings. However, we make some observations regarding implementation of DT10-v4 by Huang et al. [HEK12] and also argue that a comprehensive comparison should take into account several settings (we sketch those below and leave the task of addressing them as an interesting open problem).

1. As discussed earlier, several design factors (e.g., pipelining, CRT, etc.) significantly impact overall performance of custom PSI protocols (see Section 3.1) and it is unclear whether they were taken into account in [HEK12].

2. [HEK12] implements techniques from [DT10] and in [HEK12] in *Java*. Java usually offers slower performance than other programming languages (such as C/C++). Nonetheless, this choice might seem irrelevant, since both techniques are implemented in Java. However, we believe it remains to be seen if the use of Java penalizes techniques from [DT10] that perform a higher number of public-key operations. For instance, as mentioned earlier, a CRT-based RSA exponentiation takes 5.55 times longer in Java than in C/GMP. Does this slowdown occur, *in the same measure* for *all* Java operations (e.g., symmetric-key)? If not, then the choice of Java might not be fair, as constructions in [HEK12] heavily rely on symmetric-key operations.[5] Also, it would also be interesting to measure memory overhead for increasing set sizes incurred by all techniques. We believe that performance and scalability could be tremendously affected by, for example, inability to keep an entire circuit in memory.

3. [HEK12] employs techniques that are fundamentally and markedly different from those used by custom protocols. Thus, a different choice of parameters can significantly favor one while penalizing the other. We mention just a few:

   a) Techniques in [HEK12] are tested in settings where $|\mathcal{S}| = |\mathcal{C}|$. As a result, we believe that a more thorough comparison would include scenarios where $|\mathcal{S}| \neq |\mathcal{C}|$. Also, comparisons in [HEK12] are given only for $|\mathcal{S}| = |\mathcal{C}| = 1024$. It remains unclear how performance of protocols in [HEK12] would scale for higher set sizes, since at least some of them involve *non-linear* complexities, as opposed to their counterparts in [DT10].

   b) Some protocols in [HEK12] incur higher communication complexity than protocols in [DT10]. Therefore, we argue that a more thorough comparison must include (realistic) settings where the subject protocol is executed on the Internet, and not only over fast 100Mbps LANs. (Complexity

[5] To encrypt 1 million 64-byte strings with AES-CBC, using C/OpenSSL, it takes, on average 0.60 and 0.83 seconds, with, respectively, 128-bit and 256-bit keys. Whereas, in Java, it takes 1.22 and 1.58 seconds. Therefore, the slowdown factor here is only 2.03 for 128-bit keys and 1.90 for 256-bit keys (versus about 5.5 for modular exponentiations).

is not analyzed asymptotically but authors of [HEK12] report, on page 13, that the SCS-WN protocol consumes more bandwidth: 147–470MB, depending on the security level, versus 0.4–2.0MB.)

c) Experiments in [HEK12] measure run-times as a total execution time. However, we believe that more details – ideally, a benchmark of sub-operations – should also be provided to better understand if the testing setting and implementation choices penalize one technique while favoring another.

Finally, while research on custom PSI protocols reached the point where malicious security can be achieved efficiently – at the same asymptotic complexity as semi-honest security [HN10, JL10, DKT10] – efficiency of garbled-circuit-based techniques secure in the malicious model remains unclear.

6 Conclusion

This paper presented an optimized implementation and performance evaluation of the currently fastes PSI protocol from [DT10]. We analyzed implementation choices that impact overall performance and presented an experimental analysis, including micro-benchmarking, with different set sizes. We showed that resulting run-times appreciably outperform those reported in [HEK12]. Achieved speed up is significantly higher than what one would obtain by simply porting [HEK12] implementation of DT10-v4 from Java to C. Finally, we identified some open questions with respect to comparisons of custom PSI protocols with generic garbled-circuit based constructions.

Acknowledgments. We gratefully acknowledge Yanbin Lu, Paolo Gasti, Simon Barber, and Xavier Boyen for their help and suggestions. We would also like to thank the authors of [HEK12] for their valuable feedback.

References

[ADT11] G. Ateniese, E. De Cristofaro, and G. Tsudik. (If) Size Matters: Size-Hiding Private Set Intersection. In *PKC*, 2011.

[AL07] Y. Aumann and Y. Lindell. Security Against Covert Adversaries: Efficient Protocols for Realistic Adversaries. In *TCC*, 2007.

[BBD+11] P. Baldi, R. Baronio, E. De Cristofaro, P. Gasti, and G. Tsudik. Countering gattaca: efficient and secure testing of fully-sequenced human genomes. In *CCS*, 2011. Available from `http://arxiv.org/abs/1110.2478`.

[BCC+09] M. Belenkiy, J. Camenisch, M. Chase, M. Kohlweiss, A. Lysyanskaya, and H. Shacham. Randomizable Proofs and Delegatable Anonymous Credentials. In *CRYPTO*, 2009.

[BLHB11] E. Bursztein, J. Lagarenne, M. Hamburg, and D. Boneh. OpenConflict: Preventing Real Time Map Hacks in Online Games. In *IEEE Security and Privacy*, 2011.

[BNPS03] M. Bellare, C. Namprempre, D. Pointcheval, and M. Semanko. The one-more-RSA-inversion problems and the security of Chaum's blind signature scheme. *Journal of Cryptology*, 16(3), 2003.

[Bon98] D. Boneh. Twenty years of attacks on the RSA cryptosystem. *Notices of the AMS*, 46(2), 1998.

[CM99] J. Camenisch and M. Michels. Proving in zero-knowledge that a number is the product of two safe primes. In *EUROCRYPT*, 1999.

[CS03] J. Camenisch and V. Shoup. Practical verifiable encryption and decryption of discrete logarithms. In *CRYPTO*, 2003.

[DJL+10] E. De Cristofaro, S. Jarecki, X. Liu, Y. Lu, and G. Tsudik. Automatic Privacy Protection Program – UC Irvine Team Web Site. http://sprout.ics.uci.edu/projects/iarpa-app, 2010.

[DKT10] E. De Cristofaro, J. Kim, and G. Tsudik. Linear-Complexity Private Set Intersection Protocols Secure in Malicious Model. In *ASIACRYPT*, pages 213–231, 2010.

[DSMRY09] D. Dachman-Soled, T. Malkin, M. Raykova, and M. Yung. Efficient Robust Private Set Intersection. In *ACNS*, 2009.

[DT10] E. De Cristofaro and G. Tsudik. Practical Private Set Intersection Protocols with Linear Complexity. In *Financial Cryptography and Data Security*, 2010. available from http://eprint.iacr.org/2009/491.

[DT12] E. De Cristofaro and G. Tsudik. Experimenting with fast private set intersection. In *TRUST*, 2012.

[ElG85] T. ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE transactions on Information Theory*, 31(4), 1985.

[FIPR05] M.J. Freedman, Y. Ishai, B. Pinkas, and O. Reingold. Keyword search and oblivious pseudorandom functions. In *TCC*, 2005.

[FKJM+06] P.A. Fouque, S. Kunz-Jacques, G. Martinet, F. Muller, and F. Valette. Power attack on small rsa public exponent. *Cryptographic Hardware and Embedded Systems-CHES 2006*, 2006.

[FNP04] M.J. Freedman, K. Nissim, and B. Pinkas. Efficient private matching and set intersection. In *EUROCRYPT*, 2004.

[GGM86] O. Goldreich, S. Goldwasser, and S. Micali. How to construct random functions. *Journal of the ACM*, 33(4), 1986.

[HEK12] Y. Huang, D. Evans, and J. Katz. Private Set Intersection: Are Garbled Circuits Better than Custom Protocols? In *NDSS*, 2012.

[HL08] C. Hazay and Y. Lindell. Efficient protocols for set intersection and pattern matching with security against malicious and covert adversaries. In *TCC*, 2008.

[HMRT11] C. Hazay, G.L. Mikkelsen, T. Rabin, and T. Toft. Efficient rsa key generation and threshold paillier in the two-party setting. Cryptology ePrint Archive, http://eprint.iacr.org/2011/494, 2011.

[HN10] C. Hazay and K. Nissim. Efficient Set Operations in the Presence of Malicious Adversaries. In *PKC*, 2010.

[IKNP03] Y. Ishai, J. Kilian, K. Nissim, and E. Petrank. Extending oblivious transfers efficiently. *CRYPTO*, 2003.

[JL09] S. Jarecki and X. Liu. Efficient Oblivious Pseudorandom Function with Applications to Adaptive OT and Secure Computation of Set Intersection. In *TCC*, 2009.

[JL10] S. Jarecki and X. Liu. Fast secure computation of set intersection. In *SCN*, 2010.

[KL08] J. Katz and Y. Lindell. *Introduction to modern cryptography*. Chapman & Hall/CRC, 2008.

[KS05] L. Kissner and D. Song. Privacy-preserving set operations. In *CRYPTO*, 2005.

[MVOV97] A. Menezes, P. Van Oorschot, and S. Vanstone. *Handbook of Applied Cryptography*. CRC, 1997.

[NMH+10] S. Nagaraja, P. Mittal, C.Y. Hong, M. Caesar, and N. Borisov. BotGrep: Finding Bots with Structured Graph Analysis. In *Usenix Security*, 2010.

[NP06] M. Naor and B. Pinkas. Oblivious polynomial evaluation. *SIAM Journal on Computing*, 1–35(5), 2006.

[NTL+11] A. Narayanan, N. Thiagarajan, M. Lakhani, M. Hamburg, and D. Boneh. Location Privacy via Private Proximity Testing. In *NDSS*, 2011.

[Pai99] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *EUROCRYPT*, 1999.

[RS60] S. Reed and G. Solomon. Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics*, 8(2), 1960.

[Sha79] A. Shamir. How to Share a Secret. *Communications of ACM*, 22(11), 1979.

[Yao82] A.C. Yao. Protocols for secure computations. In *FOCS*, 1982.

A Survey of PSI Techniques

In this appendix, we survey research work on Private Set Intersection (PSI). This appendix is organized in chronological order: we first overview work prior to [DT10], then, after discussing the work in [DT10], we present recent results.

A.1 Work prior to [DT10]

Once again, recall that PSI is a protocol involving a server and a client, on inputs $\mathcal{S} = \{s_1, \ldots, s_w\}$ and $\mathcal{C} = \{c_1, \ldots, c_v\}$, respectively, that results in client obtaining $\mathcal{S} \cap \mathcal{C}$. As a result of running PSI, set sizes are reciprocally disclosed to both server and client. In the variant called *PSI with Data Transfer* (PSI-DT), each item in server set has an associated data record, i.e., server's input is $\mathcal{S} = \{(s_1, data_1), \cdots, (s_w, data_w)\}$, and client's output is defined as $\{(s_j, data_j) \in \mathcal{S} \mid \exists c_i \in \mathcal{C}\ s.t.\ c_i = s_j\}$.

We distinguish between two classes of PSI protocols: one based on Oblivious Polynomial Evaluations (OPE) [NP06], and the other based on Oblivious Pseudo-Random Functions (OPRF-s) [FIPR05].

Freedman, Nissim, and Pinkas [FNP04] introduce the concept of Private Set Intersection and and propose a protocol based on OPE. They represent a set as a polynomial, and elements of the set as its roots. A client encodes elements in its private set $\mathcal{C}$ as the roots of a v-degree polynomial over a ring R, i.e., $f = \prod_{i=1}^{v}(x - c_i) = \sum_{i=0}^{k} \alpha_i x^i$. Then, assuming pk_C is client's public key for any additively homomorphic cryptosystem (such as Paillier's [Pai99]), client encrypts the coefficients with pk_C, and sends them to server. The latter homomorphically evaluates f at each $s_j \in \mathcal{S}$. Note that $f(s_j) = 0$ if and only if $s_j \in \mathcal{C} \cap \mathcal{S}$. For each $s_j \in \mathcal{S}$, returns $u_j = E(r_j f(s_j) + s_j)$ to client (where r_j is chosen at random and $E(\cdot)$ denotes additively homomorphic encryption under pk_C). If $s_j \in \mathcal{C} \cap \mathcal{S}$ then client learns s_j upon decrypting. If $s_j \notin \mathcal{C} \cap \mathcal{S}$ then u_j decrypts to a random value. To enable data transfer, server can return $E(r_j f(s_j) + (s_j || data_j))$, for each s_j in its private set $\mathcal{S}$. The protocol in [FNP04] incurs the following complexities: The number of server operations depends on the evaluation of client's encrypted polynomial with v coefficients on w points (in $\mathcal{S}$). Using Paillier cryptosystem [Pai99] and a 1024-bit modulus, this costs $O(vw)$ of 1024-bit mod 2048-bit exponentiations.[6] On the other hand, client computes $O(v+w)$ of 1024-bit mod 2048-bit exponentiations. However, server computation can be reduced to $O(w \log\log v)$ using: (1) Horner's rule for polynomial evaluations, and (2) a hashing-to-bins method (see [FNP04] for more details). If one does not need *data transfer*, it is more efficient to use the Exponential ElGamal cryptosystem [ElG85] (i.e., an ElGamal variant that provides additively homomorphism).[7] Such a cryptosystem does not provide efficient decryption, however, it allows client to test whether a ciphertext is an encryption of "0", thus, to learn that the corresponding element belongs to the set intersection. As a result, efficiency is improved, since in ElGamal the computation may make use of: (1) very short random exponents (e.g., 160-bit) and (2) shorter moduli in exponentiations (1024-bit). The PSI protocol in [FNP04] is secure against honest-but-curious adversaries in the standard model, and can be extended to malicious in the Random Oracle Model (ROM), at an increased cost.

[6] Encryption and decryption in the Paillier cryptosystem [Pai99] involve exponentiations mod n^2: if $|n| = 1024$ bits, then $|n^2| = 2048$ bits (where n is the public modulus).

[7] In the Exponential ElGamal variant, encryption of message m is computed as $E_{g,y}(m) = (g^r, y^r \cdot g^m)$ instead of $(g^r, m \cdot y^r)$, for random r and public key y.

Hazay and Nissim [HN10] present an improved construction of [FNP04], in the presence of malicious adversaries without ROM, using zero-knowledge proofs to let client demonstrate that encrypted polynomials are correctly produced. Perfectly hiding commitments, along with an Oblivious Pseudo-Random Function evaluation protocol, are used to prevent server from deviating from the protocol. The protocol in [HN10] incurs $O(v + w(\log\log v + m))$ computational and $O(v + w \cdot m)$ communication complexity, where m is the number of bits needed to represent a set element.

Kissner and Song [KS05] also propose OPE-based protocols involving (potentially) more than two players. They present one technique secure in the standard model against semi-honest and one – against malicious adversaries. The former incurs quadratic – $O(vw)$ – computation (but linear communication) overhead. The latter uses expensive generic zero-knowledge proofs to prevent parties from deviating to the protocol. Also, it is not clear how to enable *data transfer*.

Dachman-Soled, et al. [DSMRY09] also present an OPE-based PSI construction, improving on [KS05]. Their protocol incorporates a secret sharing of polynomial inputs: specifically, as Shamir's secret sharing [Sha79] implies Reed-Solomon codes [RS60], generic (i.e., expensive) zero-knowledge proofs can be avoided. Complexity of resulting protocol amounts to $O(wk^2 \log^2(v))$ in communication and $O(wvk \log(v) + wk^2 \log^2(v))$ in computation, where k is a security parameter.

Other techniques rely on *Oblivious Pseudo-Random Functions* (OPRF-s), introduced in [FIPR05]. An OPRF is a two-party protocol that securely computes a pseudo-random function $f_k(\cdot)$ on key k contributed by the sender and input x contributed by the receiver, such that the former learns nothing from the interaction and the latter learns only the value $f_k(x)$. Most prominent OPRF-based protocols are presented below. The intuition behind OPRF-based PSI protocols is as follows: server and client interact in v parallel execution of the OPRF $f_k(\cdot)$, on input k and $c_i, \forall\ c_i \in \mathcal{C}$, respectively. As server transfers $T_{s:j} = f_k(s_j), \forall\ s_j \in \mathcal{S}$ and client obtains $T_{c:i} = f_k(c_i), \forall\ c_i \in \mathcal{C}$, client learns the set intersection by finding matching $(T_{s:j}, T_{c:i})$ pairs, while it learns nothing about values $s_l \in \mathcal{S} \setminus \mathcal{S} \cap \mathcal{C}$, since $f_k(s_l)$ is indistinguishable from random, if $f_k(\cdot)$ is a pseudo-random function.[8]

Hazay and Lindell [HL08] propose the first PSI construction based on OPRF-s. In it, server generates a secret random key k, then, for each $s_j \in \mathcal{S}$, computes $u_j = f_k(s_j)$, and sends client the set $\mathcal{U} = \{u_1, \cdots, u_w\}$. Next, client and server engage in an OPRF computation of $f_k(c_i)$ for each $c_i \in \mathcal{C}$. Finally, client learns that $c_i \in \mathcal{C} \cap \mathcal{S}$ if (and only if) $f_k(c_i) \in \mathcal{U}$. [HL08] introduces two constructions: one secure in the presence of malicious adversaries with one-sided simulatability, the other – in the presence of covert adversaries [AL07].

Jarecki and Liu [JL09] improve on [HL08] by constructing a protocol secure in the standard model against both malicious parties, based on the Decisional q-Diffie-Hellman Inversion assumption, in the Common Reference String (CRS) model, where a safe RSA modulus must be pre-generated by a trusted party. The OPRF in [JL09] is built using the Camenisch-Shoup additively homomorphic cryptosystem [CS03] (CS for short). However, this technique can be optimized, leading to the work by Belenkiy, et al. [BCC+09]. In fact, the OPRF construction could work in groups of 160-bit prime order, unrelated to the RSA modulus, instead of (more expensive) composite order groups [JL09]. Thus improved, the protocol in [JL09] incurs the following computational complexity: server needs to perform $O(w)$ PRF evaluations, specifically, $O(w)$ modular exponentiations of m-bit exponents mod n^2, where m the number of bits needed to represent set items and n^2 is typically 2048-bit long. The client needs to compute $O(v)$ CS encryptions, i.e., $O(v)$ m-bit exponentiations mod 2048 bits, plus $O(v)$ 1024-bit exponentiations mod 1024 bits. The server also computes $O(v)$ 1024-bit exponentiations mod 1024 bits and $O(v)$ CS decryptions – i.e., $O(v)$ 1024-bit exponentiations mod 2048 bits. Complexity in malicious model grows by a factor of 2. The input domain size of the pseudo-random function in [JL09] is limited to be polynomial in the security parameter, since the security proof requires the ability to exhaustively search over input domain.

[8] For more details on pseudo-random functions, we refer to [KL08, GGM86].

A.2 Protocols in [DT10]

The work in [DT10] presented two linear-complexity PSI protocols, both secure in the Random Oracle Model in the presence of semi-honest adversaries. Specifically, in [DT10], they present:

1. One protocol (Figure 3) secure under the *One-More-Gap-DH* assumption [BNPS03]. It imposes $O(w+v)$ *short* exponentiations on server, and $O(v)$ – on client. Note that the term "short" exponentiation refers to the fact that exponentiations can be of 160-bit exponents modulo 1024 bits (for 80-bit security).

2. Another protocol (Figure 4) secure under the *One-More-RSA* assumption [BNPS03], whose implementation we have presented and analyzed in this paper. Recall that, in this protocol, server computational overhead amounts to $O(w+v)$ RSA signatures using CRT optimization (i.e., 512 bits modulo 512 bits exponentiations for 80-bit security). Whereas, client complexity is dominated by $O(v)$ RSA encryptions, i.e., in practice, $O(v)$ modular multiplications if a short RSA public exponent is selected.

Both protocols incur the following communication overhead: client and server need to send and receive $O(v)$ group elements (i.e., 1024-bit); additionally, server sends client $O(w)$ hash outputs (e.g., 160-bit using SHA-1).

A.3 Recent results

Shortly after [DT10], Jarecki and Liu [JL10] also propose a PSI protocol with linear complexity and fast exponentiations. (Remark that some of the proofs in [DT10] are based on that of Jarecki and Liu.) This protocol is based on a concept related to OPRFs, i.e., *Unpredictable Functions* (UPFs). One specific UPF, $f_k(x) = H(x)^k$, is used as a basis for two-party computation (in ROM), with server contributing the key k and client – the argument x. The client picks a random exponent α and sends $y = H(x)^\alpha$ to server, that replies with $z = y^k$, such that client recovers $f_k(x) = z^{1/\alpha}$. By using a zero-knowledge discrete-log proofs of knowledge, the protocol in [JL10] can obtain *malicious security* and implement secure computation of (Adaptive) Set Intersection, under the *One-More-Gap-DH* assumption in ROM [BNPS03]. Therefore, the computational complexity of the UPF-based PSI in [JL10] also amounts to $O(w+v)$ exponentiations with short exponents at server side and $O(v)$ at client side (e.g., 160-bit mod 1024-bit). Communication complexity is also linear is input set size, i.e., $O(w+v)$.

De Cristofaro, et al. [DKT10] present another linear-complexity short-exponent PSI construction secure in ROM in the presence of *malicious* adversaries. However, compared to [JL10], its security relies on a weaker assumption – DDH vs One-More-Gap-DH. Then, Ateniese, et al. [ADT11] introduce the concept of *Size-Hiding* Private Set Intersection (SHI-PSI). Besides the standard privacy features guaranteed by the PSI primitive, SHI-PSI additionally provides *unconditional* (i.e., not padding-based) hiding of client's set size. The security of this novel protocol is under the RSA assumption in ROM, in the presence of semi-honest adversaries. Server's computational complexity amounts to only $O(w)$ exponentiations in the RSA setting, thus, it is independent of size of client's input. Whereas, client's overhead is in the order of $O(v \cdot \log v)$ exponentiations. Communication complexity is limited to $O(w)$, i.e., it is also independent of size of client's input.

Finally, Huang, et al. [HEK12] present novel PSI constructions based on garbled circuits [Yao82]. The main intuition is that, by leveraging the *Oblivious Transfer* (OT) extension [IKNP03], the complexity of such protocols is tied to a number of OTs (thus, public-key operations) equal to the security parameter k. In fact, OT extension achieves an unlimited number of OTs at the cost of (essentially) k OTs. Therefore, for increasing security parameters, the number of public-key operations with their technique grows more gracefully than with custom protocols.

```
/* Include header files */
#include ...

/* Global variables */
int sockfd, newsockfd, w, v;
RSA_params *rsa;
...

int main(int argc, char *argv[])
{
    /* Declare variables */
    pthread_t t1, t2, t3, t4, t5;
    int portno;
    socklen_t clilen;
    struct sockaddr_in serv_addr, cli_addr;
    ...

    /* Invoke socket(), bind(), listen() */
    if( (sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        printf("ERROR opening socket\n");
        exit(-1);
    }
    ..

    /* Load RSA keypair from storage */
    load_keys(rsa);

    /* Accept incoming connections */
    if( (newsockfd = accept(sockfd,(struct sockaddr *) &cli_addr,&clilen))< 0) {
        printf("ERROR on accept\n");
        exit(-1);
    }

    /* If supporting pre-computation the following
       instruction can be moved before the accept() */
    /* Compute ts_j values  */
    pthread_create(&t1, NULL, (void *)compute_tags, NULL);

    /* Read \mu_i values from socket */
    pthread_create(&t2, NULL, (void *)receive_from_client, NULL);

    /* Compute \mu'_i=\mu_i^d  */
    pthread_create(&t3, NULL, (void *)blind_sign, NULL);

    /* Wait for t2 to complete before start sending on the channel
       (conservative choice if channel not duplex) */
    pthread_join(t2, NULL);

    /* Send \mu'_i values to client  */
    pthread_create(&t4, NULL, (void *)send_bsig, NULL);

    /* Wait for t4 to complete */
    pthread_join(t4, NULL);

    /* Send \ts_j values to client  */
    pthread_create(&t5, NULL, (void *)send_tags, NULL); //compute mu_i-s

    /* That's all, folks! */
    pthread_join(t1, NULL);
    pthread_join(t3, NULL);
    pthread_join(t5, NULL);

    close(newsockfd);
    close(sockfd);
    sleep(1);
}
```

Figure 12: DT10-v4 PSI server software.

```
void receive_from_client(){
    int i;
    char buffer[N_size];

    /* Read \mu_i values from socket */
    for(i=0;i<v;i++) {
        if(!read_from_socket(newsockfd,N_size,buffer)) {
            printf("Error reading from socket\n");
            exit(-1);
    }
    /* Put \mu_i values into a shared buffer for blind_sign()
       thread to process them, e.g., write them to a pipe */
    write_to_pipe(pipe1[1], buffer, strlen(buffer));
    }
}

void blind_sign() {
    int i;
    mpz_t mu_i, mup_i; //mpz_t is GMP big-integer type
    char buffer[N_size];

    for(i=0;i<v;i++) {
        mpz_init(mup_i);
        mpz_init(mu_i);
        bzero(buffer,N_size);

        /* Get \mu_i read in receive_from_client() */
        read_from_pipe(pipe1[0], buffer, N_size);
        mpz_set_str(&mu_i, buffer, base); //transform into mpz_t

        /* Compute \mu'_i=\mu_i^d mod N, using CRT in GMP */
        sig_crt_gmp(mup_i,mu_i,rsa);

        /* Put \mu'_i values in a shared buffer for send_bsig() */
        ...
    }
}

void compute_tags() {
    int j;
    mpz_t ks_j;
    mpz_t hs_j;
    char buffer[max_size];

    for (j=0; j<w;j++) {
        /* Obtain s_j */
        ...

        /* Compute ts_j=H'(H(s_j)^d mod N) */
        FDH(s_j, strlen(s_j), &hs_j);
        sig_crt_gmp(ks_j,hsj,rsa);
        ts_j = SHA-1(ks_j);

        /* Put ts_j values in a shared buffer for send_tags() */
        ...
    }
}

void send_bsig(){
    ..

    /* Get \mu'_i from shared buffer and send to the client */
    for(i=0;i<v;i++) {
        write_to_socket(newsockfd, ...)
    }
    ...
}


void send_tags(){
    ...

    /* Get \ts_j from shared buffer and send to the client */
    for(j=0;i<w;j++) {
        write_to_socket(newsockfd, ...)
    }

    ...
}
```

Figure 13: DT10-v4 PSI server software (continued).

```
/* Include header files */
#include ...

/* Global variables */
int sockfd, w, v;
RSA_params *rsa;
...

int main(int argc, char *argv[])
{
    /* Declare variables */
    pthread_t t1, t2, t3, t4, t5, t6, final;
    int portno;
    socklen_t clilen;
    struct sockaddr_in serv_addr, cli_addr;
    ...


    /* Open Socket */
    if( (sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        printf("ERROR opening socket\n");
        exit(-1);
    }

    ...

    /* Connect to server socket */
    if (connect(sockfd,(struct sockaddr *) &serv_addr,sizeof(serv_addr)) < 0) {
        printf("ERROR connecting\n");
        exit(-1);
    }

    load_pub_key(rsa);

    /* Read client items */
    pthread_create(&t1, NULL, (void *)read_ci, NULL);

    /* Compute \mu_i=H(c_i)*r_i^e mod N */
    pthread_create(&t2, NULL, (void *)compute_mui, NULL);

    /* Send \mu_i values to the server */
    pthread_create(&t3, NULL, (void *)send_mui, NULL);

    /* Compute modular inverse of r_i values */
    pthread_create(&t4, NULL, (void *)invert_ri, NULL); //send mu_i-s

    /* Wait to finish using the socket */
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    pthread_join(t3, NULL);
    pthread_join(t4, NULL);

    /* Receive \mu'_i values from the server */
    pthread_create(&t5, NULL, (void *)receive_mupi, NULL);

    /* Proces \mu'_i values */
    pthread_create(&t6, NULL, (void *)process_mupi, NULL);

    /* Wait to finish receiving \mu'_i */
    pthread_join(t5, NULL);
    pthread_join(t6, NULL);

    /* Receive ts_j values and output intersection */
    pthread_create(&final, NULL, (void *)finalize, NULL);

    /* That's all folks! */
    pthread_join(final, NULL);
    close(sockfd);
}
```

Figure 14: DT10-v4 PSI client software.

```
void FDH(const unsigned char *input, int len, mpz_t *output){

    int i, nbits=rsa_key_length;
    int repss = (nbits + hash_size - 1) / hash_size;
    unsigned char out[rsa_key_length/8*2], ibuf[len*2];
    sprintf(out,"");

    for (i = 0; i < reps; i++) {
        sprintf(ibuf,"%s||%d",input,i);
        strcat(out, SHA1((const unsigned char*)ibuf, strlen(ibuf), NULL));
    }
    mpz_import(*output, strlen(out), 1, sizeof(char), 0, 0,
               (const unsigned char *)out);
}


int sign_crt_gmp(mpz_t sig, mpz_t msg, RSA_params *rsa) {
    mpz_t m1, m2, tmp, hq;
    mpz_init(m1);mpz_init(m2);mpz_init(tmp);mpz_init(hq);
    mpz_powm(m1, msg, rsa->dmp1, rsa->p); //dmp1=d mod p-1
    mpz_powm(m2, msg, rsa->dmq1, rsa->q); //dmq1=d mod q-1
    mpz_add(tmp, m1, rsa->p);
    mpz_sub(tmp, tmp, m2);
    mpz_mod(tmp, tmp, rsa->p);
    mpz_mul(hq, rsa->iqmp, tmp);
    mpz_mod(hq, hq, rsa->p);
    mpz_mul(hq, hq, rsa->q);
    mpz_add(sig, m2, hq);
}
```

Figure 15: Implementation of (some) auxiliary functions: Full-Domain Hash and CRT-based RSA signature.