PriSearch: Efficient Search on Private Data

M. Sadegh Riazi[†], Ebrahim M. Songhori[*], Farinaz Koushanfar[†]
[†]University of California San Diego, [*]Rice University
mriazi@ucsd.edu, ebrahim@rice.edu, farinaz@ucsd.edu

ABSTRACT

We propose PriSearch, a provably secure methodology for two-party string search. The scenario involves two parties, Alice (holding a query string) and Bob (holding a text), who wish to perform a string search while keeping both the query and the text private without relying on any third party. Such privacy-preserving string search avoids any data leakage when handling sensitive information, e.g., genomic data. PriSearch provides an efficient solution where two parties only need to interact for a constant number of rounds independent of the query and text size. Our approach is based on the provably secure Yao's Garbled Circuit (GC) protocol that requires the string search algorithm to be described as a Boolean circuit. We leverage logic synthesis tools to generate an optimized Boolean circuit for PriSearch such that it incurs the minimum communication/computation cost. We achieve approximately 2× and 140× performance improvements compared to the best prior non-GC and GC-based solutions, respectively.

Keywords

String Search, Garbled Circuit, Logic Synthesis

1. INTRODUCTION

String search allows one to learn whether or not a query string is present within a usually much larger text. It is a very well-studied problem in computer science, and many ingenious algorithms have been proposed to solve it efficiently. It also has numerous applications in different fields, ranging from search engines to surveillance to genomic data processing. Many of these applications involve processing private information. It is ideal to perform the search while keeping this information secret. For example, string search on genomic data is used in personalized medicine and identifying criminals using FBI Combined DNA Index System (CODIS)[1]. Meanwhile, an individual's genomic data carries highly sensitive information about her and her family. Therefore, it is necessary to devise a solution for genomic string search with respect to certain security and privacy requirements. Another example is a government surveillance system that can search patterns or particular words (e.g., "bomb") in chat logs, text messages, and documents without undermining citizens' privacy.

The prior work on string search typically aims to reduce execution time or memory usage, and less focus has given to security and data privacy. A number of obfuscation and anonymization approaches have been proposed, but it has been shown that they have serious pitfalls [6, 9]. The ultimate solution is to keep the query and the text private to their respective owners. Such *privacy-preserving* string search allows two parties, Alice holding the query and Bob holding the text, to perform string search while keeping both the query and text private.

There have been a few attempts to solve the privacy-preserving string search [1, 4, 19]. The shortcomings of the previous work can be categorized as follows: (i) Lack of solid security proofs for heuristic and custom designed protocols [6, 9]. (ii) The number of communication rounds increases as the sizes of text and query increase [20]. (iii) Prohibitive computation overhead, e.g., solutions based on Homomorphic Encryption (HE) require several expensive asymmetric encryptions [3]. (iv) Extension to support other variants of the string search, e.g., providing the number of matches and longest prefix match is not trivial [19].

PriSearch provides an efficient solution for privacy-preserving string search based on Yao's Garbled Circuit (GC) protocol. GC is a provably secure protocol that allows two parties to evaluate a function on their private inputs. A challenging step in the GC protocol is to convert the underlying function (here the string search algorithm) into a Boolean circuit such that it incurs minimum communication/computation cost. Utilizing the GC protocol guarantees the privacy requirements, however, a naive implementation of string search as a Boolean circuit results in a huge cost because it requires multiple random accesses to the text (see Section 3.2). Here, the address of a random access depends on the query and the text, thus, it has to be hidden from both parties, a costly operation in GC which is called *oblivious access*. A *single* oblivious access to the text has linear computation and communication complexity $\mathcal{O}(n)$ with respect to the text size (n). The cost of oblivious access can be improved using Oblivious RAM (ORAM) inside GC to achieve polylogarithmic complexity $\mathcal{O}(polylog\ n)$ per access. Sadly, the ORAM scheme with the best asymptotic complexity [20] needs at least $\mathcal{O}(\log n)$ sequentially dependent rounds of communication per access, making the overall execution prohibitively slow.

We introduce PriSearch, a novel privacy-preserving string search which does not need random access to the text and it has constant round complexity ($\mathcal{O}(1)$). The computations required in PriSearch are mainly based on symmetric key encryption and are far less expensive compared to HE. We also design new synthesis libraries and leverage logic synthesis tools to generate an optimized sequential circuit for PriSearch automatically to achieve minimal cost in GC. Moreover, PriSearch is designed such that it can be easily extended to support multiple variants of string search. In brief, our contributions are as follows:

- Analyzing five major string search algorithms and their performance in the GC protocol and providing their respective complexity.
- Developing a string search algorithm (based on the

[1] https://www.fbi.gov/services/laboratory/biometric-analysis/codis

DAC '17, June 18-22, 2017, Austin, TX, USA
© 2017 ACM. ISBN 978-1-4503-4927-7/17/06…$15.00
DOI: http://dx.doi.org/10.1145/3061639.3062305

Knuth-Morris-Pratt algorithm) that does not require random access to the text.

- Automatically generating the optimized Boolean circuit for string search by leveraging logic synthesis tool and new synthesis libraries.
- Performing extensive experiments on six different benchmarks and achieving approximately 2× and 140× performance improvements compared to the best prior non-GC and GC-based solutions, respectively.

2. PRELIMINARIES AND BACKGROUND

2.1 Problem Statement and Notation

The set of all alphabet letters is denoted as Σ. For example, in genomic processing, the alphabet set is $\Sigma = \{$'A', 'C', 'G', 'T'$\}$. The size of the alphabet set is defined as $R = |\Sigma|$. Given a query Q and a text T, string search looks for a contiguous substring in T which is equal to Q. More precisely the output o is:

$$o = \begin{cases} 1, & \text{iff } \exists i \mid Q = T[i : i+m-1],\ 0 \leqslant i \leqslant n-m \\ 0, & \text{otherwise} \end{cases}$$

where $Q \in \{\Sigma\}^m$ and $T \in \{\Sigma\}^n$ (usually $m << n$). In a two-party privacy-preserving string search, Alice holds Q, and Bob holds T, and the goal is that one or both parties learn o while the contents of Q and T remain secret from the other party (m and n are public).

2.2 Security Model

The security model in this paper is *honest-but-curious* (semi-honest) in which both parties follow the protocol but are curious to extract more information from the data that they receive. There are stronger security models such as security against covert and malicious adversaries where any party can diverge from the protocol at any time. However, honest-but-curious is the standard security model in the literature and is a first step for building protocols with stronger security models. The GC protocol can be readily extended to ensure security against malicious adversary [11].

2.3 Oblivious Transfer Protocol

Oblivious Transfer (OT) is a two-party cryptographic protocol in which Alice has two messages s_0 and s_1 and Bob has one bit b [13]. Bob wants to learn s_b without letting Alice learn his choice b. Bob also should not learn anything about the other message s_{1-b}.

2.4 Garbled Circuit Protocol

Yao's Garbled Circuit (GC) is a two-party protocol in which Alice holds an input a, Bob holds an input b, and they want to evaluate a function $o = f(a, b)$ while keeping their inputs private. At the end of the protocol, one or both parties learn the output o. In the GC protocol, the function f has to be described as a Boolean circuit consisting of 2-input gates. It has been shown that synthesizing a circuit for GC can be viewed as a hardware synthesis task and can be done efficiently using standard synthesis tools [17].

As the first step in GC, for each wire w in the circuit, Alice generates a pair of k-bit *keys* X_w^0 and X_w^1 corresponding to Boolean values 0 and 1, where k is the security parameter and typically is 128. For each gate in the circuit, she replaces the Boolean values in the truth table with the corresponding keys. She encrypts the keys of the output entries using the corresponding input keys of the same row in the truth table. The four rows of the encrypted table are randomly permuted by Alice and together are called a *garbled table*.

Alice sends the garbled tables of all gates to Bob. Alice also sends the keys corresponding to her input a to Bob. For example, if the first bit of her input $a[0]$ is 1 and is connected to input wire w, then she sends X_w^1 to Bob. Bob also receives his input keys through the OT protocol from Alice without letting Alice learn his input. Next, for each gate, given the two input keys, Bob decrypts one row of the garbled table and achieves the output key. Bob continues decryption until he reaches the final output wires. At the end, Alice has the mapping of the output keys to the actual Boolean values and Bob has the output keys. Then, they can share their information to learn the output o.

2.5 Garbled Circuit Protocol Optimizations

Here are some of the optimizations used in PriSearch:

- Row Reduction [12]: This method reduces the number of rows in the garbled tables from four to three and as a result, reduces the communication by 25%.
- Free XOR [10]: This technique eliminates the encryption, decryption, and communication of garbled tables for XOR gates, making them almost free of cost. Any Boolean function can be represented with 2-input AND and XOR gates. Therefore, the computation/communication costs of the GC protocol is proportional to the number of AND gates in the Boolean circuit.
- Half-Gate [22]: This optimization reduces the number of rows in the garbled table from three to two for AND gates, reducing the computation and communication costs by almost 33%.
- Garbling with fixed-key block cipher [2]: This method uses a fixed-key block cipher as encryption method instead of previously used costlier hash functions.
- The GC protocol in its simplest form supports only combinational circuits. Recently, [17] introduced an approach for garbling/evaluating sequential circuits. A sequential circuit includes memory elements (flip-flops) along with 2-input gates and has to be garbled for a predetermined number of clock cycles. Sequential circuits allow users to create a more compact representation of the underlying function compared to the combinational circuits.

3. PRISEARCH

In the GC protocol, the underlying function that is evaluated securely has to be described as a Boolean circuit. In PriSearch, the function is string search and the inputs are the text and the query. There are several algorithms for string search and each one incurs a different cost if described as a Boolean circuit for the GC protocol. In the following, we explore these algorithms and in Section 3.2, their computation/communication costs in GC are discussed.

3.1 String Search Algorithms

We briefly describe and compare brute-force and four most efficient string search algorithms [16].

- **Brute-force**: The simplest algorithm is to start from the beginning of the text and compare it with the query. If the first characters match, proceed to the next ones. If all characters have been matched, then the algorithm returns true. Upon a mismatch, the starting pointer of the text is incremented by one and the search is continued until the end of the text. This algorithm has $\mathcal{O}(mn)$ computational complexity.
- **Rabin-Karp**: This algorithm compares the hash value of the query against the hash value of each part of text of size m. The elementary implementation incurs complexity of $\mathcal{O}(mn)$ but using a more efficient hash

Table 1: Summary of characteristics for different string search algorithms.

Algorithm	Non-random access to the text	Low-cost operations	Linear-size pre-computed information
Rabin-Karp	✓	-	✓
KMP	-	✓	✓
Booyer-Moore	-	✓	✓
FSM-based	✓	✓	-
PriSearch	✓	✓	✓

computation, called Horner's method, the complexity can be reduced to $\mathcal{O}\big(n\big)$.

- **KMP**: Knuth-Morris-Pratt algorithm is similar to brute-force algorithm. However, upon a mismatch, the pointer on the text is not incremented by one and instead, it jumps for a certain value based on a precomputed information. The complexity of KMP is $\mathcal{O}\big(n\big)$.
- **Boyer-Moore**: This algorithm starts the comparison from the *end* of the query. If it matches, it compares the second to the last character and so on. If a mismatch happens, it chooses the next character for comparison in the text based on the mismatched character. On average, it takes $\mathcal{O}\big(n/m\big)$ to finish and is considered one of the best algorithms in the literature but in the worst-case scenario, it takes $\mathcal{O}\big(nm\big)$.
- **FSM-based**: The core idea is to create a Finite State Machine (FSM) based on the query and feed the FSM with one character of the text at a time. Based on the current state of the FSM (number of matches so far) and the new input character of the text, the next state is chosen. If it reaches the final state, a match is found. Its computational complexity is $\mathcal{O}\big(n\big)$.

Precomputed information. Beside the computational complexity, the other difference between these algorithms is the size of the precomputed information that they need. In the GC protocol, precomputed data has to be accessed without revealing any information about the text or query. This access cost increases with the size of the precomputed information. Here, we analyze the size of precomputed information for all of the algorithms.

Brute-force algorithm does not require any precomputed data. Rabin-Karp requires $\mathcal{O}\big(1\big)$ precomputed information, only a hash value is computed and stored. In KMP algorithm, once a mismatch is identified, the size of the jump on the text is determined based on the mismatch location in the query (m possible locations), whereas, in the Booyer-Moore algorithm, it is based on the mismatched character in the text (R possible characters). Therefore, KMP needs to store $\mathcal{O}\big(m\big)$ precomputed information while Booyer-Moore needs to store $\mathcal{O}\big(R\big)$.

FSM-based algorithm makes the decision based on the current state (m possible states) and the input character of the text (R possible characters). Thus, it must store mR possible combinations ($\mathcal{O}\big(mR\big)$ precomputed information).

3.2 String Search in GC

We have identified three main requirements for efficient realization of string search in the GC protocol.

1. Non-random access to the text. In Booyer-Moore and KMP algorithms, the access sequence of the text depends on the content of the text and the query. Therefore, the access pattern should be hidden in order to keep the text and the query private. This can be done in the GC protocol using a random access memory (multiplexers and flipflops) in the Boolean circuit. The computation/communication cost of naive implementation of random access memory inside GC is $\mathcal{O}\big(n\big)$ per access. Therefore, complexities of Booyer-Moore and KMP algorithms in GC are $\mathcal{O}\big(n^2m\big)$ and $\mathcal{O}\big(n^2\big)$, respectively.

A better solution for random access memory in GC is by utilizing Oblivious Random Access Memory (ORAM). The ORAM scheme with the best asymptotic complexity incurs $\mathcal{O}\big(polylog(n)\big)$ access cost [20]. However, each access to ORAM requires $\mathcal{O}\big(\log n\big)$ sequentially dependent rounds of communication between two parties. Considering the network latency, ORAM makes the overall private search extremely slow and limits its scalability for large n. In contrast, the access sequence to the text in Rabin-Karp and FSM-based algorithms do not depend on the content of the text nor the query. In these algorithms, each character of the text is accessed strictly in order and only once. As such, the costly random access memory is not required. Therefore, the most efficient string search algorithm for GC has to have a non-random access pattern to the text.

2. Low-cost operations. The big O notation expresses how the private search protocol scales as n and m grow. However, the constant coefficient hidden in the Big O notation plays an important role in the overall performance. The constant coefficient corresponds to the computation that needs to be performed for processing one character of the text. In the sequential GC, the constant coefficient is proportional to the size of the Boolean circuit implemented for the string search algorithm. Rabin-Karp algorithm includes modular exponentiations and multiplications that result in a very large circuit size. In contrast, Our objective is to construct the smallest possible circuit.

3. Linear-size precomputed information. The FSM-based algorithm meets the first two requirements. However, a precomputed information of size $\mathcal{O}\big(mR\big)$ has to be accessed once for processing each character of the text, yielding the $\mathcal{O}\big(nmR\big)$ complexity. Our goal is to have the minimum precomputed data possible to reduce the overall complexity. Rabin-Karp is the only algorithm that needs $\mathcal{O}\big(1\big)$ precomputed information. In the Rabin-Karp algorithm, once the hash value of a part of the text becomes equal to the hash value of the query, a final check should take place to verify a true match between the two strings. In GC, one cannot distinguish whether a hash match occurred or not, hence, the final check has to be performed in every iteration. This makes Rabin-Karp algorithm's performance in GC even worse than the brute-force algorithm.

None of the above algorithms has $\mathcal{O}\big(1\big)$ precomputed information that also satisfies the first two requirements. Thus, our goal is to have an algorithm with linear size precomputed information in terms of m (or R). Table 1 summarizes the characteristics of different string search algorithms.

3.3 PriSearch Algorithm

We design PriSearch, a string search algorithm, based on KMP that satisfies the three aforementioned requirements for efficient privacy-preserving string search. Algorithm 1 shows the pseudocode of the PriSearch algorithm. The algorithm is implemented as a sequential Boolean circuit in order to be evaluated in the GC protocol. Each iteration of the algorithm is computed in one clock cycle of the sequential circuit. As shown in the Algorithm 1, at each clock cycle, a certain character of the text ($T[l]$) is accessed. The index of this character, l, increases by one regardless of the content of the query and the state of the matching process at each iteration (Requirement 1). The index i shows which character of the query Q is compared against $T[l]$ at each

Table 2: Summary of different algorithms for string search and their complexity; $R = |\Sigma|$, n is the number of characters in the text, and m is the number of characters in the query. The missing complexities in the fifth column means utilizing ORAM does not improve the performance.

Algorithm	Plaintext Average-case Complexity	Plaintext Worst-case Complexity	GC Complexity ($\mathcal{O}(1)$ Round Complexity)	GC + ORAM Complexity ($\mathcal{O}(n \log n)$ Round Complexity)	Size of Precomputed Information
Brute-force	$\mathcal{O}(nm)$	$\mathcal{O}(nm)$	$\mathcal{O}(nm \log R)$	-	-
Rabin-Karp	$\mathcal{O}(n)$	$\mathcal{O}(nm)$	$\mathcal{O}(nm \log R)$	-	$\mathcal{O}(1)$
KMP	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n(n+m+\log R))$	$\mathcal{O}(n(\log^3 n + \log^3 m + \log R))$	$\mathcal{O}(m)$
Booyer-Moore	$\mathcal{O}(n/m)$	$\mathcal{O}(nm)$	$\mathcal{O}(nm(n+m+R+\log R))$	$\mathcal{O}(nm(\log^3 n + \log^3 m + \log^3 R + \log R))$	$\mathcal{O}(R)$
FSM-based	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(nmR)$	$\mathcal{O}(n(\log^3 mR))$	$\mathcal{O}(mR)$
PriSearch	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(nm \log R)$	-	$\mathcal{O}(m)$

iteration. PriSearch algorithm starts by comparing the first character of the text ($l = 0$) with the first character of the query ($i = 0$). If they match, i is incremented by one. If they do not match, similar to KMP algorithm, the algorithm uses a precomputed array P in order to update i for the next iteration. The algorithm finishes when l reaches the n.

Algorithm 1 PriSearch.

Inputs: Bob's text T, Alice's precomputed array P and query Q.
Output: Output o indicating match has been found or not.

```
$i = 0$
$o = 0$
// processing one character of the text per clock cycle
for $l = 0$ to $n - 1$ do
    $c = T[l]$
    if $Q[i] == c$ then
        // next character match
        $i = i + 1$
        if $i == m$ then
            // complete string match is found
            $o = 1$
        end if
    else if $Q[P[i]] == c$ then
        // mismatch, check the array $P$ to see where
        // to check next
        $i = P[i] + 1$
    else
        // total mismatch, restart from beginning of $Q$
        $i = 0$
    end if
end for
```

Figure 1 shows an example to illustrate the steps in PriSearch. The final position of the matched substring is shown with a solid green line. After the first four characters (shown in blue color) are matched, a mismatch happens (shown in red). At this moment, it is not possible to reset i to zero and l to one, similar to the brute-force algorithm, because l is automatically increased by one. This means that one cannot access $T[1:3]$ to restart the search process. However, one can leverage the knowledge that $T[0:3]$ is equal to $Q[0:3]$ to properly update i.

Array P stores the index of the next character to compare (new i) based on the current mismatch position (i). Since the mismatch occurs at $i = 4$ and given that $P[4] = 2$, $T[l]$ has to be compared against $Q[2]$, i.e., updating i to 2. Thereby, $T[l] = T[4] =$'A' is compared with $Q[P[4]] = Q[2] =$'A'. This comparison has to be done in the current iteration because $T[4]$ cannot be accessed in the next iteration (unlike KMP algorithm). Since $T[4]$ and $Q[2]$ are equal, in the next iteration $T[5]$ is compared with $Q[3]$ ($i = P[i] + 1 = 3$). $Q[3:4]$ also match in the following iterations and the output o becomes 1. The only operations used in PriSearch algorithm are the comparison of two characters and addition by one which are far less expensive operations than modular exponentiations and multiplications (Requirement 2). We design a new synthesis libraries that include optimized implementations of these operations (i.e., addition, comparison, and selection) for the GC protocol.

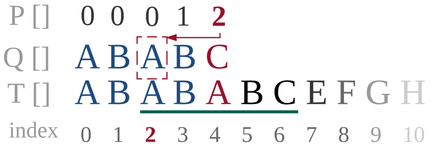

Figure 1: Illustration of steps in PriSearch algorithm.

Array P is precomputed based on query Q using Algorithm 2, also used in KMP. $P[i]$ is used when query has been matched for the first i characters but not for the $(i+1)^{th}$ character. Upon a mismatch, the search needs to be restarted from $T[l - i + 1]$. However, instead of $T[l-i+1 : l-1]$, one can use $Q[1 : i-1]$ to search against the query itself since $Q[0 : i-1]$ is equal to $T[l-i : l-1]$. The content of array P is independent of the text and only depends on whether or not any prefix of the query is repeated within itself. In PriSearch, Algorithm 2 is done offline and locally by Alice, the owner of the query. Therefore, precomputation is performed without engaging in the GC protocol and it takes several orders of magnitude less time than the main string search. Along with the query Q, Alice provides the precomputed array P of size m (Requirement 3) as her secret inputs to the GC protocol.

Table 2 summarizes the complexity of different string search algorithms in plaintext (average-case and worst-case) and in GC. Plaintext complexity is the computational complexity of the algorithm when it runs locally without the GC protocol. GC complexity provides computation/communication complexity when an algorithm is described as a Boolean circuit and garbled/evaluated in the GC protocol. The difference between plaintext and GC complexities arises from two reasons: (i) In plaintext, for a given conditional (`if`) statement in the algorithm, only the taken branch of the statement is evaluated. In contrast, in the GC protocol, both branches should be garbled/evaluated in order to avoid information leakage. (ii) Random access to an array of size n has an $\mathcal{O}(1)$ complexity in plaintext, whereas in the GC protocol, as mentioned before, it has $\mathcal{O}(n)$ when using naive memory (GC complexity column) and $\mathcal{O}(polylog(n))$ when using state-of-the-art ORAM [20] (GC + ORAM complexity column). However, ORAM needs $\mathcal{O}(\log n)$ rounds of communication for a single memory access. This means the overall round complexity using ORAM is $\mathcal{O}(n \log n)$, which

Algorithm 2 Precomputing array P, done by Alice.

```
Input: Query Q.
Output: Array P.
$P[0] = 0$
if $length(Q) > 1$ then
    $P[1] = 0$
end if
$pos = 2$ //current position filling in P
$ind = 0$ //zero-based index in Q
while $pos < length(Q)$ do
    if $Q[pos - 1] == Q[ind]$ then
        //case 1: substring continues
        $P[pos] = ind + 1$
        $ind = ind + 1$
        $pos = pos + 1$
    else if $ind > 0$ then
        //case 2: it doesn't match, fall back
        $ind = P[ind]$
        $P[pos] = 0$
    else
        //case 3: run out of candidates, ind = 0
        $P[pos] = 0$
        $pos = pos + 1$
    end if
end while
```

considering the network latency, makes the overall string search prohibitively slow. On the contrary, the GC protocol without ORAM has a constant round complexity of $\mathcal{O}(1)$. As can be seen, our algorithm achieves the best GC complexity with a constant round complexity.

The Boolean circuit of PriSearch is automatically generated using a standard synthesis tool. The synthesizing constraints are set such that the number of AND gates in the circuit becomes minimum because according to the Free XOR optimization [10], XOR gates do not incur any cost.

3.4 Different Variants of PriSearch

Up to this point, we have focused on the simplest form of the protocol, which is string search with 1-bit output, indicating whether the query is present in the text or not. We now show how PriSearch can be readily adapted to yield other variants of privacy-preserving string search. We briefly mention two of these variants together with the required changes to the main algorithm:

- **Providing the total number of matches**: One can keep track of the total number of matches using a counter. To do this, *counter* needs to be initialized and the following line has to be inserted after Line 10 in Algorithm 1:

  ```
  11:    $counter = counter + 1$
  ```

- **Size of longest prefix match**: A very important and useful information in the string search is the length of the longest prefix of query matched in the text. For example, having "ABCD" as query and text as "ABCEFG", the algorithm should output 3. Although an exact match is not found, a prefix of size 3 ("ABC") is present in the text. The change required for this is to have another value, say *prefix*, and insert the following lines after Line 8:

  ```
  09:  if $i > prefix$ then
  10:      $prefix = i$
  11:  end if
  ```

3.5 Security of PriSearch

The GC protocol is proven to be secure against honest-but-curious adversaries for any function $f(a, b)$ [2]. In our setting, the function is the PriSearch algorithm, a is Alice's input which is composed of precomputed array P and query Q, and b is Bob's input which is the text T. Thus, the security of PriSearch immediately follows from the security of the GC protocol.

4. RESULTS

We perform our experiments using two Intel Core i7-2600 CPU @ 3.4GHz processors, 12GB RAM and 64-bit Ubuntu 14 operating system. The security parameter in our setup is 128-bit ($k = 128$). We describe the PriSearch algorithm in Verilog, a hardware description language. We then use Synopsys Design Compiler 2010 along with our constraints and synthesis libraries to generate its Boolean circuit.

Table 3 shows the results for different string search benchmarks and their comparison with previous work whenever applicable. PriSearch achieves 2× improvement compared to the best prior privacy-preserving string search [1] that is secure against the same adversary model. They report DNA pattern matching with a query of size 100 characters against a text with 100k characters takes 64 seconds [1] while PriSearch finishes the same task under 34 seconds. PriSearch can be used for real-time processing stream of characters. It can securely process up to 15k characters per second when searching for a query of size 10. Comparing our solution with the best GC implementations (utilizing KMP algorithm and state-of-the-art ORAM implementation [20]), we observe more than 140× faster execution when $m = 100$, $n = 64k$, and $R = 256$.

5. RELATED WORK AND COMPARISON

There are several studies addressing secure string search (or pattern matching) due to its significance. A number of earlier work consider securing against malicious and covert adversaries [7, 8]. They incur a significantly larger overhead compared to PriSearch. Such a comparison is not always straightforward or fair due to the difference in the adversary models. Thus, we report the comparison of PriSearch to protocols that are secure against the honest-but-curious adversary model.

De Cristofaro et al. introduced a secure pattern matching protocol based on Additively Homomorphic Encryption (AHE) for genomic applications [3]. They have implemented their protocol based on two different AHE libraries, namely AH-ElGamal and EC-ElGamal. The most expensive part of their approach is initial encryption of the genome data which as they have reported takes 115 and 2,580 hours using AH-ElGamal and EC-ElGamal respectively for genome data with 3 billion letters. An additional 2.7 hours and 8.7 hours of data transmission are required given a 1Gbps link. In contrast, our approach takes almost 80 hours for finding a pattern considering the same text size and communication bandwidth. Besides, they (and also [14]) consider searching in a particular predetermined part of the genome, whereas, we search the entire genome.

Baron et al., propose to reduce the problem of substring search to a series of linear operations (e.g., inner products and matrix multiplication) such that the operations can be efficiently computed using AHE [1]. As shown in Table 3, PriSearch is 2× faster compared to their approach.

A number of studies [8, 19] rely on oblivious polynomial evaluations to perform binary substring search. However, they cannot search on non-binary alphabet like genomic data. Similarly, the method proposed in [21] can only find

Table 3: Timing and communication results for different benchmarks.

Benchmark	n	m	$\log_2(R)$	Precomp.(ms)	Online Comp.(s)	Comm.	Exec.(s)	Prior Art(s)	Improv.
DNA Matching 1 [4]	10†	10k	2 bit	1.38	0.15	55.34MB	0.53	8	15×
DNA Matching 2 [1]	100k	100	2 bit	0.03	9.30	3.50GB	33.84	64	1.9×
Document Search	64k	100	8 bit	0.03	12.51	4.74GB	45.49	6424‡	141×
Longest Prefix Search	100k	10	32 bit	0.01	6.26	2.41GB	23.16	-	-
1MB ASCII Document Search	1M	10	8 bit	0.01	18.72	7.21GB	69.37	-	-
1% Human Genome Search††	30M	50	2 bit	0.02	21.65min	49.39GB	1.32h	-	-

†Per each 10 characters of the text as reported in [4]. ‡ State-of-the-art GC+ORAM Impl. [20]. ††In practice, processing properly chosen 1% of human genome yields comparable accuracy to processing the entire genome (3B letters) [5].

patterns in binary vectors and is based on *somewhat homomorphic* encryption.

Another approach is to construct an automaton based on the query and then obliviously evaluate it on the text as proposed in [4,18]. The method in [4] is two to three orders of magnitude faster than the one proposed in [18] and as shown in Table 3, our method is 15× more efficient than [4].

Please note that we cannot utilize the "Early Termination" technique [15] since the runtime of the protocol would reveal the index of the match which as a result reveals the query.

6. CONCLUSION

We introduce PriSearch, an efficient approach for secure string search where both query and text from two parties are kept private. Our approach is based on the provably secure Garbled Circuit protocol. We report 2× performance improvement compared to the state-of-the-art solution [1]. We design a new string search algorithm that makes the access pattern to the text strictly in order, enabling us to search large texts without using expensive Oblivious RAM. The experimental results demonstrate the efficiency of PriSearch and its applicability to real-world problems.

Acknowledgment

This work is supported in part by NSF Trust Hub (CNS-1649423), NSF SRC (CNS-1619261), and MURI (FA9550-14-1-0351) grants to ACES lab at UCSD.

7. REFERENCES

[1] J. Baron, K. El Defrawy, K. Minkovich, R. Ostrovsky, and E. Tressler. 5pm: Secure pattern matching. In *International Conference on Security and Cryptography for Networks*. Springer, 2012.

[2] M. Bellare, V. T. Hoang, S. Keelveedhi, and P. Rogaway. Efficient garbling from a fixed-key blockcipher. In *S&P*. IEEE, 2013.

[3] E. De Cristofaro, S. Faber, and G. Tsudik. Secure genomic testing with size-and position-hiding private substring matching. In *Proceedings of the 12th ACM workshop on Workshop on privacy in the electronic society*. ACM, 2013.

[4] K. B. Frikken. Practical private DNA string searching and matching through efficient oblivious automata evaluation. In *IFIP Annual Conference on Data and Applications Security and Privacy*. Springer, 2009.

[5] J. R. Gibbs and A. Singleton. Application of genome-wide single nucleotide polymorphism typing: simple association and beyond. *PLoS Genet*, 2006.

[6] M. Gymrek, A. L. McGuire, D. Golan, E. Halperin, and Y. Erlich. Identifying personal genomes by surname inference. *Science*, 339(6117), 2013.

[7] C. Hazay and Y. Lindell. Efficient protocols for set intersection and pattern matching with security against malicious and covert adversaries. In *Theory of Cryptography Conference*. Springer, 2008.

[8] C. Hazay and T. Toft. Computationally secure pattern matching in the presence of malicious adversaries. In *ASIACRYPT*, 2010.

[9] N. Homer, S. Szelinger, M. Redman, D. Duggan, W. Tembe, J. Muehling, J. V. Pearson, D. A. Stephan, S. F. Nelson, and D. W. Craig. Resolving individuals contributing trace amounts of DNA to highly complex mixtures using high-density snp genotyping microarrays. *PLoS Genet*, 4(8), 2008.

[10] V. Kolesnikov and T. Schneider. Improved garbled circuit: Free XOR gates and applications. In *Automata, Languages and Programming*. Springer, 2008.

[11] Y. Lindell and B. Pinkas. Secure two-party computation via cut-and-choose oblivious transfer. *Journal of Cryptology*, 25(4), 2012.

[12] M. Naor, B. Pinkas, and R. Sumner. Privacy preserving auctions and mechanism design. In *Proceedings of the 1st ACM conference on Electronic commerce*. ACM, 1999.

[13] M. O. Rabin. How to exchange secrets with oblivious transfer, 2005. Harvard University Technical Report 81 talr@watson.ibm.com 12955 received 21 Jun 2005.

[14] M. S. Riazi, N. K. Dantu, L. V. Gattu, and F. Koushanfar. GenMatch: Secure DNA compatibility testing. In *IEEE HOST*, 2016.

[15] M. S. Riazi, E. M. Songhori, A.-R. Sadeghi, T. Schneider, and F. Koushanfar. Toward practical secure stable matching. *Proceedings on Privacy Enhancing Technologies*, 2017(1):62–78, 2017.

[16] R. Sedgewick and K. Wayne. *Algorithms*. Pearson Education, 2011.

[17] E. M. Songhori, S. U. Hussain, A.-R. Sadeghi, T. Schneider, and F. Koushanfar. TinyGarble: Highly compressed and scalable sequential garbled circuits. In *IEEE S & P*, 2015.

[18] J. R. Troncoso-Pastoriza, S. Katzenbeisser, and M. Celik. Privacy-preserving error resilient DNA searching through oblivious automata. In *CCS*. ACM, 2007.

[19] D. Vergnaud. Efficient and secure generalized pattern matching via fast fourier transform. In *International Conference on Cryptology in Africa*. Springer, 2011.

[20] X. Wang, H. Chan, and E. Shi. Circuit ORAM: On tightness of the Goldreich-Ostrovsky lower bound. In *ACM CCS'15*. ACM, 2015.

[21] M. Yasuda, T. Shimoyama, J. Kogure, K. Yokoyama, and T. Koshiba. Secure pattern matching using somewhat homomorphic encryption. In *Proceedings of the 2013 ACM workshop on Cloud computing security workshop*. ACM, 2013.

[22] S. Zahur, M. Rosulek, and D. Evans. Two halves make a whole. In *EUROCRYPT*. Springer, 2015.