Match Quest: Fast and Secure Pattern Matching

Pranav Jangir
New York University
pj2251@nyu.edu

Nishat Koti
Aztec Labs
nishat@aztec-labs.com

Varsha Bhat Kukkala
IIT Tirupati
varshabhat@iittp.ac.in

Arpita Patra
Indian Institute of Science
arpita@iisc.ac.in

Bhavish Raj Gopal
Indian Institute of Science
bhavishraj@iisc.ac.in

Abstract

Pattern matching (PM) is the technique of identifying occurrences of a short pattern in a long text, where both, the pattern and text, are a string of characters. Since several applications demand the privacy of the pattern and the text in the process of identifying matches, designing secure solutions for PM is gaining popularity. Moreover, given the variety of applications that consider PM, we design secure solutions for three popular variants of PM—exact, wildcard and approximate. Our solutions are designed using the techniques of secure multiparty computation (MPC) in the two-party semi-honest setting. All of our solutions attain a fast response time, which is the time taken from submission of the input to obtaining the output, and forms a crucial parameter when analysing the performance of any protocol. Moreover, our protocols also provide an improved online communication complexity in comparison to prior works. Since determining if two secret-shared values are equal forms a crucial component in all the PM variants, we design a novel constant-round equality protocol in the two-party semi-honest setting. Our equality protocol outperforms all the prior works in the considered setting and can also be of independent interest. We implement all our protocols on the MPC framework of MOTION2NX to showcase the practicality of the designed solutions. In comparison to prior works that consider DNA matching (over 2-bit characters), our pattern matching protocols see improvements of up to 2 orders of magnitude in response time. Our equality protocol, too, excels over all existing constructions. To analyse the performance of our equality protocol in comparison to prior work, we benchmark it for varying input sizes. We observe that with increasing input sizes, the improvement in response time of our protocol keeps on increasing, with improvements of up to 9.7× for 256-bit inputs.

Keywords

Secure computation, secure pattern matching, secure equality

1 Introduction

The problem of identifying occurrences of a given string in a text, commonly referred to as *pattern matching* (PM), finds use in various application scenarios. This includes applications such as text-processing [28], information retrieval [32, 44], intrusion detection systems [23], DNA sequence analysis [29, 63], spam filtering [9], etc. Typically, the applications involve two parties, where one party holds a (small) pattern and wishes to identify every occurrence of this pattern within a (long) text that is held by a different party. Here, both the pattern and text comprise a string of characters. Given the wide variety of applications that rely on pattern matching, there are well-defined variants of the problem that have been explored. For instance, in the case of text retrieval systems, it is often required to identify and retrieve occurrences where the given pattern exactly matches a substring in the text. This is known as *exact* pattern matching and is the most commonly considered variant. On the other hand, there are applications that require the flexibility of searching with errors or allowing a small divergence. In the case of pattern matching with *wildcards*, the flexibility is limited to the extent that certain characters in predetermined positions in the pattern are allowed to match any character in the text. These characters, denoted by $*$ in the pattern, are known as wildcard characters. Pattern matching with wildcards finds use in bioinformatics [18, 26], software patching [11, 36, 62] and DNA analysis [37]. Alternatively, in applications such as face recognition systems [58] and DNA profiling [63], where the position of errors is not known a priori, *approximate* pattern matching is used. Here, when comparing the pattern to a substring in the text, we bound the total number of positions where the pattern and the substring mismatch.

Several applications that rely on pattern matching deal with patterns and text that comprise sensitive user information. This has motivated the need for designing privacy-preserving variants for pattern matching, and has been well studied in the literature [37, 52, 57, 58]. We showcase the need for designing privacy-preserving solutions for pattern matching through the following illustrative use case. Consider a hospital that stores the genomic data of its patients and a research organisation that identifies genetic markers[1]. This organisation may wish to validate that an identified genetic marker corresponds to a specific disease. For this, they may be required to identify the frequency as well as the occurrences of this genetic marker in the genome sequence of patients who have suffered from the said disease. This would require collaboration between the research organisation and the hospital to share their respective data. However, since each organisation considers its data to be sensitive and private, it may not be willing to disclose this information. That is, the hospital wishes to keep the genome data (text) private, and the research organisation wishes to keep its genetic marker (pattern) private. This motivates the need to design secure solutions for pattern matching, which ensure that

[1] A genetic marker is a short DNA sequence that captures a genetic mutation or variation that helps identify diseases.

Proceedings on Privacy Enhancing Technologies 2025(4), 308–328

https://doi.org/10.56553/popets-2025-0132

the pattern is not leaked to the text owner and vice versa. Further, the pattern owner must only learn positions in the text where a match to the pattern is found.

There are various privacy-enhancing technologies that can be used to design privacy-preserving solutions for pattern matching. Keeping efficiency at the centre stage, the current work relies on the technique of secure multiparty computation. Secure multiparty computation (MPC) enables a set of n parties to jointly compute a function on their private inputs while guaranteeing that no subset of at most $t < n$ parties, controlled by an adversary, learns anything other than the function output. In the current scenario, the private inputs are the pattern and the text against which matches are to be identified. The function to be computed securely is pattern matching. We next highlight our contributions which revolve around designing efficient MPC protocols for pattern matching.

1.1 Our Contributions

To ensure wider usability, we design secure protocols for the three popular variants of exact, wildcard and approximate pattern matching. Since determining if two values are equal to each other forms a key operation across all the variants, we additionally design secure *equality check* protocol for the same, which can be of independent interest. When designing these protocols, we make several choices that drive toward efficient constructions. The objective and the design choice made to achieve it are described next.

– *Lightweight clients:* To facilitate participation of lightweight data owners and reduce computational overhead, we design protocols in the secure outsourced computation (SOC) setting. This entails hiring powerful servers to carry out the MPC protocol on inputs (pattern and text) which are *secret-shared*[2] among the servers by the respective owners. Our work considers a *two* server model, where protocols are designed in a 2-party computation (2PC) setting.

– *Fast response time:* Since several applications such as intrusion detection and DNA analysis, are time-sensitive, designing MPC protocols with a fast response time is crucial. That is, the time taken from submitting the inputs to generating the output must be minimised. To aid in this, the protocols are cast in the *preprocessing* model where the compute-intensive input-independent computations are pushed to the *preprocessing phase*[3]. This facilitates fast input-dependent computations in the *online phase*, leading to a fast response time. To further aid in improving online efficiency, we aim to design protocols with constant online rounds, while keeping the communication and computation cost minimal. Furthermore, the protocols are designed to operate over the *ring* algebraic structure that leverages the system architecture [20, 38, 51] to improve overall efficiency.

We next elaborate on our contributions in designing the protocols for the variants of pattern matching (§1.1.1) considered and our generic protocol for equality (§1.1.2), followed by highlights of our benchmarks (§1.1.3).

[2]The input is distributed among servers such that they cannot learn anything about the underlying secret input based on the share that they receive.

[3]Note that the preprocessing phase facilitates performing multiple operations in parallel since the computations are input independent. Thus, the preprocessing phase is not a bottleneck for response time, and we focus on attaining an efficient online phase. Hence, unless stated otherwise, the reported costs in terms of both rounds and communication are always with respect to the online phase.

1.1.1 Pattern matching. To achieve a fast online phase, we design pattern matching protocols that have constant rounds, and highly efficient communication and computation complexity in the online phase. For this, we consider pattern P comprising s_P characters and text T comprising s_T characters. Here, each character is denoted by an ℓ-bit value chosen from the alphabet set $\Sigma \subset \mathbb{Z}_{2^\ell}$. For all three variants of pattern matching, to identify every occurrence of pattern, P is *matched* against every s_P-sized contiguous substring in T. There are $s_T - s_P + 1$ such substrings in T and all these matches can be computed in parallel. Therefore, a constant-round protocol to identify a match will yield a constant-round pattern matching protocol. The equality protocol serves as the main building block for identifying a match. However, despite having a constant-round equality protocol, naively using the same does not suffice to obtain a constant-round pattern matching protocol. The challenges faced in attaining the same, while satisfying distinct matching criteria for each pattern matching variant, are discussed next.

Exact: Recall that this variant of pattern matching checks whether every character in the pattern is an exact match with the corresponding character in the substring of text under consideration. Since there are s_P number of characters to be matched, it would appear that invoking the generic equality check protocol s_P times, on each pair of characters, in parallel, would suffice. However, this additionally requires an overhead of $\log_2(s_P)$ rounds to verify that all the s_P equality check invocations pass before concluding it is a match. In contrast, we design an efficient 2-round protocol for exact pattern matching with $O(s_T)$-bits communication. A detailed comparison of our protocols with prior works appears in Table 1. We note that our work is the first to achieve a communication complexity independent of the pattern size and character size.

Wildcard: Unlike the previous variant, wildcard pattern matching offers the flexibility of having one or more wildcard characters in the pattern which can match with any character in the text. Thus, the challenge is to not only perform a match against an array of character inputs, but also do so in the presence of wildcard characters. A naive solution to this can be obtained similar to the naive approach for exact pattern matching, where we perform a character by character match while accounting for the wildcards. However, this would require one additional round to determine the positions of the wildcard in the pattern. In contrast, we design a wildcard pattern matching protocol that overcomes this. This results in attaining a wildcard pattern matching protocol with the same online complexity of exact pattern matching. A comparison of our protocol with relevant prior works appears in Table 1.

Approximate: Unlike the previous case, where only the presence of a wildcard allows for a mismatch, the approximate variant instead bounds the maximum number of mismatches between the pattern and the substring of the text, and hence is more generic. Thus, we are required to track the number of mismatches, say k, and determine if it is below a predetermined threshold τ. A naive solution would entail invoking our equality check protocol s_P times, on each pair of characters, in parallel, to determine the total number of mismatches. Following this, relying on a comparison protocol [4] would allow determining if the number of mismatches k is lesser than τ. Thus, the naive solution would require 3 rounds (i.e., 2-rounds for

[4]We rely on the state-of-the-art protocol of [16].

Protocol	Reference	Rounds	Communication (bits)		Computation
			Online	Preprocessing	
Exact	[58]†	3	$O(s_T s_P \ell(\kappa + s_P \ell))$	-	$O(s_T s_P \ell)$-OTs
	[63]‡	2	$O(s_T \kappa + s_P \lvert\Sigma\rvert \kappa)$	-	$O((s_T + s_P \lvert\Sigma\rvert)$-HE
	Naive	$\log_2(s_P) + 2$	$O(s_T s_P \ell)$	$O(s_T s_P \ell(\kappa + \ell))$	$O(s_T s_P \log(\ell))$-PRGs
	Ours	2	$O(s_T \lambda)$	$O(s_T(\lambda(\kappa + \lambda)))$	$O(s_T \log(\lambda))$- PRGs + $O(s_T)$-Hashes
Wildcard	[37]	2	$O(s_P \ell + (\kappa\ell + \kappa')s_T))$	$O(s_T s_P(\kappa + \ell))$	$O(s_T)$-Hashes
	Naive	$\log_2(s_P) + 3$	$O(s_T s_P \ell)$	$O(s_T s_P \ell(\kappa + \ell))$	$O(s_T s_P \log(\ell))$-PRGs
	Ours	2	$O(s_T \lambda)$	$O(s_T(\lambda(\kappa + \lambda) + s_P(\kappa + \ell)))$	$O(s_T \log(\lambda))$-PRGs + $O(s_T)$-Hashes
Approximate	[58]†	3	$O(s_T s_P \ell(\kappa + s_P \ell))$	-	$O(s_T s_P \ell)$-OTs
	[63]‡	2	$O(s_T \kappa + s_P \lvert\Sigma\rvert \kappa)$	-	$O((s_T + s_P \lvert\Sigma\rvert)$-HE
	Naive	3	$O(s_T s_P \ell)$	$O(s_T s_P \ell(\kappa + \ell)$	$O(s_T s_P \ell)$-PRGs
	Ours	1	$O(s_T \ell)$	$O(s_T s_P \ell(\kappa + \ell)$	$O(s_T s_P \ell)$-PRGs

† The OTs in the online phase of [58] can be preprocessed. Despite this, the online communication cost of the protocol is $O(s_T {s_P}^2 \ell^2)$ which is prohibitive.
‡ [63] provides an HE based solution for exact and approximate pattern matching which has a high computational overhead.
ℓ- size of the character; s_T - size of the text; s_P - size of the pattern; Σ - alphabet set; $|\Sigma|$ - the size of the alphabet set. Note that the number of entries in the alphabet set $|\Sigma|$ may only be a proper subset of 2^ℓ. Hence $\Sigma < 2^\ell$. λ denotes the hash size (256 bits), κ denotes the computational security parameter (128) and κ' denotes the statistical security parameter (40). Note that ℓ is application-specific(2 bits for DNA-matching, 8 bits for ASCII). s_P ranges from some Bytes to several KiloBytes, making $s_P.\ell$ much greater than λ. All protocols are computationally secure and in semi-honest two party computation setting.

Table 1: Comparison of pattern matching protocols.

equality as described in §1.1.2 and 1-round for comparison) and communication complexity of $O(s_T s_P \ell)$-bits. However, we design a approximate pattern matching protocol that requires only 2 rounds and $O(s_T \ell)$ communication. Further, by taking the inputs in an encoded fashion, we reduce the complexity to 1 rounds. A comparison of our protocol with prior works appears in Table 1.

1.1.2 Equality. Equality forms an important primitive for pattern matching. To this end, we design a novel constant-round equality check protocol over rings in the 2-party setting. Specifically, our protocol incurs 2 rounds and $2(\ell + \log_2(2\ell))$ bits of communication in the online phase. A comparison of our equality protocol with the relevant prior works that operate in 2 party setting over rings is provided in Table 2. This can be of independent interest as equality check is used in various applications such as heavy hitters [4, 33], ML inference [51], and decision tree training [1], to name a few.

Ref.	Rounds	Communication		#PRG invocations†
		Online	Preprocessing	
[12]	$\log_2(\ell)$	$\approx 4\ell$	$O(\ell\kappa)$	-
[50]	$\log_4(\ell)$	$\approx \frac{8}{3}\ell$	$O(\ell\kappa)$	-
[16]	1	2ℓ	$O(\ell\kappa^2)$	$O(\ell)$
Ours	2	$2(\ell + \log_2(2\ell))$	$O(\ell(\ell + \kappa))$‡	$O(\log(\ell))$

†- number of local sequential PRG invocations in the online phase.
‡- note that $\ell\kappa$ is the dominating term here.
ℓ denotes the size of the input.

Table 2: 2PC equality protocols over rings.

Alternative variant: Our equality protocol works over the additive sharing semantics. However, some works [51] design equality protocols over augmented additive sharing semantics (see §3) to achieve better communication. Our equality protocol also extends to operate on augmented sharing semantics which allows for an improved online communication complexity of $O(\log \ell)$ bits in two rounds. Our augmented sharing-based equality protocol outperforms the same of [51], since the latter requires higher online communication of approximately $5\ell/3$ bits and $O(\log_4(\ell))$ rounds. We believe that this equality protocol may be of independent interest, such as for applications considered in [51].

1.1.3 Benchmarks. We establish the practicality of all our protocols by empirically evaluating them on MOTION2NX [17] framework.
– *Pattern matching*: We compare the performance of each variant of our pattern matching protocols with their respective naive variants while varying s_T and s_P. We report below the improvements observed for s_T = 10KB, s_P = 1KB and $\ell = 8$[5]. In comparison to the naive variants, our solution for exact and wildcard pattern matching achieves up to 3 orders of magnitude improvement in both response time and communication. With respect to approximate pattern matching, we witness an improvement of up to 2 orders of magnitude in communication. We also compare our protocols to the prior work and report the performance while retaining the settings considered in the prior work. With respect to [63], we see an improvement of 100× and 10× for exact and approximate pattern matching respectively. Due to the lack of concrete costs and the absence of code in [58], we are unable to provide such a comparison. Despite having improved communication over [37] for larger values of ℓ, we are unable to provide an empirical comparison since the implementation of [37] does not allow varying ℓ. To provide a close comparison with works that do not have code, we fall back on the detailed complexities reported in Table 1.
– *Equality*: To showcase the performance of our protocol, we compare against both circuit based approach as well as DPF based equality. Specifically, with respect to the circuit based approach of [12], we observe improvements up to 4× in run time and up to 1.9× in communication for inputs of size 256 bits. With respect to the protocol of [50], we observe improvements up to 1.97× in run time and up to 1.28× in communication for inputs of size 256

[5]Since most applications deal with characters encoded in ASCII, we note that $l = 8$ suffices and hence our benchmarks are reported for $l = 8$.

bits. With respect to DPF based approach of [16], we observe improvements up to 9.7× in run time for 256 bits inputs while having comparable communication costs. Further, when considering the augmented additive sharing, we compare our protocol against that of [51] where we observe improvements up to 3.6× in run time and up to 11.2× in communication.

1.2 Organisation

The rest of the paper is organised as follows. We discuss the related works in §2 followed by preliminaries in §3. This is followed by the equality protocol in §4. The exact, wildcard and approximate pattern matching protocols are discussed in §5, §6 and §7, respectively. Finally, we provide the benchmarks in §8. Supplementary information for a comprehensive understanding of our equality and pattern matching protocols such as complexity analysis, security proofs of protocols, other orthogonal works, other prerequisites, etc., appear in appendices §A-G.

2 Related Works

Pattern matching: There are several works in the literature that consider designing privacy-preserving solutions to pattern matching [7, 28, 31, 40, 46, 54, 56, 60, 63]. These works not only differ in terms of the approaches used to design the solution (such as homomorphic encryption, garbled circuits and secret-sharing based techniques) but also differ with respect to the variants of PM considered. For instance, there are several works in the literature that consider homomorphic encryption based solutions [7, 31, 40, 54, 60, 63]. Since these rely on a large number of compute-intensive public key encryption operations, they are known to be computationally expensive and prohibitive. Among the HE-based solutions, [63] considers exact and approximate pattern matching and forms the state-of-the-art for the said variants of PM. A comparison with [63] is hence included in Table 1. Despite having a constant round protocol, the need to perform computationally intensive encryptions renders their solution prohibitively expensive. Specifically, the number of encryptions required is in the order of the size of the pattern and alphabet, making the solution infeasible for large pattern sizes and alphabet sets (such as ASCII). Although, [54] designs a HE-based solution for wildcard pattern matching, the MPC based solution of [37] outperforms [54]. Hence we directly compare against [37] which forms the state-of-the-art.

We next discuss works that consider MPC based solutions. To the best of our knowledge, the work of [30] was the first to consider the problem of secure pattern matching. However, their approach required public key operations and heavy computations in the online phase, making it inefficient. The work of [34] improved upon this by using a modified Yao's garbled circuit for exact pattern matching, which required only a constant number of OPRF invocations. Although the garbled circuit based approach gives a constant round protocol, the communication cost is known to be prohibitively expensive. More recently, the work of [58] used Shamir secret sharing and outsourced OT to achieve constant-round protocols for exact and approximate pattern matching in the secure outsourced setting. However, not only do their protocols have high communication complexity, but they also rely on OTs in the online phase, making the protocols inefficient. It is important to note that even if the OTs can be preprocessed using the offline-online OT paradigm, it does not help in reducing the prohibitively high communication costs in the online phase, which continue to render the protocols inefficient. The protocol in [58] was extended for wildcard pattern matching in the work of [57] by employing cut-and-choose OT. However, the work of [37], which also considers wildcard pattern matching, outperforms [57] as well as the HE-based protocol of [54]. This makes [37] the state-of-the-art for wildcard pattern matching. Despite its improvements, the work of [37] requires a higher online communication cost of $O(s_T \ell)$-bits. Further, unlike our protocol, [37] requires knowledge of s_P and s_T in the preprocessing. This assumption may not be well suited for all application scenarios. Moreover, the protocol of [37] is designed specifically for the client-server setting. That is, to design an efficient 2-round protocol, they leverage the fact that the inputs are held on clear by the two computing parties, and the output can be learnt on clear by the client. It is not clear how to extend their protocol to the outsourced setting. Furthermore, their protocol entails the client performing heavy computations and, thereby, is unsuitable for lightweight clients. On the other hand, our protocols in the secure outsourced setting are more inclusive and cater to a wider set of application scenarios.

Beyond traditional pattern matching, there are other works that explore other variants such as string similarity matching, keyword search with public key encryption, and regular expression matching. We note that these works are orthogonal to ours and a brief discussion of the same appears in A.

Equality: Equality is an important primitive that finds use in various applications. There are several works in the literature that consider MPC-based solutions for designing privacy-preserving equality protocols. The initial works of [22, 41, 48] introduced constant-round equality protocols over fields, by leveraging the properties of field. Among these, [41] forms the state of the art for equality over fields. However, their applicability is limited to fields and cannot be extended to rings. With respect to protocols over rings, [64] designs a 2-round protocol, albeit in the 3-party setting. Further, despite having an honest majority, [64] has communication complexity quadratic in ℓ. Subsequently, the work of [12] proposed a generic circuit-based equality protocol that also works over rings, albeit at the cost of requiring a logarithmic number of rounds, specifically $O(\log_2 \ell)$. Following this, the work of [50] optimised the circuit-based approach using multi-input multiplication gates, thereby reducing the round complexity to $O(\log_4 \ell)$ and communication to $\frac{8}{3}\ell$. The work of [51] further improved the communication to $\frac{5}{3}\ell$ by relying on augmented secret sharing. More recently, the works of [15, 16] introduced a highly efficient equality protocol by leveraging function secret sharing. The protocol relies on distributed point functions(DPFs) to get a constant round equality protocol. Specifically, these works design 1-round protocols that require 2ℓ bits of communication. Despite having an efficient round and communication complexity, DPF-based constructions lag in terms of computation cost. Specifically, the protocol requires each party to locally invoke ℓ sequential calls to a pseudorandom generator (PRG). Instead, our protocol is designed to reduce this to $\log(2\ell)$ invocations. Note that this significantly improves the response time as corroborated by our benchmark results in §8. Further, in the 2-party setting, these protocols suffer from high preprocessing costs,

making them inefficient when dealing with large input sizes. Works such as [21, 25, 45] design generic equality protocols in the n-party setting. While these protocols can be instantiated in the 2PC setting, they are less efficient than the customised 2PC equality protocols discussed above.

3 Preliminaries

Threat model: We design protocols in the 2-party computation (2PC) setting that operate over additive secret sharing. We let $\mathcal{P} = \{P_0, P_1\}$ denote the set of two parties connected via pairwise private and authentic channels over a synchronous network. We assume a static, semi-honest, probabilistic, polynomial time adversary $\mathcal{A}$ that corrupts at most one of the two parties in $\mathcal{P}$. Our protocols are proven secure in the standard real-world/ideal-world simulation paradigm. Further, we design our protocols in the secure outsourced computation setting where two powerful servers[6] are hired to carry out the computation. These two servers enact the role of the two parties in the 2PC. The client(s) secret-shares its input (text T or pattern P) among the servers such that the individual share received by the server does not leak any information about the client's input. The servers run the 2PC protocols for pattern matching and obtain the output in secret shares and reconstruct it towards the intended recipient. Our protocol uses the following sharing semantics.

– Additive sharing ($[\cdot]$-sharing): We say a value $\mathsf{x} \in \mathbb{Z}_{2^\ell}$ is $[\cdot]$-shared or additively shared over $\mathbb{Z}_{2^\ell}$ if P_i for $i \in \{0, 1\}$ holds $[\mathsf{x}]_i$ such that $\mathsf{x} = [\mathsf{x}]_0 + [\mathsf{x}]_1$.

– Augmented additive sharing($[\![\cdot]\!]$-sharing): A value $\mathsf{x} \in \mathbb{Z}_{2^\ell}$ is $[\![\cdot]\!]$-shared if (i) there exists an input-independent mask $\delta_\mathsf{x} \in \mathbb{Z}_{2^\ell}$ that is $[\cdot]$-shared, and (ii) there exists a masked value $\mathsf{m_x} = \mathsf{x} + \delta_\mathsf{x}$ such that $\mathsf{m_x}$ is known to both the parties in $\mathcal{P}$.

Sharing over $\mathbb{Z}_{2^\ell}$ is referred to as arithmetic sharing ($[\cdot]$, $[\![\cdot]\!]$) while over $\mathbb{Z}_2$ as Boolean sharing ($[\cdot]^{\mathbf{B}}$, $[\![\cdot]\!]^{\mathbf{B}}$), where Boolean XOR replaces arithmetic operations (addition/subtraction). Note that the above sharing schemes are linear, i.e., given shares of $\mathsf{x}, \mathsf{y} \in \mathbb{Z}_{2^\ell}$ and public constants $\mathsf{c}_1, \mathsf{c}_2 \in \mathbb{Z}_{2^\ell}$, parties can non-interactively generate shares of $\mathsf{c}_1\mathsf{x} + \mathsf{c}_2\mathsf{y}$.

MPC protocols: We assume parties have access to a common PRF key that is established as part of a setup phase which facilitates them to non-interactively sample common random values among themselves. This allows them to non-interactively generate $[\cdot]$-shares and $[\![\cdot]\!]$-shares for random values. Further, this also allows party P_i to non-interactively generate $[\cdot]$-shares of a value x. For this, the parties jointly sample a random value r using the common PRF key and set the shares as $[\mathsf{x}]_i = \mathsf{x} - \mathsf{r}$ and $[\mathsf{x}]_{1-i} = \mathsf{r}$. Additionally, our protocols rely on a collision-resistant hash function, denoted by $\mathsf{H}(\cdot)$. We rely on the following MPC protocols for designing our protocols—(i) Multiplication (Π_{mult}): Given $[\cdot]$ −shares of values x and y, the multiplication protocol generates $[\cdot]$-shares of $\mathsf{z} = \mathsf{x} \cdot \mathsf{y}$. We rely on the standard beaver protocol [8] for multiplication. The online phase of this protocol requires one round of interaction and 4 elements of communication. (ii) Bit to arithmetic conversion (Π_{bit2A}): Given $[\cdot]^{\mathbf{B}}$-shares of a bit $\mathsf{x} \in \mathbb{Z}_2$, protocol Π_{bit2A} allows to generate its $[\cdot]$-shares. The details appear in §B.

Non-Interactive Multiplication: The multiplication protocol can be made non-interactive when the inputs are taken as augmented additive shares ($[\![\cdot]\!]$-shares) instead of additive shares. We refer to this protocol as $\Pi^{\mathsf{NI}}_{\mathsf{Mult}}$. Given $[\![\cdot]\!]$−shares of values x and y, the protocol $\Pi^{\mathsf{NI}}_{\mathsf{Mult}}$ generates $[\cdot]$-shares of $\mathsf{z} = \mathsf{x} \cdot \mathsf{y}$. The protocol follows along the lines of the multiplication protocol of [51]. While the online phase of this protocol requires one round of interaction, we showcase how it can be made non-interactive when $[\cdot]$-shares of z are desired. Recall that a value x is said to be $[\![\cdot]\!]$-shared if $\mathsf{x} = \mathsf{m_x} - \delta_\mathsf{x}$ such that $\mathsf{m_x}$ is known to both the parties and δ_x is $[\cdot]$-shared (additive) between P_0 and P_1. Thus, the output $\mathsf{z} = \mathsf{x} \cdot \mathsf{y}$ can be computed using the following equation.

$$\mathsf{z} = (\mathsf{m_x} - \delta_\mathsf{x})(\mathsf{m_y} - \delta_\mathsf{y}) = \mathsf{m_x}\mathsf{m_y} - \mathsf{m_x}\delta_\mathsf{y} - \mathsf{m_y}\delta_\mathsf{x} + \delta_\mathsf{x}\delta_\mathsf{y}$$

Observe that parties can non-interactively generate the $[\cdot]$-shares of the first three terms in the above equation due to the linearity of $[\cdot]$-sharing. Additionally, the parties generate $[\cdot]$-shares of $\delta_\mathsf{x}\delta_\mathsf{y}$ in the preprocessing phase by invoking $\Pi_{\mathsf{setupMULT}}$ (from [51]). This enables them to compute $[\cdot]$-shares of $\mathsf{m_z}$ in the online phase non-interactively. The protocol of [51] requires one round of interaction to generate $[\![\cdot]\!]$-shares of z which can be omitted here. The formal protocol for $\Pi^{\mathsf{NI}}_{\mathsf{Mult}}$ appears in Fig. 7.

DPF based equality: Function secret sharing (FSS) [16] allows to succinctly split a function $f(\cdot)$ into additive shares, where each share of the function is represented by a separate key. Each key allows the owner to efficiently generate the additive share of the output $f(x)$ on a given input x. Distributed point functions (DPFs) are a special case of FSS where $f(\cdot)$ is a point function $f_{\alpha,\beta}(x) := \beta$ if $x = \alpha$, or 0 otherwise. A DPF consists of two algorithms: $Gen(\cdot)$ and $Eval(\cdot)$. The $Gen(\cdot)$ algorithm takes as input the function $f_{\alpha,\beta}(\cdot)$ and outputs two keys k_0 and k_1. The $Eval(\cdot)$ algorithm evaluates an input x such that $Eval(0, k_0, x) + Eval(1, k_1, x) = \beta$ for $x = \alpha$, and 0 for $x \neq \alpha$. Privacy ensures (α, β) remains hidden from an adversary in possession of one of the keys (but not both).

Recent works leverage DPFs to design constant round protocols for checking equality in the 2PC-with-a-dealer setting. Elaborately, the protocols work as follows. The dealer generates the keys k_0, k_1 using the $Gen(\cdot)$ algorithm on the point function $f_{1,\mathsf{r}}(\cdot)$ for some random r and distributes it to the 2 parties along with shares of r. Then to check $\mathsf{x} = \mathsf{y}$, where x and y are additively shared between the parties, they first compute $[\mathsf{x}'] = [\mathsf{x}] - [\mathsf{y}]$ and reconstruct $\mathsf{x}' + \mathsf{r}$. Observe that if $\mathsf{x} = \mathsf{y}$ then $\mathsf{x}' + \mathsf{r} = \mathsf{r}$ since $\mathsf{x}' = 0$. Hence, when parties locally evaluate the $Eval(\cdot)$ algorithm on their respective keys, they get additive shares of 1 if $\mathsf{x} = \mathsf{y}$ and additive shares of 0 otherwise. The formal protocol for the same appears in Fig. 9. In the 2PC setting, the dealer can be realised using a distributed key generation protocol. When the input size is not too big, the distributed generation of the keys can be realised with good concrete efficiency using the distributed DPF key generation protocol of [24]. Otherwise, one can use general-purpose secure computation protocols for emulating the dealer [59].

Pattern matching: We let T denote the text comprising $\mathsf{s_T}$ characters, where each character belongs to the alphabet Σ and is an ℓ-bit string. We let P denote the pattern comprising $\mathsf{s_P}$ characters which

[6]We note that in many practical scenarios, the servers are reputed companies(Example–AWS,GoogleCloud) that do not have an incentive to collude, as their reputation is at stake. Hence a semi-honest setting suffices. Moreover, semi-honest behaviour can be enforced by attestation using tools— like Intel SGX or ARM TrustZone.

belong to Σ, where $s_P \leq s_T$. We let $T[i : j]$ denote the substring of the text starting at (and including) the i^{th} character to the j^{th} character. Thus, $T[i : j]$ is a substring of T comprising $j - i + 1$ characters. To keep protocols generic, we instantiate the pattern-matching protocols with ASCII 8-bit character encoding scheme. This encoding includes lowercase and uppercase letters A-Z, numbers 0 to 9, and common symbols. We note that our protocols are generic and can be instantiated with any alphabet set. Other common alphabet sets that are considered in the literature are binary {0,1} and DNA alphabet set {A, T, G, C}.

Hamming distance: We use hamming distance as a measure of similarity between two strings for some of our pattern matching protocols. Given two m-length strings X, Y comprising m characters each, hamming distance $d_H(X, Y)$ measures number of positions in which X and Y have different characters. In general, hamming distance can be computed between two ℓ-bit strings, where it captures the number of bit positions the two strings differ in.

Parameters and notations: Our protocols are instantiated over an ℓ-bit ring $\mathbb{Z}_{2^\ell}$. κ denotes the computational security parameter. For experiments, we use $\ell = 8$, $\kappa = 128$. We use uppercase letters to denote arrays such as X, where $X[i]$ denotes the i^{th} element of X. We assume that for an m length array, the elements are indexed from 1 to m. For an ℓ-bit value $r \in \mathbb{Z}_{2^\ell}$, we let r_i denote the i^{th} bit of r with r_0 denoting its least significant bit. We use 1^ℓ to denote a string of ℓ 1's. We use $1\{x \ominus y\}$ to denote that the output is a 1 if the constraint $x \ominus y$ is satisfied where $\ominus$ is a binary operator, and a 0 otherwise. $\equiv$ denotes the congruence operator, and we use $\log(\cdot)$ to denote the logarithm to base 2, unless otherwise stated.

4 Equality Check

Equality check protocol is an essential primitive for pattern matching. Given two secret shared values $x, y \in \mathbb{Z}_{2^\ell}$, the protocol outputs $[z]^{\mathbf{B}}$ where $z = 1\{x = y\}$. Checking $x = y$ can be reduced to checking $x - y = 0$. Thus, for subsequent discussions, without loss of generality, we restrict our discussion to checking $x = 0$, where x is $[\cdot]$-shared among the parties.

We design a constant round equality check protocol, Π_{Eq}, with linear online communication in the input size (ℓ bits). We take the following approach to achieve this. We reduce checking $x = 0$, for an ℓ-bit input x, to checking $y = 0$ for an $O(\log(\ell))$-bit input y. Given the reduced input length, checking $y = 0$ can be realised efficiently by relying on an existing efficient equality check protocol, which we abstract out as a functionality $\mathcal{F}_{\mathsf{EqZ}}$. We showcase that instantiating $\mathcal{F}_{\mathsf{EqZ}}$ using a DPF-based equality protocol allows Π_{Eq} to attain a constant round complexity with linear communication in the online phase and outperform prior equality protocols [12, 15, 16, 22, 41, 48]. In fact, when operating with augmented additive sharing semantics (§3), we note that the online communication complexity can be further brought down to $O(\log \ell)$ bits (further details are deferred to §C.2).

To check if $x \in \mathbb{Z}_{2^\ell}$ is 0 in Π_{Eq}, we first compute the hamming distance between a random value $r \in \mathbb{Z}_{2^\ell}$ and the masked value $x + r$, followed by checking if this distance is 0. Recall from §3, that the hamming distance between two values $p, q \in \mathbb{Z}_{2^\ell}$ is the number of bits that are different in their bit representation. Let $a = x + r$ for some randomly chosen mask $r \in \mathbb{Z}_{2^\ell}$. Let $c = d_H(a, r)$ be the hamming distance between a and r. Observe that if $x = 0$ then $a = r$ and thus, the hamming distance $c = 0$. If $x \neq 0$, then the bit representation of a and r differ in at least one position. Hence, the hamming distance $c \neq 0$. Thus, checking $x = 0$ can be reduced to checking $c = 0$. Further, observe that a and r can differ in at most ℓ bit positions. Hence, $0 \leq c \leq \ell$. Therefore, c and its $[\cdot]$-shares can be represented in ring $\mathbb{Z}_{2^\ell}$[7]. Thus, if parties generate $[\cdot]$-shares of c over $\mathbb{Z}_{2^\ell}$ (henceforth, sharing over $\mathbb{Z}_{2^\ell}$ is denoted as $[\cdot]^{2^\ell}$), then they can invoke an existing equality protocol, abstracted as a functionality $\mathcal{F}_{\mathsf{EqZ}}$ (Fig. 3), on it. Note that $\mathcal{F}_{\mathsf{EqZ}}$ will be applied on a smaller-sized $O(\log \ell)$-bit input c.

We next explain the steps for computing the $[\cdot]$-shares of c over the ring $\mathbb{Z}_{2^\ell}$ via the hamming distance computation protocol Π_{Ham}. Following this, steps for *non-interactively* translating $[\cdot]$-shares from $\mathbb{Z}_{2^\ell}$ to $[\cdot]^{2^\ell}$-shares over the smaller ring $\mathbb{Z}_{2\ell}$ is given. We then discuss how $\mathcal{F}_{\mathsf{EqZ}}$ can be instantiated efficiently to realise a constant round Π_{Eq} protocol.

Protocol Π_{Ham}: The protocol Π_{Ham} takes as input $[x]$ for $x \in \mathbb{Z}_{2^\ell}$ and outputs $[c]$ (over $\mathbb{Z}_{2^\ell}$) where $c = d_H(x + r, r)$ for some random mask $r \in \mathbb{Z}_{2^\ell}$. Since c denotes the number of bits where $x + r$ and r differ, observe that c can be obtained by computing the XOR of their bits followed by computing the sum of the output of the XOR. To facilitate this XOR computation, the bit representation of r and $x + r$ are required, which are generated as follows. Parties can generate $[\cdot]$-shares of a random $r \in \mathbb{Z}_{2^\ell}$ during the preprocessing phase. In the online phase, once the input $[x]$ is available, they can compute $[x] + [r]$ and reconstruct $x + r$. Given $x + r$ is known on clear, its bit representation can be computed locally. In order to generate the bit representation of r, where r is required to be $[\cdot]$-shared, instead of first sampling $[\cdot]$-shares of r (as described above) and then generating its bit representation, parties first sample random bits r_i for $i \in \{0, \ldots, \ell - 1\}$, and then generate r by composing these bits. Elaborately, parties non-interactively generate $[\cdot]^{\mathbf{B}}$-shares of random bits $r_i \in \mathbb{Z}_2$ for $i \in \{0, \ldots, \ell - 1\}$ (as discussed in §3). Following this, they generate their arithmetic shares, $[r_i]$, using the Π_{bit2A} protocol (§3), and set $[r] = \sum_{i=0}^{\ell-1} 2^i [r_i]$. Having obtained arithmetic shares of bits in r and $a = x + r$, parties compute their XOR by using the arithmetic equivalent of XOR, where $p \oplus q$ can be written as $p + q - 2pq$. Specifically, they compute $[c_i] = a_i \oplus [r_i] = a_i + [r_i] - 2a_i \cdot [r_i]$ and set $[c] = \sum_{i=0}^{\ell-1} [c_i]$. The Π_{Ham} protocol appears in Fig. 1.

The online phase of Π_{Ham} involves performing one reconstruction, requiring communicating 2ℓ bits in 1 round of interaction. The preprocessing phase involves ℓ calls to bit to arithmetic conversion protocol which requires communicating $\ell(\kappa + \ell)$ bits in 2 rounds of interaction. Observe that although the hamming distance $c \leq \ell$, the protocol Π_{Ham} outputs shares of c in the ring $\mathbb{Z}_{2^\ell}$. We next describe how to non-interactively convert the shares of c over the ring $\mathbb{Z}_{2^\ell}$ to shares over $\mathbb{Z}_{2\ell}$.

[7] Note that one can rely on the ring $\mathbb{Z}_\ell$. However, this results in the protocol having a probability of failure that is $1/2^\ell$. This failure probability can be eliminated by relying on the ring $\mathbb{Z}_{2\ell}$. The detailed failure probability analysis when working over $\mathbb{Z}_\ell$ appears in §C.1

Protocol Π_{Ham}

Inputs: $[\cdot]$-shares of a value $x \in \mathbb{Z}_{2^\ell}$.
Outputs: $[\cdot]$-shares of c over $\mathbb{Z}_{2^\ell}$ where $c = d_H(x + r, r)$ for a random $r \in \mathbb{Z}_{2^\ell}$.
Protocol:
Preprocessing:

- Parties sample $[\cdot]^{\mathbf{B}}$-shares of $r_i \in \mathbb{Z}_2$ for $i \in \{0, \ldots, \ell-1\}$
- Parties compute $[r_i] = \Pi_{\mathsf{bit2A}}([r_i]^{\mathbf{B}})$ for $i \in \{0, \ldots, \ell-1\}$
- Parties compute $[r] = \sum_{i=0}^{\ell-1} 2^i \cdot [r_i]$

Online:

- Parties compute $[a] = [x] + [r]$ and reconstruct a
- Parties compute $[c] = \sum_{i=0}^{\ell-1} (a_i + [r_i] - 2a_i \cdot [r_i])$

Figure 1: Hamming distance protocol

Share conversion from ring $\mathbb{Z}_{2^\ell}$ to $\mathbb{Z}_{2\ell}$: Given $[c]$, to generate the additive shares of c over the ring $\mathbb{Z}_{2\ell}$, denoted as $[c]^{2\ell}$, party P_i locally computes $[c]_i^{2\ell} = [c]_i \bmod 2\ell$. The correctness of this is argued below.

Case 1: $c = [c]_0 + [c]_1$ (over $\mathbb{Z}$) : then $[c]_i \leq 2\ell$ for $i \in \{0, 1\}$
Since $c \leq 2\ell$, $\Rightarrow [c]_i \equiv [c]_i \bmod 2\ell$ for $i \in \{0, 1\}$
$\Rightarrow c \equiv c \bmod 2\ell \equiv ([c]_0 + [c]_1) \bmod 2\ell$
$\Rightarrow c \equiv [c]_0^{2\ell} + [c]_1^{2\ell}$

Case 2: $c = [c]_0 + [c]_1 - 2^\ell$ (over $\mathbb{Z}$) : $\Rightarrow c + 2^\ell = [c]_0 + [c]_1$
$\Rightarrow (c + 2^\ell) \bmod 2\ell \equiv ([c]_0 + [c]_1) \bmod 2\ell$
$\Rightarrow c \bmod 2\ell + 2^\ell \bmod 2\ell \equiv [c]_0 \bmod 2\ell + [c]_1 \bmod 2\ell$
$\Rightarrow c \equiv [c]_0^{2\ell} + [c]_1^{2\ell}$ since $2\ell | 2^\ell$ due to the choice of $\ell = 8$

In this way, $[\cdot]^{2\ell}$-shares of c can be generated, followed by invoking $\mathcal{F}_{\mathsf{EqZ}}$ on it. Note that for correctness to hold, 2ℓ should divide 2^ℓ. Hence, we select the smallest ring size that satisfies this condition.

The complete protocol: To summarise, Π_{Eq} takes as input two $[\cdot]$-shared values $x, y \in \mathbb{Z}_{2^\ell}$, and outputs $[\cdot]^{\mathbf{B}}$-shares of bit $z = 1\{x = y\}$. It begins by locally computing $[x'] = [x] - [y]$. This is followed by invoking Π_{Ham} on $[x']$ to compute $[c]$ such that $c = d_H(x' + r, r)$ for a random $r \in \mathbb{Z}_{2^\ell}$. $[c]$ is non-interactively converted to $[c]^{2\ell}$ via the ring change transformation. Finally, $\mathcal{F}_{\mathsf{EqZ}}$ is invoked on $[c]^{2\ell}$ to generate $[z]^{\mathbf{B}}$. Formal Π_{Eq} protocol appears in Fig. 2 with security proof in §G.

Protocol Π_{Eq}

Inputs: $[\cdot]$-shares of $x, y \in \mathbb{Z}_{2^\ell}$.
Outputs: $[\cdot]^{\mathbf{B}}$-shares of a bit z such that $z = 1\{x = y\}$.
Protocol:

- Parties compute $[x'] = [x] - [y]$
- Parties compute $[c] = \Pi_{\mathsf{Ham}}([x'])$ (Fig. 1)
- Party P_i for $i \in \{0, 1\}$ locally sets $[c]_i^{2\ell} = [c]_i \bmod 2\ell$
- Parties invoke $\mathcal{F}_{\mathsf{EqZ}}$ (Fig. 3) on $[c]^{2\ell}$ to generate $[z]^{\mathbf{B}}$.

Figure 2: Equality protocol

Instantiating $\mathcal{F}_{\mathsf{EqZ}}$: $\mathcal{F}_{\mathsf{EqZ}}$ (Fig. 3) takes as input a k-bit value x and outputs $[z]^{\mathbf{B}}$ such that $z = 1\{x = 0\}$. To ensure that Π_{Eq} has linear communication complexity with constant round complexity in the online phase, we instantiate $\mathcal{F}_{\mathsf{EqZ}}$ with a distributed point function (DPF) based equality protocol [16], denoted as Π_{EqZ}, since these are the most efficient with respect to the online complexity. The DPF-based equality protocol requires to communicate $2k$ bits in 1 round of interaction in the online phase. The preprocessing phase of Π_{EqZ} essentially involves generating the DPF keys. For this we rely on the key generation protocol of [24] which requires communication $k(3\kappa + 2)$bits in k rounds of interaction (for k-bit input). Since $\mathcal{F}_{\mathsf{EqZ}}$ is invoked on $\log(2\ell)$-bit input in Π_{Eq}, the online communication cost due to $\mathcal{F}_{\mathsf{EqZ}}$ is only $2\log_2(2\ell)$ bits. Thus, Π_{Eq} requires 2 rounds and $2\ell + 2\log_2(2\ell)$ bits of online communication where the linear communication of 2ℓ bits is due to the reconstruction that occurs as part of Π_{Ham}. It is interesting to note that this linear communication overhead in the online phase can be completely eliminated if one chooses to operate over augmented additive sharing semantics (§3). Elaborately, the reconstruction step in Π_{Ham} can be made non-interactive, thereby eliminating the 2ℓ bits of online communication. This implies that the online cost of Π_{Eq} will mainly be due to the reliance on $\mathcal{F}_{\mathsf{EqZ}}$, which is only $2\log(2\ell)$ bits. Additionally, 2 bits of communication are required to generate the augmented shares from additive shares of z (output of $\mathcal{F}_{\mathsf{EqZ}}$). Further details of this protocol are deferred to §C.2.

Functionality $\mathcal{F}_{\mathsf{EqZ}}$

$\mathcal{F}_{\mathsf{EqZ}}$ interacts with parties in $\mathcal{P}$ and ideal-world adversary $\mathcal{S}$. It receives as input $[\cdot]$-shares of $x \in \mathbb{Z}_{2^k}$ from the parties and $\mathcal{S}$, and proceeds as follows.

- Reconstruct x using the received $[\cdot]$-shares.
- If $x = 0$, set $z = 1$ else set $z = 0$, and generate $[\cdot]^{\mathbf{B}}$-shares of z.
- Send (Output, $[z]_i^{\mathbf{B}}$) to P_i for $i \in \{0, 1\}$.

Figure 3: Ideal functionality for Π_{EqZ}

Π_{Eq} vs. DPF-based equality: As can be observed from above, the online communication and round complexity of DPF-based equality turns out to be better than that of Π_{Eq}. Given this, one may be misled to think that DPF-based equality is better than Π_{Eq}. It is important to note here that despite DPF-based equality having slightly better online communication and round complexity than Π_{Eq}, the main bottleneck in the former is the computation cost. Elaborately, when operating on an ℓ-bit input, it requires ℓ *sequential* PRG computations. On the contrary, since Π_{Eq} invokes the DPF-based equality on a $O(\log(\ell))$-bit input, the number of sequential PRG calls required is only $\log(2\ell))$. This saving in the number of sequential PRG computations allows us to witness a significant saving of around 4× for $\ell = 64$-bit inputs, as evident from the experimental numbers reported in Table 3 (§8). With respect to the preprocessing, we note that using the key generation protocol of [24] requires an exponential (in the input length ℓ) number of PRG computations, which is highly inefficient when $\ell > 16$. While there exist other key generation protocols such as [59] which require only a linear number of PRG calls, they have a very high communication cost. Thus, existing DPF-based equality protocols either have highly inefficient computation cost or high communication cost in the preprocessing phase. On the other hand, since Π_{Eq} invokes the DPF-based equality on $O(\log(\ell))$-bit inputs, the number of PRG calls required in the preprocessing phase, when relying on [24], is only linear therein. In this way, since computation

cost is the bottleneck in DPF-based equality protocols, and we require invoking the latter only on $O(\log(\ell))$-sized inputs, Π_{Eq} turns out to be more efficient than a DPF-based equality.

5 Exact Pattern Matching

We next present a secure two-party protocol for exact pattern matching. The protocol takes as input two arrays representing the text T and the pattern P of size s_T and s_P, respectively. The protocol outputs an array O of size $s_T - s_P + 1$ where the i^{th}-index of O denotes whether P matches the s_P-length subarray $T[i : i + s_P - 1]$. Note that the pattern can only start in positions 1 to $s_T - s_P + 1$ in T since the pattern itself is s_P-characters long. At a high level, the protocol works by searching for the occurrence of the pattern in every $s_T - s_P + 1$ possible position. Note that there exist efficient cleartext algorithms (e.g., Knuth-Morris-Pratt or Boyer-Moore) which leverage information such as positions in the text or pattern that contain the same characters to eliminate searching at certain positions and thereby improve efficiency. However, a secure solution cannot leverage such information to improve efficiency as these algorithms are not data-oblivious and leak information about the pattern and text. Hence, to ensure data-obliviousness a secure solution would require searching for the occurrence of the pattern at every $s_T - s_P + 1$ possible positions. Further note, the occurrence of the pattern in each position in the text can be checked in parallel. Hence, without loss of generality, we next discuss our approach to check the occurrence of the pattern at position i in T.

A straightforward approach to check for the occurrence of the pattern at position i in the text is as follows. Begin by checking the equality of the characters in the pattern with the corresponding characters in the text from position i to $i + s_P - 1$, i.e., check $P[j] \stackrel{?}{=} T[i+j-1]$ for $j \in \{1, \ldots, s_P\}$. Let the result of these equality checks be stored in an array O'_i of size s_P, i.e., $O'_i[j] = 1\{P[j] = T[i+j-1]\}$. Observe that these s_P equalities can be performed in parallel and will require 2 rounds using the equality protocol described in §4. Note that if P occurs at $T[i]$, then each entry in O'_i should be a 1. To check for this, one can compute the AND of all the s_P entries in O'_i. Computation of this AND can be accomplished in $\log(s_P)$ rounds via the tree-based approach. Hence, the total number of rounds required for checking the occurrence of P at $T[i]$ is $2 + \log(s_P)$. To check for the occurrence of P in T, since this check can be performed in parallel at each $T[i]$ for $i \in \{1, \ldots, s_T - s_P + 1\}$, performing exact pattern matching requires $2 + \log(s_P)$ rounds. The protocol for the same appears in Fig. 14. Observe here that despite our constant round equality protocol, naively using it for exact pattern matching does not yield a constant round protocol and instead results in a protocol with round complexity dependent on the size of P.

We instead strive to design an exact pattern matching protocol which has a constant online round complexity and also improves in terms of the online communication and computation cost (Fig. 4). For this, we proceed as follows. Let T_i denote the s_P-length substring of text beginning at position i in T, i.e., $T_i = T[i : i + s_P - 1]$. We compute the difference between the characters in T_i and P and store it in X_i. To check if P matches T_i, our goal boils down to checking if each character in X_i is a 0. While this can be realised by computing the OR of all the entries in X_i using the tree-based approach discussed earlier, this will again require logarithmic number of rounds. Instead, we achieve this in constant rounds as follows.

Since parties have $[\cdot]$-shares of the entries in X_i, observe that if each entry in X_i is 0, then shares of X_i held by one party will be the negative of the shares held by the other party. That is, if $x = 0$ then $[x]_0 + [x]_1 = 0$ and hence $[x]_0 = -[x]_1$. Thus, we let one party, say P_0, compute hash of concatenation of shares of each entry in X_i, while the other party P_1 compute the hash of the concatenation of the negative of its shares. That is, P_0 computes $h_{i0} = H([X_i[1]]_0 \,||\, [X_i[2]]_0 \,||\, \ldots \,||\, [X_i[s_P]]_0)$ and P_1 computes $h_{i1} = H(-[X_i[1]]_1 \,||\, -[X_i[2]]_1 \,||\, \ldots \,||\, -[X_i[s_P]]_1)$. Our goal reduces to checking for equality of h_{i0} and h_{i1}, which can be realised in constant rounds by invoking Π_{Eq} on $[\cdot]$-shares of h_{i0}, h_{i1} of λ bits (hash size).

Observe that this protocol requires 2 rounds in the online phase. Further, it has a communication complexity that is $O(s_T\lambda)$ (where λ denotes the bit-length of the hash), unlike the naive protocol whose complexity is $O(s_T s_P)$. Moreover, observe that the number of calls to Π_{Eq} is also reduced to $O(s_T)$ in comparison to the $O(s_T s_P)$ in the naive solution.

Protocol Π_{ExactPM}

Inputs: $[\cdot]$-shares of text T with s_T characters and pattern P with s_P characters, where each character in these two arrays is $[\cdot]$-shared.
Outputs: $[\cdot]^{\mathbf{B}}$-shares of $(s_T - s_P + 1)$-sized array O, where $O[i] = 1$ if P occurs at position i in T, and 0 otherwise.
Protocol:

- For $i = 1$ to $s_T - s_P + 1$
  - $[T_i] = [T[i : i + s_P - 1]]$, $[X_i] = [T_i] - [P]$
  - P_0 computes $h_{i0} = H([X_i[1]]_0 \,||\, [X_i[2]]_0 \,||\, \ldots \,||\, [X_i[s_P]]_0)$ and P_1 computes $h_{i1} = H(-[X_i[1]]_1 \,||\, -[X_i[2]]_1 \,||\, \ldots \,||\, -[X_i[s_P]]_1)$
  - Parties generate $[\cdot]$-shares of h_{i0}, h_{i1} non-interactively (§3)
  - $[O[i]]^{\mathbf{B}} = \Pi_{\mathsf{Eq}}([h_{i0}], [h_{i1}])$ (Fig. 2)

Figure 4: Exact pattern matching

As described earlier, note that relying on the equality protocol that takes as input augmented secret shares results in inflating the round complexity of Π_{ExactPM}. This is because in Fig. 4, generating $[\![\cdot]\!]$-shares of h_{i0}, h_{i1} (to be fed as input to the equality protocol) will require one round of interaction. This is unlike in the current scenario where generating $[\cdot]$-shares of h_{i0}, h_{i1} can be performed non-interactively.

Output of Pattern Matching: Following along prior works, we designed our protocols to output positions where matches occur. However, the type of output required may depend on the considered application. Alternate output types include revealing only the existence of a match or the count of matches. We note that our protocols can be extended efficiently to cater to these output types as well–(1) count of matches can be computed without additional overhead by summing the output vector (with 1s and 0s) generated in our protocol, (2) existence of a match can be computed by checking if count equals zero, which requires one additional equality check. Looking ahead, we note that these adaptations, to provide alternative outputs, are also applicable to our wildcard and approximate pattern matching protocols.

6 Wildcard Pattern Matching

Unlike exact pattern matching, wildcard pattern matching allows the pattern to include one or more occurrences of a special character, called a wildcard, which can match any character in the text. This wildcard character can be represented as a 0 in the pattern. To account for wildcards, the protocol takes an additional array W of length s_P as input, where $W[j] = 0$ if the j^{th} character in P is a wildcard, and a 1 otherwise. A pattern with wildcards is said to match the substring in the text starting at position i if for all for $j \in \{1, \ldots, s_P\}$, $P[j] = T[i + j - 1]$ or $W[j] = 0$.

Unlike the case of exact pattern matching, the presence of wildcard characters introduces an additional challenge. Recall that in exact pattern matching, it sufficed to compute the difference of the characters in P and T_i denoted as X_i, followed by checking if each entry in X_i is 0. In wildcard pattern matching, however, the presence of wildcards will not result in X_i computed in this way to comprise entirely of all 0s since the wildcard character may not match the corresponding character in T_i. Hence, to account for wildcards, we ensure that before we compute X_i, the characters in T_i that correspond to wildcard positions are forced to be 0. To achieve this, we proceed as follows. Recall that W is defined to have a 0 in positions that contain a wildcard and 1 elsewhere. Hence, to force the characters in T_i corresponding to positions having wildcard characters in P to 0, we perform a component-wise multiplication of elements in W with those in T_i. In this way, for positions where there exists a wildcard character, the corresponding difference between characters in T_i and P in X_i will be 0. For the rest of the positions, if the characters in T_i and P match, only then the entries in X_i will be 0. Thus, if T_i and P match in all positions except for where a wildcard is present, the protocol correctly outputs a 1. Having computed X_i in this way, the protocol can now proceed as in the case of exact pattern matching, where parties compute the hash of the concatenation of the shares of entries in X_i and check for the equality of the hashes.

Protocol Π_{WildPM}

Inputs: $[\cdot]$-shares of pattern P with s_P characters, where each character in the array is $[\cdot]$-shared and $P[i] = 0$ if the i^{th} position in P is a wildcard. $[\![\cdot]\!]$-shares of text T. Additionally, $[\![\cdot]\!]$-shares of array W of length s_P such that $W[j] = 0$ if $P[j]$ is a wildcard character, and 1 otherwise.
Outputs: $[\cdot]^{\mathbf{B}}$-shares of $(s_T - s_P + 1)$-sized array O, where $O[i] = 1$ if P occurs at position i in T, and 0 otherwise.
Protocol:

- For $i = 1$ to $s_T - s_P + 1$
  - $[\![T_i]\!] = [\![T[i : i + s_P - 1]]\!]$
  - Component-wise multiply elements in $[\![T_i]\!]$ and $[\![W]\!]$ via $\Pi^{\text{NI}}_{\text{Mult}}$ to generate $[\![T'_i]\!]$, and set $[X_i] = [\![T'_i]\!] - [P]$
  - P_0 computes $h_{i0} = H([X_i[1]]_0 \,||\, [X_i[2]]_0 \,||\, \ldots \,||\, [X_i[s_P]]_0)$ and P_1 computes $h_{i1} = H(-[X_i[1]]_1 \,||\, -[X_i[2]]_1 \,||\, \ldots \,||\, -[X_i[s_P]]_1)$
  - Generate $[\cdot]$-shares of h_{i0}, h_{i1}
  - $[O[i]]^{\mathbf{B}} = \Pi_{\text{Eq}}([h_{i0}], [h_{i1}])$ (Fig. 2)

Figure 5: wildcard pattern matching

Note that in the online phase, the above protocol requires one round for the component-wise multiplication and 2 rounds for equality check. Instead, we observe that the component-wise multiplication can be performed non-interactively in the online phase by taking the inputs in an encoded fashion. Elaborately, we take as input the augmented additive shares ($[\![\cdot]\!]$-shares) of T and W, which allows generating additive shares ($[\cdot]$-shares) of $T_i * W$, non-interactively, by invoking $\Pi^{\text{NI}}_{\text{Mult}}$ (Fig. 7). The rest of the computation continues operating on additive shares[8]. Finally, note that taking $[\![\cdot]\!]$-shares of T, W as input and naively executing the steps discussed earlier introduces the challenge that the pattern size and text size should be known in the preprocessing phase. We discuss in §D.2 how this can be circumvented. The formal protocol for wildcard pattern matching appears in Fig. 5.

7 Approximate Pattern Matching

Unlike wildcard pattern matching, the approximate variant allows mismatches to occur even without specifying wildcard characters. These mismatches can occur in any position. However, P is said to approximately match a substring in T, if the number of mismatches between P and the substring in T is within some public threshold τ.

Unlike the exact and wildcard pattern matching protocols, we are now interested in identifying the number of mismatches, h, between P and a substring in T, followed by determining if $h \leq \tau$. Our approach to determining h is as follows. Consider the scenario of matching P with T at position i, i.e., the substring $T_i = T[i : i+s_P-1]$, where $i \in \{1, \ldots, s_T - s_P + 1\}$. To determine h_i that counts the number of mismatches between P and T_i, we perform character-wise equality (using Π_{Eq} described in §4) and count the number of positions where these two strings mismatch. To facilitate counting the number of mismatches, the outputs of the character-wise equality checks have to be added. Since addition is an arithmetic operation, the output of Π_{Eq} is required to be generated as an arithmetic share rather than a Boolean share, unlike as described in §4. Such an equality protocol can be realised similar to Π_{Eq} with the difference that $\mathcal{F}_{\text{EqZ}}$ (within Π_{Eq}) is instantiated with a DPF-based equality protocol such that it generates arithmetic shares of the output instead of Boolean shares. This requires additionally communicating $\ell \log(\ell)$ bits in the preprocessing phase. Having generated the arithmetic shares of the output of equality, the number of mismatches h_i can now be computed locally by summing up these outputs of equality check protocol, followed by checking if $h_i \leq \tau$ by invoking $\mathcal{F}_{\text{comp}}$ (Fig. 10).

A straightforward realization of the above steps for approximate pattern matching results in requiring 2 online rounds for equality protocol and 1 online round for $\mathcal{F}_{\text{comp}}$ (assuming $\mathcal{F}_{\text{comp}}$ is instantiated via distributed comparison function (DCF) [16]). Thus, the number of online rounds required is 3. Moreover, the online communication is $O(s_T s_P \ell)$ bits due to the need for $O(s_T s_P)$ calls to equality protocol required to check the equality of each character in P with each character in T_i for $i \in \{1, \ldots, s_T - s_P + 1\}$. We reduce this round as well as communication complexity to 1 round and $O(s_T \ell)$ bits, respectively, as follows. Note that unlike in the case of exact and wildcard pattern matching where the equality protocol is invoked on $\lambda = 256$ bit input (size of the hash), in approximate

[8] Recall, as discussed in §5, that we do not rely on invoking the equality protocol on $[\![\cdot]\!]$-shared inputs because this would require an additional round of interaction to generate $[\![\cdot]\!]$-shares of h_{i0}, h_{i1}. This is unlike in the current scenario where the $[\cdot]$-shares of h_{i0}, h_{i1}, that are fed as input to Π_{Eq}, can be generated non-interactively.

pattern matching, it is invoked on $\ell = 8$ bit inputs. Hence, instead of relying on our 2 round equality protocol, we can perform the equality check by directly relying on DPF-based equality protocol. Observe that since the latter is invoked on small-sized input, its computation cost is reasonable, and it also results in having a reduced online round complexity of 1 in comparison to our equality protocol. In fact, we are able to completely eliminate the online cost due to the equality protocol by making the online phase of the DPF-based equality protocol non-interactive, as follows.

DPF-based equality protocol [16] takes as input $[\cdot]$-shares of an ℓ-bit input. In its online phase, it reconstructs a masked value corresponding to the input, which can abstracted out as generating $[\![\cdot]\!]$-shares of the input. This step requires communicating 2ℓ-bits in 1 round of interaction. This is followed by local computation that results in generating $[\cdot]$-shares of the output. Hence, to save on the interaction within the DPF-based equality, we take the inputs to approximate pattern matching in an encoded fashion. This encoding is essentially a $[\![\cdot]\!]$-sharing of the inputs instead of $[\cdot]$-sharing. Thus, encoding the inputs in this way allows us to realise the online phase of the DPF-based equality protocol non-interactively. Further, encoding the inputs also allows us to save on $O(\mathsf{s_T s_P}\ell)$ bits of communication which would have otherwise been required during the online phase for each of the $O(\mathsf{s_T s_P})$ invocations of equality (recall that each character in P is compared with each character in T_i for equality, for $i \in \{1, \ldots, \mathsf{s_T} - \mathsf{s_P} + 1\}$). Note that, as in the case of wildcard pattern matching, taking as input $[\![\cdot]\!]$-shares of T and P introduces challenges such as requiring the knowledge of pattern size and text size in the preprocessing. We discuss in §D.2 how this can be avoided. In this way, the approximate pattern matching protocol requires only 1 round of interaction and communication of $O(\mathsf{s_T}\ell)$ bits in the online phase (as required when instantiating $\mathcal{F}_{\mathsf{comp}}$ via DCFs). The formal protocol for approximate pattern matching appears in Fig. 6 where $\Pi_{\mathsf{EqZ^A}}$ denotes the DPF-based equality protocol with a non-interactive online phase. $\Pi_{\mathsf{EqZ^A}}$ takes $[\![\cdot]\!]$-shares of the input, say $\mathsf{x} \in \mathbb{Z}_{2^\ell}$ and generates $[\cdot]$-shares of z such that $\mathsf{z} = 1$ if $\mathsf{x} = 0$ and $\mathsf{z} = 0$ otherwise.

Protocol Π_{ApprxPM}

Inputs: $[\![\cdot]\!]$-shares of text T with $\mathsf{s_T}$ characters and pattern P with $\mathsf{s_P}$ characters, where each character in these two arrays is $[\![\cdot]\!]$-shared. A public threshold τ.

Output: $[\cdot]^{\mathbf{B}}$-shares of $(\mathsf{s_T} - \mathsf{s_P} + 1)$-sized array O, where $\mathsf{O}[i] = 1$ if P occurs at position i in T with a mismatch of at most τ characters, and 0 otherwise.

Protocol:

- For $i = 1$ to $\mathsf{s_T} - \mathsf{s_P} + 1$ do (in parallel)
  - For $j = 1$ to $\mathsf{s_P}$ do (in parallel)
    - $[\![\mathsf{X}_{ij}]\!] = [\![\mathsf{T}_i[j]]\!] - [\![\mathsf{P}_i[j]]\!]$
    - $\left[\mathsf{h}_{ij}\right] = \Pi_{\mathsf{EqZ^A}}\left([\![\mathsf{X}_{ij}]\!], \mathbb{Z}_{2^\ell}\right)$ (Fig. 13)
  - $[\mathsf{h}_i] = \sum_{j=1}^{\mathsf{s_P}} (1 - \left[\mathsf{h}_{ij}\right])$
  - $[\mathsf{O}[i]]^{\mathbf{B}} = \mathcal{F}_{\mathsf{comp}}([\mathsf{h}_i], \tau)$ (Fig. 10)

Figure 6: Approximate pattern matching

8 Benchmarks

We empirically evaluate the performance of our equality check protocol and the protocols for the considered variants of pattern matching while accounting for various parameters. We benchmark the performance over LAN on Ubuntu servers with AMD Ryzen Threadripper PRO 5965WX, utilizing four cores with 16GB RAM. The machines have a bandwidth of 1Gbps. We implement our protocols on top of MOTION2NX framework [17]. We consider a bandwidth of 1 Gbps and 0.05 ms of latency for LAN. Our code accounts for multi-threading wherever possible (4 threads). We note that our code[9] is developed for benchmarking, is not optimised for industry-grade use. We consider online run time and communication costs of the protocols as benchmark parameters for comparison. We benchmark the preprocessing cost and report it in §F. For completeness, we report the benchmarks over WAN in §F.

8.1 Performance of Equality

We compare the performance of our equality protocol, Π_{Eq}, with that of circuit-based equality protocols [12, 50] and DPF-based equality protocol [16] across varying input lengths and report the costs in Table 3. With respect to [12], Π_{Eq} outperforms in terms of run time and communication. As expected, the improvements over [12] increase with the increase in input length, where we see improvements of up to 4× and 1.9×, respectively, in terms of run time and communication for 32B=256 bit inputs. This improvement arises since [12] and [50] require a logarithmic number of rounds as opposed to the constant 2 rounds for Π_{Eq}. Moreover, the communication cost of [12] is approximately 4ℓ bits as opposed to $2\ell + 2\log(2\ell)$ bits for Π_{Eq}. Compared to [50], we witness an improvement of 1.9× and 1.28× in run time and communication for 32B=256 bit inputs. Observe that for $\ell = 1B, 2B$, [50] has better communication complexity and the same round complexity as ours. Despite this, our costs are comparable to that of [50]. Further, as ℓ increases, our protocol outperforms the protocol of [50]. This is because, as ℓ increases, the round complexity of [50] increases by a factor of $O(\log(\ell))$ whereas our protocol has a constant round complexity of 2 rounds. With respect to the protocol of [16], Π_{Eq} witnesses improvements of up to 9.7× in run time when considering 256-bit inputs (32B) despite having a slightly higher communication cost. This is attributed to the improved computation cost of Π_{Eq}, which, as discussed in §4, requires only $\log(2\ell)$ sequential calls to a PRG unlike the ℓ sequential calls required in the protocol of [16]. Thus, as expected, with increasing ℓ, the improvements in run time over [16] also increase from 1.2× to 9.7×. In this way, our protocol attains a fast response time compared to prior approaches.

The work of [51] also provides an equality protocol in the 2PC semi-honest setting, where they leverage an augmented additive sharing scheme ($[\![\cdot]\!]$-sharing) to attain a fast online phase. When drawing a comparison with the protocol of [51], directly comparing it with Π_{Eq} may not yield a fair comparison since Π_{Eq} is designed to be generic and operate with additive shares ($[\cdot]$-shares) unlike [51]. Instead, we showcase how Π_{Eq} can also leverage operating with augmented additive shares to attain an improved online communication complexity and compare the resulting protocol with that in [51]. While our modified equality protocol, $\Pi_{\mathsf{Eq^{AS}}}$, is described

[9] https://github.com/Bhavishrg/MOTION2NX/tree/PatternMatching

Input size (B)	Protocol	Run time(ms)	Comm. (B)
1	[12]	0.26	3.50
	[50]	0.14	2.50
	[16]	0.18	2.00
	Π_{Eq}	0.14	3.00
2	[12]	0.41	7.50
	[50]	0.17	5.00
	[16]	0.23	4.00
	Π_{Eq}	0.18	5.25
8	[12]	0.58	31.50
	[50]	0.31	21.00
	[16]	0.79	16.00
	Π_{Eq}	0.21	17.75
16	[12]	1.17	127.50
	[50]	0.54	85.00
	[16]	2.74	64.00
	Π_{Eq}	0.28	66.25

Table 3: Comparison of equality for varying input sizes.

in §C.2, its comparison with [51] appears in Table 4. Observe that $\Pi_{\mathsf{Eq}^{\mathsf{AS}}}$ improves in terms of run time and communication. This is because the round complexity of $\Pi_{\mathsf{Eq}^{\mathsf{AS}}}$ is just 2 instead of $\log_4(\ell)+1$ in [51]. Moreover, $\Pi_{\mathsf{Eq}^{\mathsf{AS}}}$ requires communicating only $2\log(2\ell)+2$ bits as opposed to approximately $5\ell/3$ bits. This exponential saving in rounds and communication is witnessed as an improvement of up to 3.6× and 11.2×, respectively, for 256-bit (32B) inputs. This improvement increases with increasing input size, as evident from Table 4.

Input size (B)	Protocol	Run time (ms)	Comm. (B)
1	[51]	0.21	1.75
	$\Pi_{\mathsf{Eq}^{\mathsf{AS}}}$	0.14	1.25
2	[51]	0.35	3.25
	$\Pi_{\mathsf{Eq}^{\mathsf{AS}}}$	0.17	1.50
8	[51]	0.53	13.50
	$\Pi_{\mathsf{Eq}^{\mathsf{AS}}}$	0.21	2.00
32	[51]	0.99	28.00
	$\Pi_{\mathsf{Eq}^{\mathsf{AS}}}$	0.27	2.50

Table 4: Comparison of equality when operating with augmented additive sharing scheme for varying input sizes.

8.2 Performance of Pattern Matching Protocols

We next analyse the performance of our pattern matching protocols in comparison to their naive solutions. Since the performance is dependent on the pattern and text sizes, we vary these parameters when analysing the same.

8.2.1 Exact pattern matching. Table 5 reports performance for exact pattern matching[10]. As evident, our solution outperforms the naive solution (Fig. 14) in terms of run time as well as communication, where we see improvements of up to 3 orders of magnitude in the run time and 2 orders of magnitude in terms of communication. This improvement stems from the design of Π_{ExactPM}, which has a constant round complexity as opposed to $O(\log(\mathsf{s_P}))$ in the naive solution. Moreover, Π_{ExactPM} is designed to reduce the number of invocations to Π_{Eq} where it requires only $O(\mathsf{s_T})$ calls to Π_{Eq} as opposed to $O(\mathsf{s_T s_P})$ calls in the naive solution. This design of Π_{ExactPM} allows it to witness tremendous improvement in run time and communication.

Further, observe that for a fixed $\mathsf{s_T}$, the run time of the naive solution increases linearly with increasing $\mathsf{s_P}$. This is expected because both the round and communication complexity of the naive solution have a dependency on $\mathsf{s_P}$ as well. This is where our improvement comes in, where we make this complexity independent of $\mathsf{s_P}$ for Π_{ExactPM} by reducing the number of calls to Π_{Eq} as well as introducing the hash-based optimisation. Elaborately, the number of calls to Π_{Eq} in Π_{ExactPM} is $\mathsf{s_T} - \mathsf{s_P} + 1$, and this number reduces as $\mathsf{s_P}$ increases for a fixed $\mathsf{s_T}$. This reduction leads to a decrease in run time as well as communication of Π_{ExactPM} as $\mathsf{s_P}$ increases.

Text size ($\mathsf{s_T}$)	Pattern size ($\mathsf{s_P}$)	Protocol	Run time (s)	Comm. (MB)
100B	10B	Naive	0.10	0.14
		Ours	0.01	0.05
1KB	10B	Naive	1.06	1.58
		Ours	0.07	0.51
	100B	Naive	9.65	14.40
		Ours	0.05	0.47
10KB	10B	Naive	10.59	15.98
		Ours	0.59	5.19
	100B	Naive	104.91	158.40
		Ours	0.55	5.15
	1KB	Naive	953.76	1440.02
		Ours	0.46	4.68

Table 5: Performance of exact pattern matching for varying text size ($\mathsf{s_T}$) and pattern size ($\mathsf{s_P}$) for $\ell = 8$.

8.2.2 Wildcard pattern matching. Table 6 reports performance for wildcard pattern matching. As evident, our solution outperforms the naive solution (Fig. 15) in terms of run time as well as communication, where we see improvements of up to 3 orders of magnitude in the run time as well as communication. Similar to exact pattern matching, this improvement stems from the design of Π_{WildPM}, which has a constant round complexity as opposed to $O(\log(\mathsf{s_P}))$ in the naive solution. Moreover, Π_{WildPM} requires only $O(\mathsf{s_T})$ calls to Π_{Eq} as opposed to $O(\mathsf{s_T s_P})$ calls in the naive solution. This design of Π_{WildPM} allows it to witness tremendous improvement in run time and communication.

Further, observe that for a fixed $\mathsf{s_T}$, the run time of the naive solution increases linearly with increasing $\mathsf{s_P}$. This is expected

[10]When invoking equality on inputs from $\mathbb{Z}_{2^k}$, we invoke $\mathcal{F}_{\mathsf{EqZ}}$ over $\mathbb{Z}_k$ instead of $\mathbb{Z}_{2^k}$ (for $k = 256 = 2^8$) since implementation for $\mathcal{F}_{\mathsf{EqZ}}$ supports operating on $\mathbb{Z}_{2^8}$. Note that this change introduces a failure probability of $\frac{1}{2^{256}}$, as discussed in §C.1, which is small for the pattern matching application.

Text size (s_T)	Pattern size (s_P)	Protocol	Run time (s)	Comm. (MB)
100B	10B	Naive	0.24	0.37
		Ours	0.01	0.05
1KB	10B	Naive	2.55	4.12
		Ours	0.07	0.51
	100B	Naive	12.12	37.45
		Ours	0.08	0.47
10KB	10B	Naive	23.23	41.56
		Ours	0.61	5.19
	100B	Naive	215.21	411.85
		Ours	0.71	5.15
	1KB	Naive	1969.80	3744.05
		Ours	2.49	4.68

Table 6: Performance of wildcard pattern matching for varying text size (s_T) and pattern size (s_P) for $\ell = 8$.

Text size (s_T)	Pattern size (s_P)	Protocol	Run time (s)	Comm. (KB)
100B	10B	Naive	0.44	20.188
		Ours	0.37	2.88
1KB	10B	Naive	4.15	221.788
		Ours	4.06	31.68
	100B	Naive	37.09	201.628
		Ours	36.86	28.80
10KB	10B	Naive	41.16	2237.788
		Ours	40.91	319.68
	100B	Naive	408.04	2217.628
		Ours	405.44	316.80
	1KB	Naive	4428.64	2016.028
		Ours	4074.68	288.00

Table 7: Performance of approximate pattern matching for varying text size (s_T) and pattern size (s_P) for $\ell = 8$.

because both the round and communication complexity of the naive solution have a dependency on s_P as well. On the contrary, this complexity is independent of s_P for Π_{WildPM}. Similar to the case of exact pattern matching, the number of calls to Π_{Eq} in Π_{WildPM} is $s_T - s_P + 1$, and this number reduces as s_P increases for a fixed s_T. This reduction leads to a decrease in the communication cost Π_{WildPM} as the s_P increases. However, the run time of Π_{WildPM} continues to increase with increasing s_P, albeit at a slower rate. The upward trend in run time stems from the fact that although the communication and rounds are independent of s_P, Π_{WildPM} involves performing $O(s_T s_P)$ local multiplications in the first step. The time taken for this computation overpowers the savings due to the reduced round and communication complexity, and results in showcasing an increasing run time with increasing s_P. In fact, due to this additional computation, the run time of Π_{WildPM} is slightly higher than that of Π_{ExactPM}.

8.2.3 Approximate pattern matching. Table 7 reports performance for approximate pattern matching. As expected, Π_{ApprxPM} outperforms the naive solution (Fig. 16) due to the saving in round as well as communication complexity, where we see improvements of up to 7× in communication. Despite the improved round and communication complexity, we note that the saving brought in run time brought in by Π_{ApprxPM} over the naive solution is roughly only 8%. This is because local computations dominate the run time and form the bottleneck. Additionally, when considering a fixed s_T, we note that the run time steadily increases despite the complexity being independent of s_P. This can again be attributed to the same reason that computation costs form the bottleneck. Finally, note that although the online communication and round complexity of approximate pattern matching are better than exact and wildcard pattern matching (Table 1), the former has a significantly high computation cost. This results in the run time of approximate pattern matching being higher than exact and wildcard variants.

8.2.4 Comparison with prior works. The code for most of the prior works on secure pattern matching via MPC is not available. Hence, we are unable to provide an explicit comparison with these. However, in what follows, we draw a direct comparison based on the numbers reported in the respective works. To give a fair comparison, we benchmark our protocols in a similar environment as considered in the respective prior works. Hence, we also consider $l = 2$ when drawing a comparison.

For exact pattern matching, we report a comparison with the protocol of [63] in Table 8 (first row). We observe improvements of around two orders of magnitude in the response time. Note that we were unable to establish a comprehensive analysis comparison with respect to varying text and pattern lengths due to unavailability of such data in [63]. With respect to [58], we note that they do not tabulate costs for varying text lengths. Hence, a comparative analysis could not be conducted.

Text size (s_T)	Pattern size (s_P)	Threshold (τ)	Protocol	Run time (s)
25B	5B	0	[63]	1.00
			Ours	0.04
50B	10B	1	[63]	2.44
			Ours	0.18
240B	48B	4	[63]	9.16
			Ours	3.76
1.96KB	402B	9	[63]	77.89
			Ours	12.33
5KB	1KB	14	[63]	262.49
			Ours	25.35

Table 8: Comparison of exact (row 1) and approximate pattern matching with [63] for varying text and pattern size for $\ell = 2$.

For approximate pattern matching, Table 8 reports a comparison with the work of [63] for varying text size, pattern size and threshold, as considered in [63]. In comparison to [63], we observe improvements of up to 10.3× in response time. Moreover, for a fixed pattern size and text size, we note that since the complexity of Π_{ApprxPM} is independent of the threshold, its complexity does not vary as the threshold varies. On the contrary, the complexity of [63] increases with an increase in the threshold. We are unable to report such a comparison with respect to the approximate pattern matching protocol of [58] for the same reasons as stated earlier.

Finally, with respect to wildcard pattern matching, in comparison to the protocol of [57], although our protocol has the same round complexity, its communication cost improves for larger values ℓ.

We believe this will result in our protocol having a better response time than that of [57]. However, since the implementation of [57] does not allow varying ℓ, we are unable to empirically corroborate our claims. Further, note that [57] operates in the client-server setting. Hence, the client is required to actively participate in all the computations, which is not the case in our protocol, thereby enabling lightweight clients to utilize our protocol. Finally, unlike in our protocols, [57] requires knowledge of pattern and text size during preprocessing, which may not be feasible in practice.

9 Conclusion

We design secure solutions for three variants of pattern matching—exact, wildcard and approximate. In the process, we design a novel protocol for equality check which has a constant online round complexity and linear online communication and outperforms all the prior equality protocols [12, 15, 16, 22, 41, 48]. We implement all our protocols for the different variants of pattern matching and showcase how they improve over their respective naive solutions and prior works.

Acknowledgments

Arpita Patra would like to acknowledge the support of the Center of Excellence for Cybersecurity (CySecK) at Indian Institute of Science, Government of Karnataka and Google Privacy Research Award. Bhavish Raj Gopal would like to Acknowledge the financial support provided by the Prime Minister's Research Fellowship (PMRF-ID, 0202952).

References

[1] Mark Abspoel, Daniel Escudero, and Nikolaj Volgushev. Secure training of decision trees with continuous attributes. *Cryptology ePrint Archive*, 2020.
[2] Nitish Andola, Sourabh Prakash, S Venkatesan, and Shekhar Verma. Improved secure server-designated public key encryption with keyword search. In *CICT 2017*. IEEE, 2017.
[3] Gilad Asharov, Shai Halevi, Yehuda Lindell, and Tal Rabin. Privacy-preserving search of similar patients in genomic data. *Cryptology ePrint Archive*, 2017.
[4] Gilad Asharov, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, Ariel Nof, Benny Pinkas, Katsumi Takahashi, and Junichi Tomida. Efficient secure three-party sorting with applications to data analysis and heavy hitters. In *CCS*, 2022.
[5] Mikhail J Atallah, Florian Kerschbaum, and Wenliang Du. Secure and private sequence comparisons. In *Proceedings of the 2003 ACM Workshop on Privacy in the Electronic Society*, 2003.
[6] Joonsang Baek, Reihaneh Safavi-Naini, and Willy Susilo. Public key encryption with keyword search revisited. In *ICCSA 2008*. Springer, 2008.
[7] Joshua Baron, Karim El Defrawy, Kirill Minkovich, Rafail Ostrovsky, and Eric Tressler. 5pm: Secure pattern matching. *Journal of computer security*, 21, 2013.
[8] Donald Beaver. Efficient multiparty protocols using circuit randomization. In *CRYPTO*, 1992.
[9] Battista Biggio, Giorgio Fumera, Ignazio Pillai, and Fabio Roli. A survey and experimental evaluation of image spam filtering techniques. *Pattern recognition letters*, 2011.
[10] Wang BingJian, Chen TzungHer, and Jeng FuhGwo. Security improvement against malicious server's attack for a dpeks scheme. *Int J Inf Educ Technol*, 2011.
[11] Allison Bishop, Lucas Kowalczyk, Tal Malkin, Valerio Pastro, Mariana Raykova, and Kevin Shi. A simple obfuscation scheme for pattern-matching with wildcards. In *CRYPTO*, 2018.
[12] Dan Bogdanov, Margus Niitsoo, Tomas Toft, and Jan Willemson. High-performance secure multi-party computation for data mining applications. *International Journal of Information Security*, 11, 2012.
[13] Dan Boneh, Giovanni Di Crescenzo, Rafail Ostrovsky, and Giuseppe Persiano. Public key encryption with keyword search. In *International conference on the theory and applications of cryptographic techniques*. Springer, 2004.
[14] Dan Boneh, Eyal Kushilevitz, Rafail Ostrovsky, and William E Skeith III. Public key encryption that allows pir queries. In *Annual International Cryptology Conference*. Springer, 2007.
[15] Elette Boyle, Nishanth Chandran, Niv Gilboa, Divya Gupta, Yuval Ishai, Nishant Kumar, and Mayank Rathee. Function secret sharing for mixed-mode and fixed-point secure computation. In *Eurocrypt*, 2021.
[16] Elette Boyle, Niv Gilboa, and Yuval Ishai. Secure computation with preprocessing via function secret sharing. In *TCC*, 2019.
[17] Lennart Braun, Rosario Cammarota, and Thomas Schneider. A generic hybrid 2PC framework with application to private inference of unmodified neural networks (extended abstract). In *PriML@NeurIPS*, 2021.
[18] Philipp Bucher and Amos Bairoch. A generalized profile syntax for biomolecular sequence motifs and its function in automatic sequence interpretation. In *Ismb*, 1994.
[19] Jin Wook Byun, Hyun Suk Rhee, Hyun-A Park, and Dong Hoon Lee. Off-line keyword guessing attacks on recent keyword search schemes over encrypted data. In *Workshop on secure data management*. Springer, 2006.
[20] Harsh Chaudhari, Ashish Choudhury, Arpita Patra, and Ajith Suresh. ASTRA: High Throughput 3PC over Rings with Application to Secure Prediction. In *ACM CCSW@CCS*, 2019.
[21] Ivan Damgård, Daniel Escudero, Tore Frederiksen, Marcel Keller, Peter Scholl, and Nikolaj Volgushev. New primitives for actively-secure mpc over rings with applications to private machine learning. In *IEEE S&P*, 2019.
[22] Ivan Damgård, Matthias Fitzi, Eike Kiltz, Jesper Buus Nielsen, and Tomas Toft. Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. In *TCC*, 2006.
[23] Sarang Dharmapurikar and John W Lockwood. Fast and scalable pattern matching for network intrusion detection systems. *IEEE JSAC*, 2006.
[24] Jack Doerner and Abhi Shelat. Scaling oram for secure computation. In *CCS*, 2017.
[25] Daniel Escudero, Satrajit Ghosh, Marcel Keller, Rahul Rachuri, and Peter Scholl. Improved primitives for mpc over mixed arithmetic-binary circuits. In *CRYPTO*, 2020.
[26] Laurent Falquet, Marco Pagni, Philipp Bucher, Nicolas Hulo, Christian JA Sigrist, Kay Hofmann, and Amos Bairoch. The prosite database, its status in 2002. *Nucleic acids research*, 2002.
[27] Keith B Frikken. Practical private dna string searching and matching through efficient oblivious automata evaluation. In *Data and Applications Security XXIII: 23rd Annual IFIP WG 11.3 Working Conference*. Springer, 2009.
[28] Rosario Gennaro, Carmit Hazay, and Jeffrey S Sorensen. Text search protocols with simulation based security. In *PKC*, 2010.
[29] Ashish Prosad Gope and Rabi Narayan Behera. A novel pattern matching algorithm in genome sequence analysis. *International Journal of Computer Science and Information Technologies*, 2014.
[30] Carmit Hazay and Yehuda Lindell. Efficient protocols for set intersection and pattern matching with security against malicious and covert adversaries. In *TCC*, 2008.
[31] Carmit Hazay and Tomas Toft. Computationally secure pattern matching in the presence of malicious adversaries. *JoC*, 2014.
[32] Wing-Kail Hon, Rahul Shah, and Jeffrey S Vitter. Ordered pattern matching: Towards full-text retrieval. 2006.
[33] Pranav Jangir, Nishat Koti, Varsha Bhat Kukkala, Arpita Patra, Bhavish Raj Gopal, and Somya Sangal. Vogue: Faster computation of private heavy hitters. *TDSC*, 2022.
[34] Jonathan Katz and Lior Malka. Secure text processing with applications to private dna matching. In *CCS*, 2010.
[35] Florian Kerschbaum. Practical private regular expression matching. In *IFIP International Information Security Conference*. Springer, 2006.
[36] Etienne Kneuss, Manos Koukoutos, and Viktor Kuncak. Deductive program repair. In *International Conference on Computer Aided Verification*, 2015.
[37] Vladimir Kolesnikov, Mike Rosulek, and Ni Trieu. Swim: Secure wildcard pattern matching from ot extension. In *FC*, 2018.
[38] Nishat Koti, Mahak Pancholi, Arpita Patra, and Ajith Suresh. SWIFT: Super-fast and robust privacy-preserving machine learning. In *USENIX Security*, 2021.
[39] Peeter Laud and Jan Willemson. Universally composable privacy preserving finite automata execution with low online and offline complexity. *Cryptology ePrint Archive*, 2013.
[40] Dongmei Li, Xiaolei Dong, and Zhenfu Cao. Secure and privacy-preserving pattern matching in outsourced computing. *SCN*, 2016.
[41] Helger Lipmaa and Tomas Toft. Secure equality and greater-than tests with sublinear online complexity. In *ICALP*, 2013.
[42] Ning Luo, Chenkai Weng, Jaspal Singh, Gefei Tan, Ruzica Piskac, and Mariana Raykova. Privacy-preserving regular expression matching using nondeterministic finite automata. *Cryptology ePrint Archive*, 2023.
[43] Ning Luo, Chenkai Weng, Jaspal Singh, Gefei Tan, Mariana Raykova, and Ruzica Piskac. Privacy-preserving regular expression matching using tnfa. In *European Symposium on Research in Computer Security*, pages 225–246. Springer, 2024.
[44] Victor Wing-Kit Mak, Kuo Chu Lee, and Ophir Frieder. Exploiting parallelism in pattern matching: An information retrieval application. *ACM TOIS*, 1991.
[45] Eleftheria Makri, Dragos Rotaru, Frederik Vercauteren, and Sameer Wagh. Rabbit: Efficient comparison for secure multi-party computation. In *FC*, 2021.

[46] Payman Mohassel, Salman Niksefat, Saeed Sadeghian, and Babak Sadeghiyan. An efficient protocol for oblivious DFA evaluation and applications. In *CT-RSA*, 2012.
[47] Payman Mohassel, Salman Niksefat, Saeed Sadeghian, and Babak Sadeghiyan. An efficient protocol for oblivious dfa evaluation and applications. In *CT-RSA 2012*. Springer, 2012.
[48] Takashi Nishide and Kazuo Ohta. Multiparty computation for interval, equality, and comparison without bit-decomposition protocol. In *PKC*, 2007.
[49] Koji Nuida. Privacy-preserving datase search protocol for chemical compounds with additive-homomorphic encryption. In *Proc. Computer Security Symposium 2012, Oct.*, 2012.
[50] Satsuya Ohata and Koji Nuida. Communication-efficient (client-aided) secure two-party protocols and its application. In *International Conference on Financial Cryptography and Data Security*, 2020.
[51] Arpita Patra, Thomas Schneider, Ajith Suresh, and Hossein Yalame. ABY2.0: Improved mixed-protocol secure two-party computation. In *USENIX Security*, 2021.
[52] Hong Qin, Hao Wang, Xiaochao Wei, Likun Xue, and Lei Wu. Privacy-preserving wildcards pattern matching protocol for iot applications. *IEEE Access*, 7, 2019.
[53] Takanori Saito and Toru Nakanishi. Designated-senders public-key searchable encryption secure against keyword guessing attacks. In *CANDAR 2017*. IEEE, 2017.
[54] Mohamad Hasan Samadani and Mehdi Berenjkoub. Secure outsourced pattern matching based on bit-parallelism. *The Modares Journal of Electrical Engineering*, 16, 2016.
[55] Thomas Schneider and Oleksandr Tkachenko. Episode: efficient privacy-preserving similar sequence queries on outsourced genomic databases. In *Proceedings of the 2019 ACM Asia conference on computer and communications security*, 2019.
[56] Juan Ramón Troncoso-Pastoriza, Stefan Katzenbeisser, and Mehmet Celik. Privacy preserving error resilient dna searching through oblivious automata. In *CCS*, 2007.
[57] Xiaochao Wei, Lin Xu, Minghao Zhao, and Hao Wang. Secure extended wildcard pattern matching protocol from cut-and-choose oblivious transfer. *Information Sciences*, 529, 2020.
[58] Xiaochao Wei, Minghao Zhao, and Qiuliang Xu. Efficient and secure outsourced approximate pattern matching protocol. *Soft Computing*, 22, 2018.
[59] Peng Yang, Zoe L Jiang, Shiqi Gao, Jiehang Zhuang, Hongxiao Wang, Junbin Fang, Siuming Yiu, and Yulin Wu. Fssnn: Communication-efficient secure neural network training via function secret sharing. *Cryptology ePrint Archive*, 2023.
[60] Masaya Yasuda, Takeshi Shimoyama, Jun Kogure, Kazuhiro Yokoyama, and Takeshi Koshiba. Privacy-preserving wildcards pattern matching using symmetric somewhat homomorphic encryption. In *ACISP*, 2014.
[61] Wei-Chuen Yau, Raphael C-W Phan, Swee-Huay Heng, and Bok-Min Goi. Keyword guessing attacks on secure searchable public key encryption schemes with a designated tester. *International Journal of Computer Mathematics*, 2013.
[62] Fang Yu, Randy H Katz, and Tirunellai V Lakshman. Gigabit rate packet pattern-matching using tcam. In *Proceedings of the 12th IEEE International Conference on Network Protocols, 2004. ICNP 2004.*, pages 174–183. IEEE, 2004.
[63] Maryam Zarezadeh, Hamid Mala, and Behrouz Tork Ladani. Efficient secure pattern matching with malicious adversaries. *TDSC*, 2020.
[64] L. Zhou, Z. Wang, H. Cui, Q. Song, and Y. Yu. Bicoptor: Two-round secure three-party non-linear computation without preprocessing for privacy-preserving machine learning. In *S&P*, 2023.

A Related Works

Here, we discuss other orthogonal lines of works related to pattern matching.

String similarity: An orthogonal line of work considers the question of determining the similarity between a given string (pattern) and each string in a database of strings [3, 5, 37, 55]. Unlike our setting, where we consider the pattern and substring of the text to be of equal length, the focus in these works is on securely computing edit distance metrics to compute the similarity of two strings of different lengths. Finally, these works aim to propose new metrics for computing edit distance or focus on improving the efficiency of existing metrics.

Regular expression matching: Another set of works [27, 35, 39, 42, 43, 47] consider a more generic verison of pattern matching known as regular expressing matching. Here, the pattern is represented in the form of a regular expression and aim is to identify a match in text using this structured pattern. We note that these protocols are less efficient in comparison to customized exact, wildcard and approximate pattern-matching protocols. They often incur a round complexity linear in the pattern size and communication complexity in the order of alphabet size making them inefficient for the scenarios considered in this work.

Public-key encryption with keyword search: Another orthogonal line of work [2, 6, 10, 13, 14, 19, 49, 53, 61] considers the question of designing encryption schemes that allows performing search on the ciphertexts. These techniques are designed for applications where a client stores encrypted data on external server, subsequently performing search on this data. Here it is often assumed that text and pattern come from the same client, and pattern is required to be known in advance as encryption depends on it. It is non-trivial to extend them for general pattern-matching.

B Preliminaries

Non-interactive multiplications: The formal protocol for $\Pi^{\mathsf{NI}}_{\mathsf{Mult}}$ appears in Fig. 7

Protocol $\Pi^{\mathsf{NI}}_{\mathsf{Mult}}$

Inputs: $[\![\cdot]\!]$-shares of $\mathsf{x}, \mathsf{y} \in \mathbb{Z}_{2^\ell}$.
Outputs: $[\cdot]$-shares of $\mathsf{z} = \mathsf{xy}$.
Protocol:
Preprocessing:

- Party P_j for $j \in \{0,1\}$ samples random $[\delta_{\mathsf{z}}]_j$
- Parties execute $\Pi_{\mathsf{setupMULT}}([\delta_{\mathsf{x}}], [\delta_{\mathsf{y}}])$ to generate $[\delta_{\mathsf{xy}}]$.

Online:

Local Computation

- Party P_j for $j \in \{0,1\}$ locally computes $[\mathsf{z}] = i \cdot \mathsf{m_x m_y} - \mathsf{m_x}[\delta_{\mathsf{y}}]_i - \mathsf{m_y}[\delta_{\mathsf{x}}]_i + [\delta_{\mathsf{xy}}]_i$.

Figure 7: Non-interactive multiplication protocol

Bit to arithmetic: Given $[\cdot]^{\mathbf{B}}$ −shares of values x, the bit to arithmetic protocol generates $[\cdot]$-shares of x. Let x^{a} denote the value of x when viewed over the ℓ−bit ring $\mathbb{Z}_{2^\ell}$. Observe that, $\mathsf{x} = [\mathsf{x}]_0 \oplus [\mathsf{x}]_1$, thus x^{a} can be expressed as $\mathsf{x}^{\mathsf{a}} = [\mathsf{x}]^{\mathsf{a}}_0 + [\mathsf{x}]^{\mathsf{a}}_1 - 2[\mathsf{x}]^{\mathsf{a}}_0 \cdot [\mathsf{x}]^{\mathsf{a}}_1$. To generate $[\cdot]$-shares of x^{a}, parties generate $[\cdot]$ shares of y such that $\mathsf{y} = [\mathsf{x}]^{\mathsf{a}}_0 \cdot [\mathsf{x}]^{\mathsf{a}}_1$ using the multiplication protocol. Following this, party P_i for $i \in \{0,1\}$ can compute $[\mathsf{x}^{\mathsf{a}}]_i = [\mathsf{x}]^{\mathsf{a}}_i - 2[\mathsf{y}]_i$. We denote the protocol as Π_{Bit2A} and the formal details appear in Fig. 8.

Protocol Π_{Bit2A}

Inputs: $[\cdot]^{\mathbf{B}}$-shares of $x \in \mathbb{Z}_{2^\ell}$.
Outputs: $[\cdot]$-shares of x.
Protocol:
- Parties generate $[y] = [x]_0^a \cdot [x]_1^a$ using correlated OTs.
- Party P_i for $i \in \{0,1\}$ locally computes $[x^a]_i = [x]_i^a - 2[y]_i$

Figure 8: Bit to arithmetic protocol

DPF based equality: The formal protocol for DPF based equality of [16] appears in Fig. 9. Here, $\mathcal{F}_{\text{DistKeyGen}}$ denotes the ideal functionality that realises the DPF $Gen(\cdot)$ algorithm. In our protocols, we instantiate $\mathcal{F}_{\text{DistKeyGen}}$ using the protocol of [24].

Protocol Π_{dpfeq}

Inputs: $[\cdot]^{\mathbf{B}}$-shares of $x, y \in \mathbb{Z}_{2^\ell}$.
Outputs: $[\cdot]$-shares of y such that $z = 1$ if $x = y$ and 0 otherwise.
Protocol:
Preprocessing:
- Parties P_0 and P_1 invoke $\mathcal{F}_{\text{DistKeyGen}}$ on random $[\![\cdot]\!]$-shared value r to obtain evaluation key k_0 and k_1 respectively.

Online:
- Parties compute $[x'] = [x] - [y] + [r]$.
- Party P_i sends $[x']_i$ to P_{i-1} to reconstruct x'.
- Party P_i sets $[\![z]\!]_i = Eval(i, k_i, x')$.
- Parties output $[z]$.

Figure 9: DPF based equality protocol of [16]

Comparison: The ideal functionality for comparison appears in Fig. 10. While $\mathcal{F}_{\text{comp}}$ can be instantiated using circuit-based comparison of [51], this incurs a round complexity of $\log_4 \ell$. Instead, we instantiate $\mathcal{F}_{\text{comp}}$ with via distributed comparison function (DCF) which has a constant round complexity of 1 round.

Functionality $\mathcal{F}_{\text{comp}}$

$\mathcal{F}_{\text{comp}}$ interacts with parties in $\mathcal{P}$ and $\mathcal{S}$. It receives as input $[\cdot]$-shares of $x, y \in \mathbb{Z}_{2^k}$ from the parties and $\mathcal{S}$, and proceeds as follows.
- Reconstruct x, y using the received $[\cdot]$-shares.
- If $x \le y$, set $z = 1$ else set $z = 0$, and generate $[\cdot]^{\mathbf{B}}$-shares of z.
- Send $(\text{Output}, [z]_i^{\mathbf{B}})$ to P_i for $i \in \{0,1\}$.

Figure 10: Ideal functionality for secure comparison

C Equality

C.1 Error Analysis

Recall that in the equality protocol to check if $x \in \mathbb{Z}_{2^\ell}$ is 0, we rely on calculating the Hamming distance c between a randomly chosen value $r \in \mathbb{Z}_{2^\ell}$ and the masked value $a = x + r$. The protocol then checks if c equals zero. Since a and r can differ in at most ℓ bit positions, it follows that $0 \le c \le \ell$. Observe that when c is represented in the ring $\mathbb{Z}_\ell$, if $c = \ell$ (in $\mathbb{Z}$), then $c = \ell \bmod \ell = 0$ (in $\mathbb{Z}_\ell$). Consequently, the protocol may yield an incorrect result when $c = \ell$ since it gets mapped to 0 in $\mathbb{Z}_\ell$. However, we show that the probability that this erroneous event occurs is negligible in ℓ, i.e., the probability of $c = \ell$ is at most $\frac{1}{2^\ell}$. Hence for cases where ℓ is large, we can work with the ring $\mathbb{Z}_\ell$ instead of $\mathbb{Z}_{2^\ell}$.

This probability can be explained by considering that $c = d_H(x + r, r) = \ell$ implies that $x + r = \bar{r}$, where $\bar{r}$ represents the bitwise complement of r. Now, we claim that for any given $x \in \mathbb{Z}_{2^\ell}$, there exists at most one $r \in \mathbb{Z}_{2^\ell}$ satisfying $x + r = \bar{r}$. This can be reasoned as follows: If $x + r = \bar{r}$, then $x = \bar{r} - r$. We now consider two cases.

Case 1: x *is even.* Observe that if x is even, no such r can exist to satisfy this equation. This is because if r is odd, then $\bar{r}$ is even, and vice versa. Consequently, $\bar{r} - r$ will always be odd. Hence, the probability that the hamming distance $c = \ell$ is 0 if x is even.

Case 2: x *is odd.* In this case, there exists precisely one r for which the equation holds. Let's assume for contrary that two such values, r and r', exist such that $x = \bar{r} - r = \bar{r'} - r'$. We can consider two cases: (i) If $r > r'$, this implies that $\bar{r} < \bar{r'}$ (this is because, at the first position from MSB where r and r' differ, r will have 1, and r' will have 0 since $r > r'$. When we take the complement of both, $\bar{r}$ will have 0, and $\bar{r'}$ will have a 1 at this position (while all the previous positions from MSB will be the same) which implies $\bar{r} < \bar{r'}$). This further implies that $\bar{r} - r < \bar{r'} - r'$. This directly contradicts our assumption that $x = \bar{r} - r = \bar{r'} - r'$. (ii) Conversely, if $r < r'$, this implies that $\bar{r} > \bar{r'}$, which leads to $\bar{r} - r > \bar{r'} - r'$. Again, this contradicts our assumption that $x = \bar{r} - r = \bar{r'} - r'$. Thus, there exists a unique r such that, $x = \bar{r} - r$ when x is odd and the probability of randomly sampling such an r is $\frac{1}{2^\ell}$.

C.2 Equality With Augmented Secret Sharing

We rely on the following primitives from [51] for designing our equality protocol over augmented secret sharing, (i) Bit to arithmetic conversion ($\Pi_{\text{bit2A}^{\text{AS}}}$): Given $[\![\cdot]\!]^{\mathbf{B}}$-shares of a bit $x \in \mathbb{Z}_2$, protocol $\Pi_{\text{bit2A}^{\text{AS}}}$ allows to generate its $[\![\cdot]\!]$-shares. The protocol requires one round of interaction in the online phase. (ii) Resharing (Π_{ReSH}): Given $[\cdot]$-shares of a value $x \in \mathbb{Z}_{2^\ell}$, protocol Π_{ReSH} allows to generate $[\![\cdot]\!]$-shares of x. The protocol requires one round of interaction in the online phase. (iii) Conversion from $[\![\cdot]\!]$-shares to $[\cdot]$-shares, non-interactively: Given $[\![\cdot]\!]$-shares of a value $x \in \mathbb{Z}_{2^\ell}$, its $[\cdot]$-shares can be obtained non-interactively as—P_0 sets $[x]_0 = m_x - [\delta_x]_0$ while P_1 sets $[x]_1 = -[\delta_x]_1$.

Hamming distance computation. The protocol $\Pi_{\text{Ham}^{\text{AS}}}$ takes as input augmented shares, $[\![x]\!]$ for $x \in \mathbb{Z}_{2^\ell}$ and outputs augmented shares, $[\![c]\!]$ (over $\mathbb{Z}_{2^\ell}$) where $c = d_H(x + r, r)$ for some random mask $r \in \mathbb{Z}_{2^\ell}$. The protocol follows along the lines of protocol Π_{Ham}. However, the online phase of the protocol can be made non-interactive as follows. Recall that for a value $x \in \mathbb{Z}_{2^\ell}$ that is $[\![\cdot]\!]$-shared, there exists a uniformly random input-independent mask $\delta_x \in \mathbb{Z}_{2^\ell}$ that is $[\cdot]$-shared that (in the preprocessing phase), and there exists a masked value $m_x = x + \delta_x$ such that m_x is known to both the parties in $\mathcal{P}$. Given this, to reconstruct $a = x + r$ in the protocol, parties first reconstruct $a' = \delta_x + r$. Observe that can happen in the preprocessing phase as shares of both δ_x and r are independent of the input x and available to both parties in the preprocessing phase. Following this, the parties can reconstruct $a = a' + m_x$ locally as both parties know m_x. The rest of the protocol follows along the lines of Π_{Ham}. The formal protocol for $\Pi_{\text{Ham}^{\text{AS}}}$ protocol appears in Fig. 11. The protocol requires communication

of $\ell(\kappa+\ell)+2\ell$ bits in the preprocessing phase while it is non-interactive in the online phase.

Equality. The protocol $\Pi_{\mathsf{Eq}^{\mathsf{AS}}}$ takes as input two $[\![\cdot]\!]$-shared values $\mathsf{x},\mathsf{y}\in\mathbb{Z}_{2^\ell}$, and outputs $[\![\cdot]\!]^{\mathbf{B}}$-shares of bit $\mathsf{z}=1\{\mathsf{x}=\mathsf{y}\}$. The protocol follows along the lines of Π_{Eq} except that it works on augmented additive sharing. It begins by locally computing $[\![\mathsf{x}']\!]=[\![\mathsf{x}]\!]-[\![\mathsf{y}]\!]$. This is followed by invoking $\Pi_{\mathsf{Ham}^{\mathsf{AS}}}$ on $[\![\mathsf{x}']\!]$ to compute the $[\![\mathsf{c}]\!]$ such that $\mathsf{c}=\mathsf{d}_{\mathsf{H}}(\mathsf{x}'+\mathsf{r},\mathsf{r})$ for a random $\mathsf{r}\in\mathbb{Z}_{2^\ell}$. $[\![\mathsf{c}]\!]$ is locally converted to $[\mathsf{c}]$, followed by non-interactively generating $[\mathsf{c}]^{2^\ell}$ via the ring change transformation. Following this, parties invoke $\mathcal{F}_{\mathsf{EqZ}}$ on $[\mathsf{c}]^{2^\ell}$ to generate $[\mathsf{z}]^{\mathbf{B}}$. Finally, Π_{ReSH} is invoked on $[\mathsf{z}]^{\mathbf{B}}$ to generate $[\![\mathsf{z}]\!]^{\mathbf{B}}$. The formal protocol for Π_{Eq} appears in Fig. 12. The protocol $\Pi_{\mathsf{Ham}^{\mathsf{AS}}}$ is non-interactive in the online phase, this eliminates the linear communication overhead in the online phase. The formal protocol for $\Pi_{\mathsf{Eq}^{\mathsf{AS}}}$ protocol appears in Fig. 12. The protocol requires communication of $\ell(\kappa+\ell)+\log(2\ell)\,(3\kappa+2)+2\ell$ bits in the preprocessing phase, while it requires 2 rounds and $2(\log(2\ell)+1)$ bits of communication in the online phase. The costs follow from the costs of $\Pi_{\mathsf{Ham}^{\mathsf{AS}}}$, Π_{EqZ} and Π_{ReSH}.

Protocol $\Pi_{\mathsf{Ham}^{\mathsf{AS}}}$

Inputs: $[\![\cdot]\!]$-shares of a value $\mathsf{x}\in\mathbb{Z}_{2^\ell}$.
Outputs: $[\![\cdot]\!]$-shares of c over $\mathbb{Z}_{2^\ell}$ where $\mathsf{c}=\mathsf{d}_{\mathsf{H}}(\mathsf{x}+\mathsf{r},\mathsf{r})$ for a random $\mathsf{r}\in\mathbb{Z}_{2^\ell}$.
Protocol:
Preprocessing:
- Parties sample $[\![\cdot]\!]^{\mathbf{B}}$-shares of $\mathsf{r}_i\in\mathbb{Z}_2$ for $i\in\{0,\dots,\ell-1\}$
- Parties compute $[\![\mathsf{r}_i]\!]=\Pi_{\mathsf{bit2A}^{\mathsf{AS}}}\left([\![\mathsf{r}_i]\!]^{\mathbf{B}}\right)$ for $i\in\{0,\dots,\ell-1\}$
- Parties compute $[\![\mathsf{r}]\!]=\sum_{i=0}^{\ell-1}2^i\cdot[\![\mathsf{r}_i]\!]$ and generate $[\mathsf{r}]$ from $[\![\mathsf{r}]\!]$ (see §3)
- Parties compute $[\mathsf{a}']=[\delta_{\mathsf{x}}]+[\mathsf{r}]$ and reconstruct a'

Online:
- Parties set $\mathsf{a}=\mathsf{m}_{\mathsf{x}}+\mathsf{a}'$ and compute $[\![\mathsf{c}]\!]=\sum_{i=0}^{\ell-1}\left(\mathsf{a}_i+[\![\mathsf{r}_i]\!]-2\mathsf{a}_i\cdot[\![\mathsf{r}_i]\!]\right)$

Figure 11: Hamming distance computation with a random value

Protocol $\Pi_{\mathsf{Eq}^{\mathsf{AS}}}$

Inputs: $[\![\cdot]\!]$-shares of $\mathsf{x},\mathsf{y}\in\mathbb{Z}_{2^\ell}$.
Outputs: $[\![\cdot]\!]^{\mathbf{B}}$-shares of a bit z such that $\mathsf{z}=1\{\mathsf{x}=\mathsf{y}\}$.
Protocol:
- Parties compute $[\![\mathsf{x}']\!]=[\![\mathsf{x}]\!]-[\![\mathsf{y}]\!]$
- Parties compute $[\![\mathsf{c}]\!]=\Pi_{\mathsf{Ham}^{\mathsf{AS}}}\left([\![\mathsf{x}']\!]\right)$ (Fig. 11).
- Parties generate $[\mathsf{c}]$ from $[\![\mathsf{c}]\!]$
- Party P_i for $i\in\{0,1\}$ locally sets $[\mathsf{c}]_i^{2^\ell}=[\mathsf{c}]_i \bmod 2^\ell$
- Parties invoke $\mathcal{F}_{\mathsf{EqZ}}$ (Fig. 3) on $[\mathsf{c}]^{2^\ell}$ to generate $[\mathsf{z}]^{\mathbf{B}}$.
- Parties generate $[\![\mathsf{z}]\!]^{\mathbf{B}}=\Pi_{\mathsf{ReSH}}([\mathsf{z}])$.

Figure 12: Equality protocol

C.3 Non-interactive Protocol for Equality With Zero

The non-interactive protocol for equality with zero over augmented additive shares is given in Fig. 13. The protocol takes as input $[\![\cdot]\!]$-shares of a value x and outputs $[\cdot]$-shares of the z such that $\mathsf{z}=1\{\mathsf{x}=0\}$. Our protocol relies on the generation and evaluation of DPF as described in [16]. The preprocessing phase of the protocol involves distributively generating the DPF keys for the point δ_{x}. We abstract this as a functionality $\mathcal{F}_{\mathsf{DistKeyGen}}$ which takes as input $[\cdot]$-shares of a value r and outputs keys k_0,k_1 such that P_i receives k_i. We instantiate it with the distributed key generation protocol of [24]. The online phase involves the evaluation of the DPF function [16] on the shared keys, denoted as $Eval(\cdot)$. This function takes as input the party index i, the corresponding key k_i, and the masked input m_x and outputs party P_i's share of the output.

Protocol $\Pi_{\mathsf{EqZ}^{\mathsf{A}}}$

Inputs: $[\![\cdot]\!]$-shares of $\mathsf{x}\in\mathbb{Z}_{2^\ell}$.
Outputs: $[\cdot]$-shares of z over $\mathbb{Z}_{2^\ell}$ such that $\mathsf{z}=1\{\mathsf{x}=0\}$.
Protocol:
Preprocessing:
- Parties invoke $\mathcal{F}_{\mathsf{DistKeyGen}}([\delta_{\mathsf{x}}])$ to distributively generate keys such that P_i receives k_i.

Online:
Local Computation
- Party P_i computes $[\mathsf{z}]_i=Eval(i,\mathsf{m}_{\mathsf{x}},k_i)$.

Figure 13: Equality with zero over augmented secret sharing

D Pattern Matching

D.1 Naive Variants of Pattern Matching

The formal protocol for the naive variants of exact, wildcard and approximate PM appears in Fig. 14, Fig. 15 and Fig. 16 respectively

Protocol $\Pi_{\mathsf{ExactPM\text{-}Naive}}$

Inputs: $[\cdot]$-shares of T with s_{T} characters and P with s_{P} characters, where each character in these two arrays is $[\cdot]$-shared.
Outputs: $[\cdot]^{\mathbf{B}}$-shares of $(\mathsf{s}_{\mathsf{T}}-\mathsf{s}_{\mathsf{P}}+1)$-sized array O, where $\mathsf{O}[i]=1$ if P occurs at position i in T, and 0 otherwise.
Protocol:
- For $i=1$ to $\mathsf{s}_{\mathsf{T}}-\mathsf{s}_{\mathsf{P}}+1$ do (in parallel)
  - For $j=1$ to s_{P} do (in parallel)
    - $\left[\mathsf{O}'_i[j]\right]^{\mathbf{B}}=\Pi_{\mathsf{Eq}}([\mathsf{P}[j]],[\mathsf{T}[i+j-1]])$ (Fig. 2)
- For $i=1$ to $\mathsf{s}_{\mathsf{T}}-\mathsf{s}_{\mathsf{P}}+1$ do (in parallel)
  - $[\mathsf{O}[i]]^{\mathbf{B}}=\wedge_{j=1}^{\mathsf{s}_{\mathsf{P}}}\left[\mathsf{O}'_i[j]\right]^{\mathbf{B}}$

Figure 14: Naive exact pattern matching

Protocol $\Pi_{\mathsf{WildPM\text{-}Naive}}$

Inputs: $[\cdot]$-shares of text T with s_{T} characters and pattern P with s_{P} characters, where each character in these two arrays is $[\cdot]$-shared. Additionally, $[\cdot]$-shares of array W of length s_{P} such that $\mathsf{W}[i]=0$ if $\mathsf{P}[i]$ is a wildcard character, and 1 otherwise.

Outputs: $[\cdot]^{\mathbf{B}}$-shares of $(s_T - s_P + 1)$-sized array O, where $O[i] = 1$ if P, with wildcard characters, occurs at position i in T, and 0 otherwise.

Protocol:

- For $i = 1$ to $s_T - s_P + 1$
  - $[T_i] = [T[i : i + s_P - 1]]$
  - Component-wise multiply $[T_i]$ and $[W]$ via Π_{mult} to generate $\left[T'_i\right]$
  - For $j = 1$ to s_P do (in parallel)
    - $\left[O'_i[j]\right]^{\mathbf{B}} = \Pi_{\mathsf{Eq}}([P[j]], [T_i[j]])$ (Fig. 2)
- For $i = 1$ to $s_T - s_P + 1$ do (in parallel)
  - $[O[i]]^{\mathbf{B}} = \wedge_{j=1}^{s_P} \left[O'_i[j]\right]^{\mathbf{B}}$

Figure 15: Naive wildcard pattern matching

Protocol $\Pi_{\mathsf{ApprxPM\text{-}Naive}}$

Inputs: $[\cdot]$-shares of text T with s_T characters and pattern P with s_P characters, where each character in the arrays are $[\cdot]$-shared. A public threshold τ.

Output: $[\cdot]^{\mathbf{B}}$-shares of $(s_T - s_P + 1)$-sized array O, where $O[i] = 1$ if P occurs at position i in T with a mismatch of at most τ characters, and 0 otherwise.

Protocol:

- For $i = 1$ to $s_T - s_P + 1$ do (in parallel)
  - For $j = 1$ to s_P do (in parallel)
    - $\left[h_{ij}\right] = \Pi_{\mathsf{Eq^A}}([T_i[j]], [P_i[j]])$
  - $[h_i] = \sum_{j=1}^{s_P}(1 - \left[h_{ij}\right])$
  - $[O[i]]^{\mathbf{B}} = \mathcal{F}_{\mathsf{comp}}([h_i], \tau)$ (Fig. 10)

Figure 16: Naive approximate pattern matching

D.2 Input Sharing for Pattern Matching

Recall that our protocols work in the secure outsourced setting. This entails the clients (pattern and text owner) secret sharing the inputs to the servers that carry out the MPC protocol for pattern matching. They obtain the output in secret shares and reconstruct it towards the intended recipient. Recall that for exact pattern matching protocol the clients have to generate $[\cdot]$-shares of the input towards the server. The generation of $[\cdot]$-shares towards the server for an $\ell-$bit input x held by a client can be achieved as follows. The client locally generates $[\cdot]$ −shares of x and sends $[x]_i$ to server P_i for $i \in \{0, 1\}$. This requires 1 round and 2ℓ bits of communication from clients to the servers. In the case of wildcard pattern matching and approximate pattern matching we use special input encoding where some of the input are taken as augmented secret shares ($[\![\cdot]\!]$-shares). Recall that in $[\![\cdot]\!]$-sharing of a value x, there exists a random mask δ_x independent of x that is $[\cdot]$ shared between the parties. The generation of $[\![\cdot]\!]$-shares towards the server for an $\ell-$bit input x held by a client can be achieved as follows. First, the client receives the randomness δ_x used by the servers in the preprocessing phase following which the client generates and sends $m_x = x + \delta_x$ to the servers. This requires 2 rounds and 4ℓ bits of communication between the client and the servers. This can be further optimised by establishing a common PRF key between each of the servers and the client in the setup phase. In this case, the server P_i, for $i \in \{0, 1\}$, and the client use the shared key to sample $[\delta_x]_i$. This allows the client to non-interactively obtain δ_x and thus reduce the cost of input sharing to 1 round and 2ℓ bits of communication.

In the case of wildcard pattern matching and approximate pattern matching, naively using the encoding requires the servers to know the size of the pattern and the text in the preprocessing phase. We take the example of wildcard pattern matching to elucidate this. The inputs to wildcard pattern matching consist of $[P]$ where every character in the pattern is $[\cdot]$-shared, $[\![T]\!]$ where every character in the text is $[\![\cdot]\!]$-shared and $[\![W]\!]$ where each element in the wildcard array W is $[\![\cdot]\!]$-shared. Thus the parties generate $[\delta_T[j]]$ for $j \in \{1, ..., s_T\}$ and $[\delta_W[k]]$ for $k \in \{1, ..., s_P\}$ (using the common PRF keys established with the text and pattern owner). To check whether a match occurs at position i, the first step entails multiplying the j^{th} character in the text substring T_i with the j^{th} element in the wildcard array W for $j \in \{1, \ldots, s_P + 1\}$. This is computed using $\Pi^{\mathsf{NI}}_{\mathsf{Mult}}$ (Fig. 7) on inputs $[\![W[j]]\!]$ and $[\![T_i[j]]\!]$. The preprocessing phase for the same involves generating $[\cdot]$-shares of $\delta_{T_i[j]W[j]} = \delta_{T_i[j]} \cdot \delta_{W[j]}$. Thus, to check for match in all positions, the parties have to generate $\delta_{T_i[j]W[j]} = \delta_{T_i[j]} \cdot \delta_{W[j]}$ for all $i \in \{0, 1, ..., s_T - s_P + 1\}$ and $j \in \{1, ..., s_P\}$. Observe that for the parties to perform this computation, they should be aware of the s_P and s_T.

We next describe how the servers can perform the preprocessing when the pattern and text size are not known. The preprocessing for approximate pattern matching also follows along the same lines with the only difference being the computations involved in the preprocessing phase. In the case of approximate pattern matching, the computations involve the generation of the keys for DPFs corresponding to the point $\delta_{T_i[j]} - \delta_{P[j]}$ for all $i \in \{0, 1, ..., s_T - s_P + 1\}$ and $j \in \{1, ..., s_P\}$.. On a high level, the servers preprocess many instances of wildcard pattern matching for fixed text and pattern size. Later, during the input-sharing phase, the servers receive multiple sharings of the same input depending on the size of the pattern and text. Elaborately, let m and n be the size of the pattern and text that the parties fix during the preprocessing. The parties in the preprocessing generate $[\delta_T[j]], [\delta_W[k]]$ and $\left[\delta_{T[j]W[k]}\right] = [\delta_T[j] \cdot \delta_W[k]]$ $j \in \{1, ..., m\}, k \in \{1, ..., n\}$. There are four possible cases:

Case 1: $s_P \leq m$ *and* $s_T \leq n$. Parties use $[\delta_T[j]]$ for $j \in \{1, ..., s_T\}$ and $[\delta_W[k]]$ for $k \in \{1, ..., s_P\}$ and discard the remaining preprocessing data. Thus, parties have all necessary preprocessing data to proceed with the computation.

Case 2: $s_P > m$ *and* $s_T \leq n$. Let $s_P \leq q \cdot m$. For simplicity assume that $q = 2$. In this case, parties take 2 copies of the shares of T as input denoted by T^1, T^2. Let $W = W^1||W^2$ such that W^1 is of size m and W^2 is of size $s_P - m$. In the preprocessing, the parties perform the necessary preprocessing corresponding to T^1, W^1 and T^2, W^2 (assuming W^2 is of length m). Thus, during the preprocessing, the parties generate $\left[\delta_{T^1[j]}\right], \left[\delta_{W^1[k]}\right]$ and $\left[\delta_{T^1[j]W^1[k]}\right]$ and $\left[\delta_{T^2[j]}\right], \left[\delta_{W^2[k]}\right]$ and $\left[\delta_{T^2[j]W^2[k]}\right]$ for $j \in \{1, ..., m\}, k \in \{1, ..., n\}$. Following this the parties have $[\![T^1]\!], [\![T^2]\!]$ and $[\![W^1]\!], [\![W^2]\!]$ after the input sharing phase. Now to check for the occurrence of pattern at position i, the parties construct the text substring T_i as $T^1[i : i + m]||T^2[i + m + 1 : s_P]$. Observe that, from $[\![T^1]\!], [\![T^2]\!]$, the parties can construct $[\![T_{ij}]\!]$ non-interactively. Following this, the

parties can proceed with the computation as they have the necessary preprocessing data corresponding to $[\![\mathsf{T}_i]\!]$ and $[\![\mathsf{W}]\!]$. For any $\mathsf{s_P} \le q \cdot \mathsf{m}$, the parties take q copies of T as input.

Case 3: $\mathsf{s_P} \le \mathsf{m}$ *and* $\mathsf{s_T} > \mathsf{n}$. Let $\mathsf{s_T} \le r \cdot \mathsf{n}$. For simplicity assume that $r = 2$. In this case, parties take 2 copies of the shares of W as input denoted by $\mathsf{W}^1, \mathsf{W}^2$. Let $\mathsf{T} = \mathsf{T}^1||\mathsf{T}^2$ such that T^1 is of size n and T^2 is of size $\mathsf{s_T} - \mathsf{n}$. In the preprocessing, the parties perform the necessary preprocessing corresponding to $\mathsf{T}^1, \mathsf{W}^1$ and $\mathsf{T}^2, \mathsf{W}^2$ (assuming T^2 is of length n). Thus, during the preprocessing, the parties generate $\left[\delta_{\mathsf{T}^1[j]}\right]$, $\left[\delta_{\mathsf{W}^1[k]}\right]$ and $\left[\delta_{\mathsf{T}^1[j]\mathsf{W}^1[k]}\right]$ and $\left[\delta_{\mathsf{T}^2[j]}\right]$, $\left[\delta_{\mathsf{W}^2[k]}\right]$ and $\left[\delta_{\mathsf{T}^2[j]\mathsf{W}^2[k]}\right]$ for $j \in \{1, ..., \mathsf{m}\}, k \in \{1, ..., \mathsf{n}\}$. Following this the parties have $[\![\mathsf{T}^1]\!]$, $[\![\mathsf{T}^2]\!]$ and $[\![\mathsf{W}^1]\!]$, $[\![\mathsf{W}^2]\!]$ after the input sharing phase. Observe that the parties have the necessary preprocessing to check for the occurrence of the pattern at positions 1 to $\mathsf{n} - \mathsf{s_P} + 1$. Similarly, for positions $\mathsf{n}+1$ to $\mathsf{s_T} - \mathsf{s_P} + 1$, the parties have the necessary preprocessing to check for the occurrence of the pattern. Now to check for the occurrence of the pattern at position $i \in \{\mathsf{n} - \mathsf{s_P} + 2, ..., \mathsf{n}\}$, the parties construct the text substring T_i as $\mathsf{T}^1[i : i + \mathsf{n}]||\mathsf{T}^1[\mathsf{n} + 1 : \mathsf{s_P}]$ and W as $\mathsf{W}^1[1 : \mathsf{n} - i]||\mathsf{W}^2[\mathsf{n} - i + 1 : \mathsf{m}]$. Observe that, from $[\![\mathsf{T}^1]\!]$, $[\![\mathsf{T}^2]\!]$, $[\![\mathsf{W}^1]\!]$ and $[\![\mathsf{W}^2]\!]$ the parties can construct $[\![\mathsf{T}_{ij}]\!]$ and $[\![\mathsf{W}^2]\!]$ non-interactively. Following this, the parties can proceed with the computation as they have the necessary preprocessing data corresponding to $[\![\mathsf{T}_i]\!]$ and $[\![\mathsf{W}]\!]$. For any $\mathsf{s_T} \le r \cdot \mathsf{m}$, the parties take r copies of W as input.

Case 4: $\mathsf{s_P} > \mathsf{m}$ *and* $\mathsf{s_T} > \mathsf{n}$. Let $\mathsf{s_P} \le q \cdot \mathsf{m}$ and $\mathsf{s_T} \le r \cdot \mathsf{n}$. This case is an amalgamation of cases 2 and 3 where the parties take q copies of T and r copies of W.

Depending on the application scenario, m and n can be chosen to ensure that q and r are small constants. Thus, the client-to-server communication remains $O(\mathsf{s_T} + \mathsf{s_P})$.

E Communication Complexity

We let ℓ denote the number of bits required to represent the character, let $\mathsf{s_P}$ denote the number of characters in the pattern P and $\mathsf{s_T}$ denote the number of characters in the text T. Let κ denote the computational security parameter and λ denote the hash size.

Lemma E.1: The protocol Π_{Ham} (Fig. 1) requires communication $\ell(\kappa + \ell)$ bits in the preprocessing, while it requires 1 round and 2ℓ bits communication in the online phase.

Proof. The preprocessing cost involves ℓ invocation of Π_{bit2A} which requires $\ell(\kappa + \ell)$ bits. The online phase involves one reconstruction which requires 1 round and 2ℓ bits of communication. □

Lemma E.2: The protocol Π_{EqZ} [16] instantiated with DPFs requires communication of $\ell\,(3\kappa + 2)$ bits in the preprocessing phase, while it requires 1 round and ℓ bits of communication in the online phase.

Proof. The preprocessing cost involves distributed generation DPF of which requires a communication of $\ell\,(3\kappa + 2)$ bits ([24]). The online phase involves one reconstruction which requires 1 round and 2ℓ bits of communication. □

Lemma E.3: The protocol Π_{Eq} (Fig. 2) when instantiated with Π_{EqZ} requires communication of $\ell(\kappa + \ell) + \log(2\ell)\,(3\kappa + 2)$ bits in the preprocessing phase, while it requires 2 rounds and $2(\ell + \log(2\ell))$ bits of communication in the online phase.

Proof. The costs follow from the costs of protocols Π_{Ham} and one invocation of EqZ instantiated with DPFs where $\log(2\ell)$ bits are required to represent the input. □

Lemma E.4: The protocol Π_{ExactPM} (Fig. 4) requires communication $(\mathsf{s_T} - \mathsf{s_P} + 1)\,(\lambda(\kappa + \lambda) + \log(2\lambda)(3\kappa + 2))$ bits in the preprocessing phase, while it requires 2 rounds and $2\,(\mathsf{s_T} - \mathsf{s_P} + 1)\,(\lambda + \log(2\lambda))$ bits of communication in the online phase.

Proof. This complexity follows from $(\mathsf{s_T} - \mathsf{s_P} + 1)$ parallel invocations of Π_{Eq} on input of size λ. □

Lemma E.5: The protocol Π_{WildPM} (Fig. 5) requires communication of $(\mathsf{s_T} - \mathsf{s_P} + 1)\,(\lambda(\kappa+\lambda)+\log(2\lambda)(3\kappa+2)+\mathsf{s_P}(\kappa+\ell))$ bits in the preprocessing phase, while it requires 2 rounds and $2\,(\mathsf{s_T} - \mathsf{s_P} + 1)\,(\lambda + \log(2\lambda))$ bits of communication in the online phase.

Proof. This complexity follows from $(\mathsf{s_T} - \mathsf{s_P} + 1)\,\mathsf{s_P}$ parallel invocations of $\Pi^{\mathsf{NI}}_{\mathsf{Mult}}$ followed by $(\mathsf{s_T} - \mathsf{s_P} + 1)$ invocations of Π_{Eq} on input of size λ. □

Lemma E.6: The protocol Π_{ApprxPM} (Fig. 6) requires communication of $(\mathsf{s_T} - \mathsf{s_P} + 1)\,(\mathsf{s_P}\ell(3\kappa + 2\ell) + \ell(3\kappa + 4))$ bits in the preprocessing, while it requires 1 round and $2\,(\mathsf{s_T} - \mathsf{s_P} + 1)\,\ell$ bits of communication in the online phase.

Proof. This complexity follows from $(\mathsf{s_T} - \mathsf{s_P} + 1)\,\mathsf{s_P}$ invocations of $\Pi_{\mathsf{EqZ}^\mathsf{A}}$ and $(\mathsf{s_T} - \mathsf{s_P} + 1)$ invocations of $\mathcal{F}_{\mathsf{comp}}$. □

F Benchmarks

F.1 Preprocessing Cost

We report the preprocessing cost of our equality and pattern matching protocols. We note that the preprocessing cost reported is just an estimate computed by accounting for all the computation and communication involved in the protocol. Table 9 reports the preprocessing cost of our equality protocol. Table 10, Table 11, and Table 12 report the preprocessing cost of our exact, wildcard and approximate pattern matching protocols, respectively.

Input size (B)	Run time (ms)	Comm. (KB)
1	294.01	0.17
2	369.50	0.35
8	374.64	1.62
32	376.21	12.40

Table 9: Preprocessing cost of our equality protocol for varying input sizes.

F.2 WAN Benchmarks

In this section, we report the benchmarks of our protocol on WAN. We consider a bandwidth of 100 Mbps and 100 ms of latency for WAN.

Text size (s_T)	Pattern size (s_P) l	Run time (min)	Comm. (MB)
100B	10B	0.01	8.93
1KB	10B	0.04	98.11
	100B	0.37	89.19
10KB	10B	0.40	989.90
	100B	3.47	980.98
	1KB	37.27	891.80

Table 10: Preprocessing cost of exact pattern matching for varying text size(s_T) and pattern size (s_P) for $\ell = 8$.

Text size (s_T)	Pattern size (s_P) l	Run time (min)	Comm. (MB)
100B	10B	0.01	9.02
1KB	10B	0.08	99.13
	100B	0.43	90.12
10KB	10B	0.52	1000.21
	100B	3.87	991.20
	1KB	54.30	901.09

Table 11: Preprocessing cost of wildcard pattern matching for varying text size(s_T) and pattern size (s_P) for $\ell = 8$.

Text size (s_T)	Pattern size (s_P) l	Run time (min)	Comm. (GB)
100B	10B	0.01	0.01
1KB	10B	0.21	0.03
	100B	1.03	0.24
10KB	10B	1.39	0.27
	100B	7.97	2.67
	1KB	115.39	24.20

Table 12: Preprocessing cost of approx pattern matching for varying text size(s_T) and pattern size (s_P) for $\ell = 8$.

F.2.1 Equality: Table 13 reports the comparison of a single invocation of the equality protocol while varying ℓ from 8 to 256. As expected, our protocol witnesses improvements of up to 4× and 2× over the circuit based protocols of [12] and [50] respectively. With respect to the DPF based protocol of [16], our protocol has a higher runtime and communication. This is because when considering a single invocation of equality over WAN, the overhead due to latency dominates the computational overhead of [16]. Yet, when multiple parallel invocations are considered, typical in applications like PHH, PPML, and PM, our equality protocol outperforms [16]. For a small number of parallel equality checks, [16] has better run time. However, as the number of parallel invocations increases, the computational overhead of [16] becomes dominant, and our protocol becomes increasingly competitive. Particularly, with 100 parallel equality checks, our costs are comparable with those of [16], and beyond 150 invocations, our protocol clearly outperforms

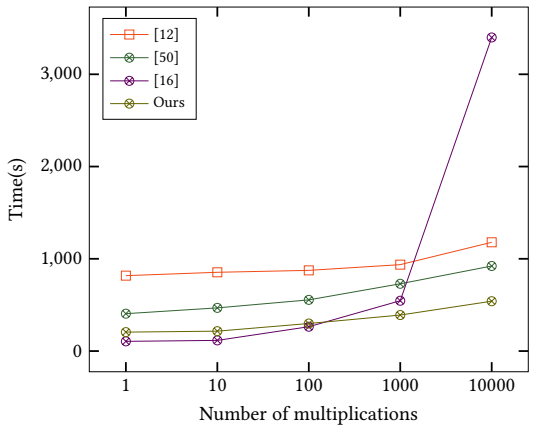

Figure 17: Comparison of equality for varying number of parallel equality invocations for $\ell = 256$.

it. Specifically, our improvements increase from 1.2× to 6.27× when increasing the number of parallel invocations of equality from 150 to 10000. This showcases that our protocols are more suitable for real-world applications.

Input size (B)	Protocol	Run time(ms)	Comm. (B)
1	[12]	303.45	3.50
	[50]	218.17	2.50
	[16]	101.32	2.00
	Π_{Eq}	199.75	3.00
2	[12]	443.60	7.50
	[50]	211.07	5.00
	[16]	102.73	4.00
	Π_{Eq}	198.29	5.25
8	[12]	490.89	31.50
	[50]	303.72	21.00
	[16]	115.27	16.00
	Π_{Eq}	196.92	17.75
16	[12]	817.44	127.50
	[50]	405.19	85.00
	[16]	105.22	64.00
	Π_{Eq}	205.26	66.25

Table 13: Comparison of equality protocol for varying input sizes on WAN.

F.2.2 Pattern matching: Table 14 reports the performance of exact pattern matching over WAN. We observe an improvement of up to 4 orders of magnitude in runtime in comparison to the naive variant. Table 15 reports performance of wildcard pattern matching. We observe an improvement of up to 4 orders of magnitude in runtime in comparison to the naive variant. Finally, Table 16 reports the performance of approximate pattern matching. Similar the the LAN setting we observe that our protocol only has slight improvement

over the naive variant. Further the runtime of the protocols remains comparable to that of WAN. This shows that the computation cost of aprroximate pattern matching dominates the run time.

Text size (s_T)	Pattern size (s_P)	Protocol	Run time (s)	Comm. (MB)
100B	10B	Naive	1.06	0.14
		Ours	2.07	0.05
1KB	10B	Naive	10.65	1.58
		Ours	2.13	0.51
	100B	Naive	98.06	14.40
		Ours	2.15	0.47
10KB	10B	Naive	108.50	15.98
		Ours	9.28	5.19
	100B	Naive	1056.67	158.40
		Ours	9.64	5.15
	1KB	Naive	9496.47	1440.02
		Ours	9.12	4.68

Table 14: Performance of exact pattern matching for varying text size (s_T) and pattern size (s_P) for $\ell = 8$ on WAN.

Text size (s_T)	Pattern size (s_P)	Protocol	Run time (s)	Comm. (MB)
100B	10B	Naive	1.29	0.37
		Ours	0.20	0.05
1KB	10B	Naive	10.94	4.12
		Ours	0.39	0.51
	100B	Naive	100.20	37.45
		Ours	0.32	0.47
10KB	10B	Naive	111.11	41.56
		Ours	0.98	5.19
	100B	Naive	1029.47	411.85
		Ours	0.96	5.15
	1KB	Naive	10250.84	3744.05
		Ours	2.50	4.68

Table 15: Performance of wildcard pattern matching for varying text size (s_T) and pattern size (s_P) for $\ell = 8$ on WAN.

Text size (s_T)	Pattern size (s_P)	Protocol	Run time (s)	Comm. (KB)
100B	10B	Naive	0.54	20.188
		Ours	0.47	2.88
1KB	10B	Naive	4.64	221.788
		Ours	4.18	31.68
	100B	Naive	38.16	201.628
		Ours	37.67	28.80
10KB	10B	Naive	42.25	2237.788
		Ours	42.00	319.68
	100B	Naive	418.11	2217.628
		Ours	413.46	316.80
	1KB	Naive	4484.34	2016.028
		Ours	4197.28	288.00

Table 16: Performance of approximate pattern matching for varying text size (s_T) and pattern size (s_P) for $\ell = 8$ on WAN.

G Security Proofs

In this section, we provide security proofs for our 2PC protocols in the standard real-world/ideal-world simulation paradigm. Let $\mathcal{A}$ denote the real-world adversary and $\mathcal{S}$ denote the ideal-world adversary. Without loss of generality, we provide the simulator proof for the case where $\mathcal{A}$ corrupts P_0. Since the protocol is symmetric, the simulation for the case of corrupt P_1 follows similarly. All our protocols are proven secure in the $\mathcal{F}_{\text{setup}}$-hybrid model where there exists an ideal functionality $\mathcal{F}_{\text{setup}}$ to establish common PRF keys among parties in $\mathcal{P}$. This allows the parties to sample common random values among themselves non-interactively. Note that the simulation begins with the simulator $\mathcal{S}$ emulating $\mathcal{F}_{\text{setup}}$ to establish the common keys with the adversary. Since $\mathcal{S}$ has access to the inputs and randomness of $\mathcal{A}$, it can simulate the steps in the real protocol. In the following, we first provide the ideal functionalities for the protocols followed by the security proofs.

Protocol Π_{Ham}. The ideal functionality for the hamming distance protocol Π_{Ham} (Fig. 1) appears in Fig. 18. Let $\mathcal{F}_{\text{Bit2A}}$ denote the ideal functionality for the bit to arithmetic conversion protocol, Π_{bit2A}. We have the following lemma.

Lemma G.1: The protocol, Π_{Ham} (Fig. 1) securely realizes the functionality $\mathcal{F}_{\text{Ham}}$ (Fig. 18) in the computational setting against a semi-honest adversary that corrupts at most one party in $\mathcal{P}$, in the $(\mathcal{F}_{\text{Bit2A}})$-hybrid model.

Functionality $\mathcal{F}_{\text{Ham}}$

$\mathcal{F}_{\text{Eqz-Exp}}$ interacts with parties in $\mathcal{P}$ and $\mathcal{S}$. It receives as input $[\cdot]$-shares of the x from the parties and $\mathcal{S}$, and proceeds as follows.

- Reconstruct x using the received $[\cdot]$-shares.
- Sample a random $r \in \mathbb{Z}_{2^\ell}$ and compute $c = d_H(x + r, r)$.
- Generate $[\![\cdot]\!]$-shares of c.
- Send $(\text{Output}, [c]_i)$ to P_i for $i \in \{0, 1\}$.

Figure 18: Ideal functionality for Π_{Ham}

Simulator $\mathcal{S}^{P_0}_{\Pi_{\text{Ham}}}$

$\mathcal{S}^{P_0}_{\Pi_{\text{Ham}}}$ proceeds as follows.

Preprocessing:

- Non-interactively generate $[\cdot]^{\mathbf{B}}$-shares for random bits $r_i \in \mathbb{Z}_2$ for $i \in \{0, \ldots, \ell - 1\}$.
- Emulates $\mathcal{F}_{\text{Bit2A}}$ on $[r_i]^{\mathbf{B}}$ to generate their $[\cdot]$-shares.

Online:

- Sample and send a random value $v \in \mathbb{Z}_{2^\ell}$ as the honest party's $[\cdot]$-share of a.

Figure 19: Simulator $\mathcal{S}^{P_0}_{\Pi_{\text{Ham}}}$ for corrupt P_0

Proof. The simulator $\mathcal{S}^{P_0}_{\Pi_{\text{Ham}}}$ for a corrupt P_0 appears in Fig. 19. The simulator emulates $\mathcal{F}_{\text{Bit2A}}$ to simulate the steps of Π_{bit2A}. Following this, $\mathcal{S}^{P_0}_{\Pi_{\text{Ham}}}$ sends a random value $v \in \mathbb{Z}_{2^\ell}$ to $\mathcal{A}$ as its $[\cdot]$-share of a′. Observe that emulating $\mathcal{F}_{\text{Bit2A}}$ guarantees indistinguishability of $\mathcal{A}$'s view in the ideal world and the real world. Moreover, the a reconstructed in the real world is random due to the use of the random mask r. In the simulation, since $\mathcal{A}$ receives a random value

from $\mathcal{S}^{P_0}_{\Pi_{\mathsf{Ham}}}$ for reconstructing a′, $\mathcal{A}$'s view in the real-world and ideal-world remain indistinguishable. Observe that the output of the honest party P_1 in the ideal world protocol is a randomly chosen share. The output of the honest party in the real protocol is computed as a linear combination of random values $[\mathsf{r}_i]_1$ for $i = 0$ to $\ell - 1$ (ref Fig. 1) and hence is a random value. Further, the output of the honest party is independent of the output of $\mathcal{S}^{P_0}_{\Pi_{\mathsf{Ham}}}$ in the ideal world, and similarly, the output of the honest party is independent of the view of $\mathcal{A}$ in the real world. Hence the joint distribution of the output of the $\mathcal{S}^{P_0}_{\Pi_{\mathsf{Ham}}}$ and the output of the P_1 in the ideal world protocol is indistinguishable from the joint distribution of the view of $\mathcal{A}$ and the output of the honest party in the real world protocol. □

Protocol Π_{Eq}. The ideal functionality for the equality protocol Π_{Eq} (Fig. 2) appears in Fig. 20.

Lemma G.2: The protocol, Π_{Eq} (Fig. 2) securely realizes the functionality $\mathcal{F}_{\mathsf{Eq}}$ (Fig. 20) in the computational setting against a semi-honest adversary that corrupts at most one party in $\mathcal{P}$, in the $(\mathcal{F}_{\mathsf{Ham}}, \mathcal{F}_{\mathsf{EqZ}})$-hybrid model.

Proof. The simulator for a corrupt P_0 appears in Fig. 21. The simulator emulates $\mathcal{F}_{\mathsf{Ham}}$. This is followed by local computations to generate $[\cdot]^{2\ell}$-shares of c. Finally, the simulator emulates $\mathcal{F}_{\mathsf{Eq}}$. Observe that emulating $\mathcal{F}_{\mathsf{Ham}}$ and $\mathcal{F}_{\mathsf{Eq}}$ guarantees indistinguishability of $\mathcal{A}$'s view in the ideal-world and real-world with respect to the corresponding MPC protocols. With respect to the local operations, the generation $[\mathsf{c}]^{2\ell}$ from $[\mathsf{c}]$ results in generating random shares of c over the ring $\mathbb{Z}_{2\ell}$. This can be argued as follows. The probability of picking a random value in $\mathbb{Z}_{2\ell}$ is $\frac{1}{2\ell}$. We will showcase that the probability of $[\mathsf{c}]^{2\ell}_i$ for $i \in \{0, 1\}$, generated via the ring change, mapping to a random value $\mathsf{k} \in \mathbb{Z}_{2\ell}$ is also $\frac{1}{2\ell}$. Observe that all values of the form $(\mathsf{k} + j \cdot 2\ell) \bmod 2^\ell$ when run through the ring change protocol map to k in $\mathbb{Z}_{2\ell}$. This is because $(\mathsf{k} \bmod 2^\ell + j \cdot 2\ell \bmod 2^\ell) \bmod 2\ell = \mathsf{k}$. Thus, there exist $\frac{2^\ell}{2\ell}$ unique values in $\mathbb{Z}_{2^\ell}$ that map to the same k in $\mathbb{Z}_{2\ell}$. Hence, the probability of $[\mathsf{c}]_i$ mapping to $\mathsf{k} \in \mathbb{Z}_{2\ell}$ is $\left(\frac{2^\ell}{2\ell}\right)/2^\ell = \frac{1}{2\ell}$. Finally, observe that the output of the honest party P_1 in the ideal world protocol is a randomly chosen share. The output of the honest party in the real protocol is a random value (ref Fig. 2). Further, the output of the honest party is independent of the output of $\mathcal{S}^{P_0}_{\Pi_{\mathsf{Ham}}}$ in the ideal world, and similarly, the output of the honest party is independent of the view of $\mathcal{A}$ in the real world. Hence joint distribution of output of $\mathcal{S}^{P_0}_{\Pi_{\mathsf{Ham}}}$ and the output of the P_1 in the ideal world protocol is indistinguishable from the joint distribution of the view of $\mathcal{A}$ and the output of the honest party in the real world protocol. □

Functionality $\mathcal{F}_{\mathsf{Eq}}$

$\mathcal{F}_{\mathsf{Eq}}$ interacts with parties in $\mathcal{P}$ and $\mathcal{S}$. It receives as input $[\cdot]$-shares of the x and y from the parties and $\mathcal{S}$, and proceeds as follows.

- Reconstruct x and y using the received $[\cdot]$-shares.
- If $\mathsf{x} = \mathsf{y}$, set $\mathsf{z} = 1$ else set $\mathsf{z} = 0$, and generate $[\mathsf{z}]^{\mathbf{B}}$.
- Send $(\mathsf{Output}, [\mathsf{z}]^{\mathbf{B}}_i)$ to P_i for $i \in \{0, 1\}$.

Figure 20: Ideal functionality for Π_{Eq}

Simulator $\mathcal{S}^{P_0}_{\Pi_{\mathsf{Eq}}}$

$\mathcal{S}^{P_0}_{\Pi_{\mathsf{Eq}}}$ proceeds as follows.

- Emulate the $\mathcal{F}_{\mathsf{Ham}}$ on $[\cdot]$-shares of $\mathsf{x}' = \mathsf{x} - \mathsf{y}$ to generate $[\cdot]$-shares of c.
- Computing $[\mathsf{c}]^{2\ell} = [\mathsf{c}] \bmod 2\ell$.
- Emulate $\mathcal{F}_{\mathsf{EqZ}}$ with input $[\mathsf{c}]^{2\ell}$.

Figure 21: Simulator $\mathcal{S}^{P_0}_{\Pi_{\mathsf{Eq}}}$ for corrupt P_0

Exact pattern matching. Observe that the protocol for exact pattern matching Π_{ExactPM} (Fig. 4) relies on invoking the equality protocol Π_{Eq}, whose security follows as described above. Hence, Π_{ExactPM} is computationally secure in the semi-honest adversarial setting with at most one corruption.

Wildcard pattern matching. Observe that the protocol for wildcard pattern matching Π_{WildPM} (Fig. 5) relies on $\Pi^{\mathsf{NI}}_{\mathsf{Mult}}$ and Π_{Eq} and performs some non-interactive operations. Note that the security of $\Pi^{\mathsf{NI}}_{\mathsf{Mult}}$ follows from [51] and the security of Π_{Eq} was established in Lemma G.2. Additionally, Π_{WildPM} relies on parties locally computing a collision-resistant hash on their $[\cdot]$-share and generating random $[\cdot]$-shares of these values, where the security of generating $[\cdot]$-shares also follows from [51]. Hence, Π_{WildPM} is computationally secure in the semi-honest adversarial setting with at most one corruption.

Approximate pattern matching. Observe that the protocol for approximate pattern matching Π_{ApprxPM} (Fig. 6) relies on $\Pi_{\mathsf{EqZ}^{\mathsf{A}}}$ and $\mathcal{F}_{\mathsf{comp}}$. The security follows from security of $\Pi_{\mathsf{EqZ}^{\mathsf{A}}}$ and $\mathcal{F}_{\mathsf{comp}}$ instantiated with DPFs and DCFs.