Homomorphic Encryption Enabled Delta Encoding

David Hasselquist*†, János Dani*, and Niklas Carlsson*
*Linköping University, Sweden
†Sectra Communications, Sweden

***Abstract*—The rapid expansion of cloud computing has transformed data storage and processing by providing unprecedented convenience and scalability. However, this progress is shadowed by significant data security challenges, primarily as users must rely on cloud service providers to enforce robust security protocols. Homomorphic encryption (HE) offers a potential solution by allowing computations on encrypted data, thereby maintaining confidentiality without compromising functionality. However, HE can be computationally intensive, raising concerns about its practicality in real-world applications, particularly when dealing with large files. Moreover, constantly re-encrypting an entire file after every modification is inefficient and introduces substantial performance overheads. While traditional delta encoding can be used to optimize bandwidth by transmitting only file modifications, it faces security and privacy concerns as the server must access unencrypted file contents. In this paper, we address these challenges by introducing a novel delta encoding scheme tailored for seamless compatibility with HE, enhancing data confidentiality while maintaining efficiency. Our approach minimizes the overhead of re-encryption and improves performance by encrypting and transmitting only the modified file parts. We evaluate the performance of our scheme across various parameters and test cases, comparing it to a state-of-the-art delta encoding approach. Our findings demonstrate many similarities while highlighting the tradeoffs of our HE-enabling solution. Additionally, we explore the performance impacts of integrating HE with our delta encoding scheme and provide insights into the practical constraints and requirements for real-world deployment in cloud computing environments.**

I. Introduction

The widespread adoption of cloud computing has increased data storage and processing, providing users worldwide with unparalleled convenience and scalability. However, this convenience comes with inherent risks, particularly concerning data security. When users entrust their data to cloud service providers, they lose control over its storage and processing, relying on the provider to uphold critical security measures [1]. These measures encompass a spectrum of concerns, from preventing intentional misuse of user data to ensuring robust security protocols that thwart unauthorized access.

A fundamental challenge in cloud computing lies in securing the transmission and storage of sensitive information. Traditional encryption methods offer a solution, but they often come at a significant cost to computational efficiency and data size. Homomorphic encryption (HE) presents a promising alternative by enabling computations on encrypted data, thereby safeguarding confidentiality while allowing for data manipulation [2]. However, adopting HE introduces challenges, including performance overhead and increased ciphertext size.

Moreover, in the context of handling large human-readable files, bandwidth optimization becomes important. Traditional approaches necessitate transmitting entire files, even for minor modifications, leading to inefficiencies in data transfer and storage. In addition, re-encrypting entire large files for small modifications is inefficient and comes with significant performance overheads. Delta encoding, a technique commonly used by cloud storage providers like Dropbox and iCloud, addresses this issue by having a client transmitting only file changes to a server. However, concerns arise regarding the security of delta encoding, particularly in preserving data confidentiality and integrity [3], as data is kept unencrypted on the servers. While there are solutions like client-side encryption [3], these do not allow computations on the encrypted data, putting a heavy load on the clients as well as the bandwidth.

Current delta encoding schemes often require accessing individual characters within dynamic containers, making them unsuitable for seamless integration with HE, which operates on non-indexable fixed-sized ciphertexts. Given the constraints of existing delta encoding methods, finding solutions that enable the interoperability of HE and delta encoding is necessary.

Therefore, in this paper, we first present a new basic delta encoding scheme designed with the use of HE in mind, enhancing data confidentiality while preserving performance efficiency, and evaluate the scheme under different parameters and test cases. By evaluating and comparing with a state-of-the-art delta encoding scheme, we show the many similarities in our design, while being tailored specifically to enable and leverage the benefits of HE. Next, we apply HE to our delta encoding scheme. By studying the performance overheads associated with this integration, we demonstrate the feasibility of our solution and shed further light on the constraints and requirements of deploying HE with delta encoding in real-world cloud computing environments.

Outline: We first present background on HE and delta encoding in Section II. Section III presents our new delta encoding scheme, its design, performance, and comparison to a state-of-the-art scheme. We combine HE with delta encoding and evaluate its performance in Section IV, present related work in Section V, and conclude in Section VI.

II. Background

A. Homomorphic encryption

Homomorphic encryption (HE), a concept once confined to theoretical discussions, has emerged as a practical tool in modern cryptography. HE allows for computations to be performed on encrypted data without prior decryption. The essence of HE lies in its ability to maintain data privacy while still enabling useful operations on that data. This capability has garnered significant attention, especially in fields where data privacy is paramount, such as facial recognition [4], privacy-preserving blockchains [5], medicine [6]–[8], finance [9], conferencing communication systems [10] and various machine learning problems [8], [11]–[16]. One prominent example is the use of HE in Microsoft's Edge browser, where users' encrypted login information is compared to known leaks [17]–[19].

At its core, HE enables operations on encrypted data, producing results as if performed on the plaintext. For instance, a server could multiply two encrypted numbers without knowing the actual values, delivering only the encrypted result back to the client for decryption. This ensures that sensitive data remains encrypted throughout processing, reducing the risk of exposure. Initially, schemes like RSA encryption [20] allowed for limited homomorphic operations, such as multiplication, but not addition, thus constraining its applicability.

The field advanced significantly with Gentry's groundbreaking work in 2009 [21], introducing the first fully homomorphic encryption (FHE) scheme. This scheme, albeit computationally intensive, enabled both addition and multiplication operations on encrypted data. Subsequent generations of HE schemes have evolved to enhance efficiency, security, and functionality. These schemes are classified based on their capabilities, ranging from partially to fully homomorphic.

One crucial optimization in achieving FHE is the bootstrapping process proposed by Gentry [2], effectively reducing noise in ciphertexts and allowing for indefinite homomorphic operations without compromising data integrity. However, the computational cost of bootstrapping remains a challenge. Some newer schemes have devised alternative methods to avoid the need for expensive bootstrapping, where two prominent examples are the BFV [22] and BGV [23] schemes.

The evolution of HE continues with each generation introducing novel approaches and improvements. For example, recent advancements, such as the CKKS scheme [24], have enabled the handling of approximate values like floating-point numbers. Despite progress, challenges persist, including balancing security, efficiency, and practicality.

As ASCII characters can be represented using integers, we use the BFV and BGV schemes implemented in the PALISADE [25] library, which offers multiple HE implementation schemes, conforming to the HE standard [26]. We selected the plaintext moduli to be 65 537, the default value in PALISADE, and the value required to fit UTF-8 characters. The remaining configuration values, i.e., the security level of 128 bits and a signal value of 3.2, were also left at their default values.

B. Delta encoding

Delta encoding produces a record of changes (delta) between an original and an updated set, enabling reconstruction of the updated set from both the original and the delta. This process can be repeated by generating additional deltas, which reconstruct the updated set when combined with the original. These deltas require accompanying specific operations that guide the delta encoding tool in reassembling the updated set.

VCDiff: Many popular delta encoding tools are built upon the foundational standard VCDiff [27]. VCDiff integrates compression and data difference to create delta compression, often known as delta encoding, producing delta files representing compressed changes between two file versions.

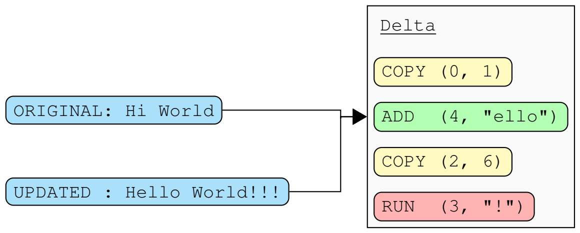


Fig. 1: Example of VCDiff encoding and generated operations.

The standard is characterized by four key features: output compactness, data portability, algorithm generic, and decoding efficiency. Output compactness refers to the creation of delta files that can be further compressed through configuration. The data portability aspect ensures platform independence of the resulting delta. Algorithm generic allows the decoding algorithm to remain independent of the encoding implementation, enabling improvements in encoding schemes without changing the decoding process. Finally, decoding efficiency improves decoding performance relative to the updated file size.

These functionalities are facilitated through string matching and window algorithms, which dictate the content of the delta, the portions to be copied from the original text, and the size of independently processable blocks [27]. Target window algorithms play a crucial role in the delta encoding process, creating non-overlapping sections called target windows within the updated file. These target windows are segments of the updated file that are compared against corresponding segments of the original text. This comparison generates a sequence of operations, allowing the reconstruction of the updated text when combined with the original text. These operations consist of three types: `COPY`, `ADD`, and `RUN`.

Figure 1 shows a simplified example of encoded operations. The initial `COPY` operation, with parameters 0 and 1, specifies the starting position in the original text and the number of characters to copy. Here, these parameters correspond to the character "H" at position 0 in the original text. Subsequently, an `ADD` operation with parameters 4 and "ello" adds characters absent in the original text. Another `COPY` operation is followed by a `RUN` operation. The `RUN` operation differs from the `ADD` operation in that the second parameter involves a single character repeated, determined by the first parameter, which in this case is 3. To decode the updated text, the operations are executed sequentially with reference to the original text.

Key observation: Several VCDiff implementations provide diverse optimizations, APIs, and interfaces. The most prominent examples are Xdelta [28] and Open-VCDiff [29]. Xdelta offers a command-line interface for VCDiff. Open-VCDiff, maintained by Google and being open-source, adds additional features and enhanced compression capabilities. While there are many VCDiff implementations, its dependency on individual character access within dynamic containers makes it unsuitable for seamless integration with HE, which operates with non-indexable fixed-sized ciphertexts.

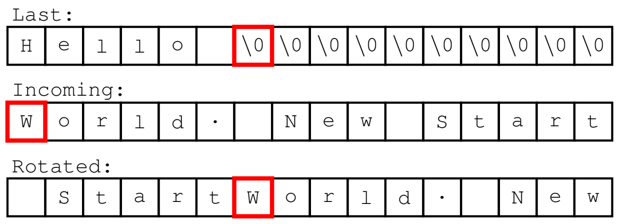

Last:
H e l l o \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
Incoming:
W o r l d · N e w S t a r t
Rotated:
S t a r t W o r l d · N e w

Fig. 2: Ciphertext right rotation of `Incoming` block with 6 steps to align with `Last` block, resulting in `Rotated` block.

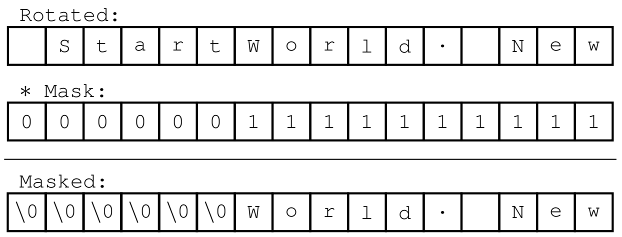

Rotated:
S t a r t W o r l d · N e w
* Mask:
0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1
Masked:
\0 \0 \0 \0 \0 \0 W o r l d · N e w

Fig. 3: Ciphertext multiplication of `Rotated` block (from Figure 2) with a `Mask` block, resulting in a `Masked` block.

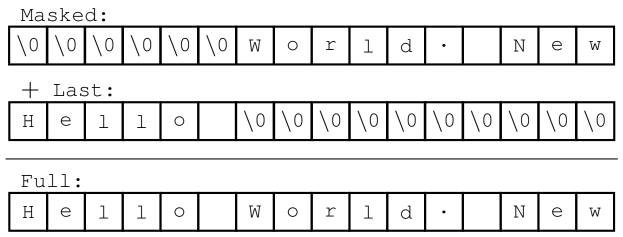

Masked:
\0 \0 \0 \0 \0 \0 W o r l d · N e w
+ Last:
H e l l o \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
Full:
H e l l o W o r l d · N e w

Fig. 4: Ciphertext addition of `Masked` block (from Figure 3) with `Last` block (from Figure 2), resulting in a `Full` block.

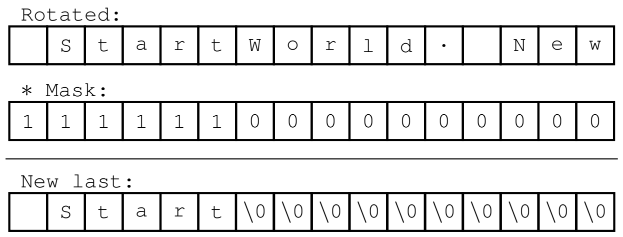

Rotated:
S t a r t W o r l d · N e w
* Mask:
1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0
New last:
S t a r t \0 \0 \0 \0 \0 \0 \0 \0 \0 \0

Fig. 5: Ciphertext multiplication of `Rotated` block (from Figure 2) with a `Mask` block, resulting in a `New last` block.

III. HE-enabling Delta encoding without HE

As a first step towards combining HE and delta encoding, we present a new delta encoding approach called Naive Delta Encoding (NDE) that addresses the specific constraints posed by the HE encryption models, and first evaluate the scheme without using HE. While we compare with Open-VCDiff, we note that the objective is not to outperform Open-VCDiff in efficiency. Instead, we strive to develop a proof-of-concept delta encoding method that aligns closely with Open-VCDiff while being designed to leverage the use of HE.

A. Design

Client-side encoding: Based on the VCDiff standard, NDE deterministically encodes the differences using `COPY` and `ADD` operations. The `COPY` operation functions identically to VCDiff, with two parameters specifying what to copy from the original. However, the `ADD` operation differs. Instead of storing the delta string in plaintext, the `ADD` operation in NDE instead points to a homomorphically encrypted ciphertext. The operations are stored separately from the encrypted text and together form the delta. By alternating between these two operations, the updated text can be encoded using the original and delta. In general, the encoding and creation of the delta using NDE is done using the following steps:

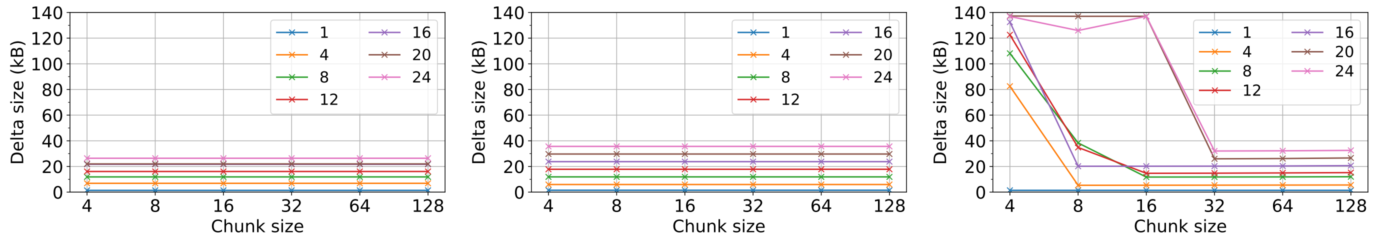
1) Identify the first position of mismatch between the original text (`OT`) and the updated text (`UT`). Add the `COPY` operation to the delta, specifying the initial position and the mismatch position.
2) Find the position where `OT` and `UT` resume matching. A predetermined variable, `chunk size`, dictates the number of consecutive characters from `OT` required to match in `UT`. This ensures that a minimal subset from `UT` is utilized for the `ADD` operation, thereby minimizing the total size of the delta.
3) Include the `ADD` operation in the delta, using the position and size determined for the subset in the previous step.
4) Iterate these steps until the end of the text, with each subsequent character serving as the initial position.

Server-side decoding: The server-side decoding processes an original ciphertext alongside a delta to generate an updated text. As HE cannot manage dynamic strings directly and is constrained to fixed-sized ciphertexts, the ciphertext is segmented into blocks, each represented by an encrypted ciphertext containing 4 096 8-bit characters (predetermined by the selected HE parameters). When extending the ciphertext with additional text, we utilize homomorphic ciphertext operations addition, multiplication, and rotation to correctly align the incoming block with the last empty position in the preceding block. By orienting and dividing the incoming block, we fill the empty spaces in the last block and create a new final block with any remaining characters.

First, using the ciphertext rotation operation, we rotate the incoming block such that the beginning of the incoming data aligns with the first empty position in the last block. Figure 2 illustrates this rotation, where the `Incoming` block is rotated 6 steps to the right, such that the highlighted position in the `Last` block aligns with the `Incoming` block.

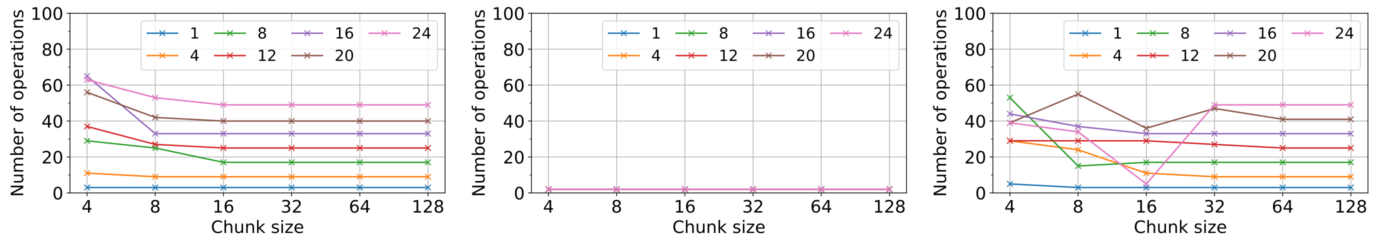
Next, we use the ciphertext multiplication operation and create a mask to extract the portion that fits inside the `Last` block from the `Incoming` block. Figure 3 demonstrates this masking multiplication, yielding the `Masked` block by multiplying the `Rotated` block with the `Mask`. The mask, comprising of zeros and ones, causes either null characters or the original 8-bit character values from the `Incoming` block.

Following that, we use ciphertext addition to merge the `Masked` block with the current `Last` block. Figure 4 shows this addition, combining the two blocks without destroying their individual components.



(a) add (b) append (c) replace

Fig. 6: Delta sizes over various chunk sizes using different test cases.



(a) add (b) append (c) replace

Fig. 7: Number of delta operations over various chunk sizes using different test cases.

Finally, any remaining part of the incoming block that wrapped around during rotation (as shown in Figure 2) becomes the new final ciphertext block. Figure 5 illustrates this process, again using ciphertext multiplication to mask the `Rotated` block, transforming it into the `New last` block. We note that this final step is only necessary if rotation occurs during the decoding process. If no rotation is required, the `Incoming` block directly becomes the `New last` block.

Key contribution: We have created a new delta encoding scheme that takes into consideration the specific constraints and limitations of HE. While not required, the scheme can utilize HE operations to add, multiply, and rotate ciphertext.

B. Evaluation

Framework: To more closely resemble a real-world use case, our evaluation framework consists of a local client and a virtual machine in the cloud. Our local client uses Linux Mint Cinnamon 20.2 with an 8-core Intel Core i7-2700K 3.5GHz and 16 GB RAM, while our cloud server uses Ubuntu 20.04 with a 16-core Intel Xeon E312xx 2.5GHz and 8GB RAM.

Test cases: For testing, we use the Alice in Wonderland book (148 480 bytes) from the Calgary collection of the Canterbury Corpus [30]. To generate a delta between an original version and an updated version, we employ the full text as the updated version and derive the original by replacing or removing various portions of the text. We categorize our test cases into three groups: `append`, `add`, and `replace`, each containing seven variations with different numbers of changes in the text. The `append` category represents a user adding text towards the end of the file, with each change representing an addition of 1% of the original book. For the `add` and `replace` categories, using a uniform distribution, we randomize (1) a size between 512 and 2 048 bytes, and (2) the starting position of the change. In the `add` category, text is added at a random position, while the `replace` category involves altering an existing text. In this case, we invert every character in that region. For instance, the sentence "Hello world" would become "dlrow olleH", giving a change that is unlikely to appear elsewhere in the text.

Size tuning: To determine the best chunk size parameter for client-side encoding in NDE, we run the test cases and obtain the delta sizes using different chunk sizes and number of delta operations. In general, a larger chunk size leads to fewer operations, with the `ADD` operations containing more data. This is in contrast to using a smaller chunk size, which results in a greater number of operations but with smaller `ADD` operations. Since each operation requires significant computational power, especially when performed homomorphically, and since sizes impact the data transmitted over a network, we aim to find a balance between computational time and delta size.

Delta size: Figure 6 shows the delta sizes for (a) the `add`, (b) `append`, and (c) `replace` categories over different chunk sizes and number of text changes. For the `add` and `append` categories, we observe a constant delta size, regardless of the chunk size, and as expected, a greater number of changes results in a larger delta. For the `replace` test case, we generally observe that a larger chunk size correlates with smaller delta sizes. However, regardless of the test case, we do not observe a decrease in delta size when using a chunk size larger than 32. In most cases, the decrease ends at 16.

These results show that different chunk sizes do not produce smaller deltas in the case of simple appends or additions in the middle of the text. However, when changes were made to existing text, i.e., the `replace` test cases, the size difference is quite clear, where larger chunk sizes produce smaller deltas. Here, a larger chunk size is better at determining actual changes and, as such, minimizes the delta size. Furthermore, we observe that the decrease of delta size stops at a specific chunk size, as it has reached the optimal value, with this stop being earlier with fewer text changes.

TABLE I: Comparison of NDE with Open-VCDiff using different changes and configurations for `add` test cases.

Test	Configuration	Delta size	#OP	Ratio
`add 1`	NDE 4,8,16,32	1 374 B	3	0.93 %
	VCD	1 409 B	3	0.95 %
`add 4`	NDE 4	6 878 B	11	4.63 %
	NDE 8,16,32	6 878 B	9	4.63 %
	VCD	6 896 B	11	4.64 %
`add 8`	NDE 4	11 860 B	29	7.99 %
	NDE 8	11 860 B	25	7.99 %
	NDE 16,32	11 860 B	17	7.99 %
	VCD	11 723 B	31	7.89 %
`add 12`	NDE 4	16 088 B	37	10.83 %
	NDE 8	16 088 B	27	10.83 %
	NDE 16,32	16 088 B	25	10.83 %
	VCD	16 027 B	30	10.79 %
`add 16`	NDE 4	21 686 B	65	14.60 %
	NDE 8.16,32	21 686 B	33	14.60 %
	VCD	21 606 B	45	14.55 %
`add 20`	NDE 4	21 986 B	56	14.80 %
	NDE 8	21 986 B	42	14.80 %
	NDE 16,32	21 986 B	40	14.80 %
	VCD	22 083 B	46	14.87 %
`add 24`	NDE 4	26 408 B	63	17.78 %
	NDE 8	26 408 B	53	17.78 %
	NDE 16,32	26 408 B	49	17.78 %
	VCD	26 220 B	72	17.66 %

Number of operations: Figure 7 shows the number of delta operations with different chunk sizes. For the `add` category (Figure 7a), we observe a reduction in the number of operations with larger chunk sizes, with diminishing returns beyond a chunk size of 16, as it reaches the optimal number of delta operations ($2n+1$, where n represents the number of changes). For the `append` category (Figure 7b), the number of operations remains constant at two, regardless of the chunk size or the number of changes. For the `replace` category (Figure 7c), we observe a similar trend with fewer operations using larger chunk sizes. However, a notable exception is observed with 24 `replace` changes, where a significant increase is seen from a chunk size of 16 to 32. As the delta size significantly drops for the same configuration (Figure 6c), this indicates difficulties in the encoding process in determining the actual difference, resulting in a larger delta size but fewer operations with a chunk size of 16, while the opposite holds true for a chunk size of 32.

Key observation: We determine the best chunk size to be 32 or less, observing little to no improvement with using a larger chunk size. Furthermore, using different test cases, we observe the most impact to be caused by replacing text.

C. Comparison with Open-VCDiff

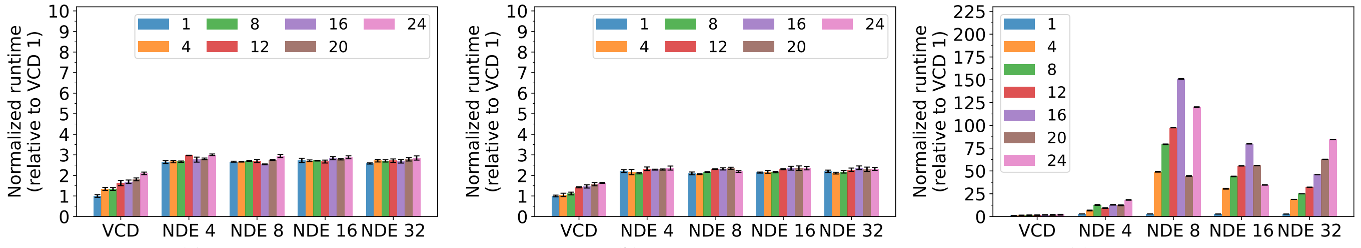
To show that our HE-enabled delta encoding approach does not add much overhead compared to basic OpenVCDiff, we next evaluate the performance and present a comparison, again using the test cases `append`, `add`, and `replace`. Here, we limit chunk sizes only up to 32, as we observe minimal improvements in the delta size or reduction in the number of operations with larger chunk sizes. Tables I and II present the comparison results for `add` and `append` scenarios, including the delta size, number of delta operations, and delta ratio relative to the original file size. The comparison reveals similar delta sizes between NDE and Open-VCDiff despite the lack of compression in NDE. This suggests that NDE is similarly good at identifying differences, occasionally surpassing Open-VCDiff by employing fewer operations. However, such parity is not evident for the `replace` category, as shown in Table III. For smaller chunk sizes, NDE struggles to generate reasonable-sized deltas, while Open-VCDiff tends to produce better delta sizes, albeit with more operations. Nevertheless, these results demonstrate that with a carefully chosen chunk size, NDE can perform similarly to Open-VCDiff.

TABLE II: Comparison of NDE with Open-VCDiff using different changes and configurations for `append` test cases.

Test	Configuration	Delta size	#OP	Ratio
`append 1`	NDE 4,8,16,32	1 485 B	2	1.00 %
	VCD	1 513 B	2	1.02 %
`append 4`	NDE 4,8,16,32	5 940 B	2	4.00 %
	VCD	5 942 B	4	4.00 %
`append 8`	NDE 4,8,16,32	11 880 B	2	8.00 %
	VCD	11 882 B	4	8.00 %
`append 12`	NDE 4,8,16,32	17 820 B	2	12.00 %
	VCD	17 740 B	10	11.95 %
`append 16`	NDE 4,8,16,32	23 760 B	2	16.00 %
	VCD	23 360 B	22	15.73 %
`append 20`	NDE 4,8,16,32	29 700 B	2	20.00 %
	VCD	29 192 B	29	19.66 %
`append 24`	NDE 4,8,16,32	35 640 B	2	24.00 %
	VCD	34 791 B	45	23.43 %

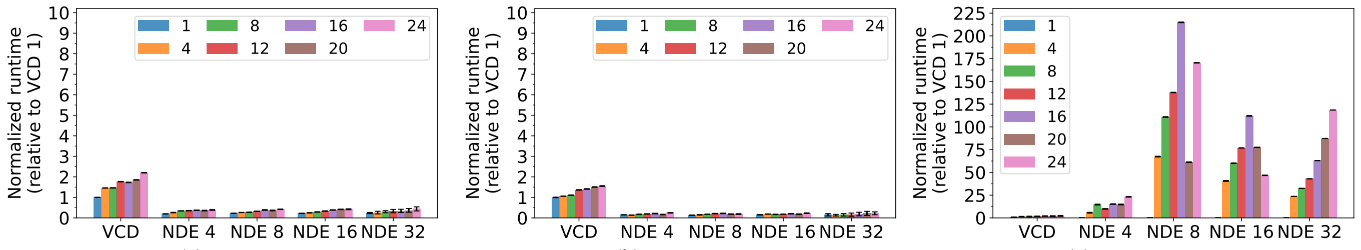
TABLE III: Comparison of NDE with Open-VCDiff using different changes and configurations for `replace` test cases.

Test	Configuration	Delta size	#OP	Ratio
`replace 1`	NDE 4	1 398 B	5	0.94 %
	NDE 8,16,32	1 398 B	3	0.94 %
	VCD	1 432 B	3	0.96 %
`replace 4`	NDE 4	82 521 B	29	55.57 %
	NDE 8	5 340 B	24	3.60 %
	NDE 16	5 388 B	11	3.63 %
	NDE 32	5 436 B	9	3.66 %
	VCD	5 337 B	16	3.59 %
`replace 8`	NDE 4	108 161 B	53	72.83 %
	NDE 8	38 344 B	15	25.82 %
	NDE 16	11 755 B	17	7.92 %
	NDE 32	11 803 B	17	7.95 %
	VCD	11 487 B	28	7.73 %
`replace 12`	NDE 4	122 597 B	29	82.55 %
	NDE 8	34 816 B	29	23.44 %
	NDE 16	14 676 B	29	9.88 %
	NDE 32	14 772 B	27	9.95 %
	VCD	14 429 B	39	9.72 %
`replace 16`	NDE 4	132 532 B	44	89.24 %
	NDE 8	20 177 B	37	13.59 %
	NDE 16	20 217 B	33	13.61 %
	NDE 32	20 265 B	33	13.65 %
	VCD	19 966 B	49	13.44 %
`replace 20`	NDE 4	137 251 B	39	92.42 %
	NDE 8	137 011 B	55	92.26 %
	NDE 16	137 032 B	36	92.27 %
	NDE 32	25 957 B	47	17.48 %
	VCD	25 247 B	80	17.00 %
`replace 24`	NDE 4	136 905 B	39	92.19 %
	NDE 8	125 948 B	34	84.81 %
	NDE 16	137 004 B	5	92.25 %
	NDE 32	32 046 B	49	21.58 %
	VCD	31 464 B	77	21.19 %



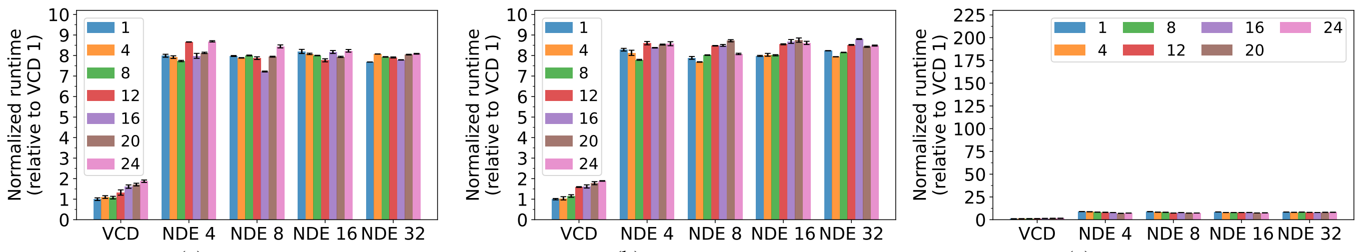
(a) add (b) append (c) replace

Fig. 8: Runtime (client and server) for NDE using different chunk sizes and VCDiff over 1–24 number of changes.



(a) add (b) append (c) replace

Fig. 9: Encoding time for NDE using different chunk sizes and VCDiff over 1–24 number of changes.



(a) add (b) append (c) replace

Fig. 10: Decoding time for NDE using different chunk sizes and VCDiff over 1–24 number of changes.

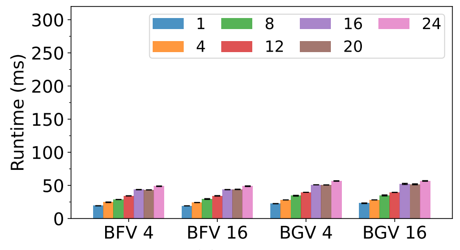
Runtime performance: While the delta sizes and number of delta operations are similar between NDE and Open-VCDiff, we observe larger differences in the runtime performance. Figure 8 shows the normalized runtime (averaged over 20 runs) for Open-VCDiff, as well as NDE using chunk sizes $\{4, 8, 16, 32\}$ when considering different numbers of changes (1 to 24). Here, we normalize all runtimes relative to Open-VCDiff with one change (which we denote as the baseline and give a normalized value of 1). Notably, for the `add` and `append` test cases (Figures 8a and 8b), we observe consistent performance, regardless of the chunk size or number of changes, with the runtimes being 2 to 3 times longer than the baseline. On the other hand, Open-VCDiff appears to have a more linear relation with the number of changes, indicating that increased changes lead to longer runtimes. However, in the `replace` test cases (Figure 8c), we instead observe a different behavior. Here, NDE exhibits significantly longer runtimes, with no discernible correlation between runtime and delta size or number of operations. However, for NDE with a chunk size of 32, a clear trend emerges with an increase in the number of changes resulting in longer runtimes. Similar patterns occur with chunk sizes of 8 and 16, but only up to 16 changes.

To better understand these timings, we study the encoding and decoding runtimes separately. Figures 9 and 10 show the performance results for client-side encoding and server-side decoding, respectively. Overall, NDE mostly spends its runtime on server-side decoding, while Open-VCDiff dedicates more time to encoding. More specifically, for encoding, NDE achieves better performance than Open-VCDiff for the `add` and `append` test cases. Moreover, we see that the variable and prolonged runtimes observed in total runtime for NDE (Figure 8c) stem from difficulties in encoding. Conversely, for decoding, consistent results are evident across test cases for NDE, with relatively stable performance, while Open-VCDiff shows a slight increase in runtimes with more changes.

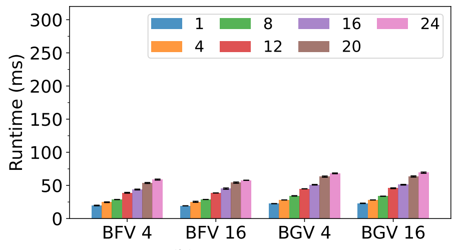
Key takeaway: Many similarities can be seen when comparing NDE to Open-VCDiff, especially regarding delta size and number of operations. Most of the runtime is spent on server-side decoding for NDE, and we again see the most impact to be related to when replacing text.

IV. Delta encoding with HE

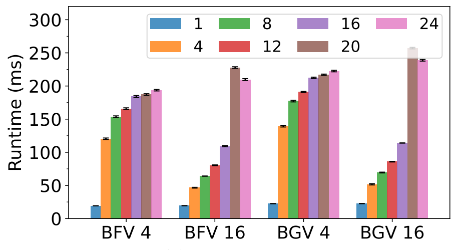
We next apply HE to our HE-enabled delta encoding to compute the updated file from the encrypted original file and the delta, and evaluate the extra overhead impact from HE. Initially, the client generates encryption keys, including multiplication and automorphism keys used for ciphertext calculations and rotations. Given that the number of rotation steps is unknown beforehand and the maximum rotation step is $2^{11} = 2\,048$ per ciphertext, we generate 11 automorphism keys



(a) `add`

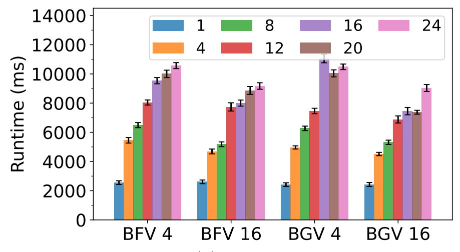


(b) `append`

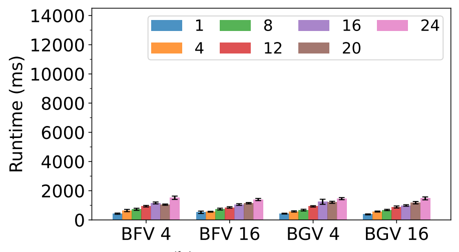


(c) `replace`

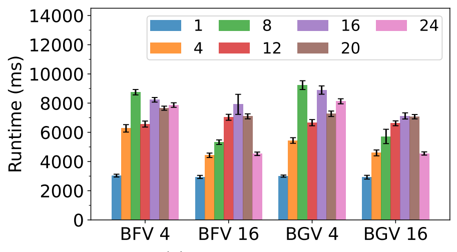
Fig. 11: Client-side runtime using homomorphic encrypted NDE per configuration and 1–24 number of changes.



(a) `add`



(b) `append`



(c) `replace`

Fig. 12: Server-side runtime using homomorphic encrypted NDE per configuration and 1–24 number of changes.

permitting 2^x rotations, where $x \in [1, 11]$. We then aggregate several keys based on the required rotation steps. As these keys are reusable for multiple encodings and decoding, we consider them as the initial setup cost between the client and server.

On the client side, we encrypt the original file and send it to the server. We then apply changes to the file, create a delta, and send it to the server. In practice, the original file and public keys would already be on the server, and only the delta needs to be sent. The client-side runtime includes creating the delta, encrypting it, and sending the ciphertext to the server.

On the server side, after receiving the original file and the delta, we evaluate the delta by homomorphically calculating the updated file and then verifying the correctness of the calculated file. The server runtime mainly consists of evaluating the ciphertext between the original encrypted file and the delta.

Evaluation: Figures 11 and 12 show the runtime when combining our delta encoding scheme with the HE algorithms BFV and BGV for the client and server sides, respectively. Using chunk sizes of 4 and 16 across 1 to 24 changes, we generally observe client runtime in the tens of milliseconds, contrasting with several seconds for the server, with increased changes correlating with longer processing time, underscoring the expense of HE ciphertext evaluation. Notably, an average of only 1.5% of the total time is spent on the client side.

While we again observe larger runtimes with a greater number of changes, we see marginal differences between the test cases, irrespective of using a chunk size of 4 or 16. This aligns with our expectations, as NDE encoding and decoding operate in the millisecond range, exerting minimal influence when coupled with HE. Finally, we observe that BFV exhibits marginally superior encoding performance compared to its BGV counterpart, while the opposite is true for decoding.

Key takeaway: When applying HE to delta encoding, we observe that most of the runtime is spent decoding on the server side, with client-side encoding being in the millisecond range. While decoding and calculating the updated file may take several seconds to complete, this operation may be less time-sensitive without requiring real-time response and may be acceptable to be performed in the background in the cloud.

V. Related Work

While we are the first to study the combination of HE and delta encoding in the cloud, some related works have focused on one of the two aspects, often tailored to specific use cases.

HE in the cloud: Early work in 2011 by Lauter et al. [31] demonstrated multiple practical applications of HE, some of which were utilized within cloud services, notably in the medical, financial, and advertising sectors. In 2014, Hrestak and Piecek [32] evaluated the suitability of applying HE to cloud services, noting both strengths and weaknesses. They observed that HE might not have been fully prepared for cloud computing at the time but recognized rapid advancements since Gentry's proposal of the FHE scheme [2]. Within a year of Gentry's proposal, the evaluation time for AES decreased significantly from 36 hours to just three hours [33]. Shortly after, Chauhan et al. [34] reached a similar conclusion regarding the potential of HE to address privacy concerns in cloud services, deeming current HE schemes to be insufficiently fast. Sethi et al. [35] introduced in 2017 a practical HE suitable for cloud services, enabling computations of basic arithmetic and simple statistical functions like averages. This implementation featured multi-thread support, resulting in an 80% efficiency increase over single-threaded implementations.

Alternative methods for secure computation in the cloud include the use of a trusted execution environment [36], possibly with the use of specialized hardware like Intel SGX [37].

Delta encoding: Numerous cloud storage solutions today employ various techniques to optimize bandwidth and storage efficiency. Drago et al. [38] compared Cloud Drive, Dropbox, Google Drive, SkyDrive, and Wuala, finding that Dropbox stands out by leveraging delta encoding alongside other features. While Dropbox demonstrated superior bandwidth-saving capabilities, its speed performance was only average. However, none of the five providers utilize client-side encryption (CSE) to prevent unencrypted data from residing on their servers, necessitating decryption for functionality. Expanding on this, Henzinger et al. [3] compared different cloud storage providers implementing CSE, assessing the overhead compared to non-CSE solutions. They observed that features like compression and deduplication [39] imposed similar overheads across both CSE and non-CSE services, and that delta encoding incurred greater overheads in CSE-based setups. Notably, CSE services exhibited higher bandwidth consumption, larger storage footprints, and increased resource usage for clients and servers. Subsequently, Henzinger et al. [40] proposed a model to reduce bandwidth and storage requirements while leveraging existing solutions. This model yielded a policy with a minimally compromised worst-case scenario, being only twice as slow as non-CSE solutions and generally outperforming them.

VI. Conclusion

In this paper, we have presented a novel delta encoding scheme compatible with HE, preserving data confidentiality while addressing the significant security concerns associated with traditional delta encoding methods. Our evaluation and comparison with Open-VCDiff demonstrate its effectiveness and many similarities, while also emphasizing the unique benefits of our HE-enabling solution. Our findings indicate that while there are performance impacts associated with HE, these may be manageable within the context of cloud computing environments, as most of the runtime is spent on decoding on the server side. Our approach enhances data confidentiality without sacrificing efficiency, thereby providing a solution for modern cloud computing environments.

Acknowledgment

This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation. Experimentation performed in this work was partially supported by the WASP WARA-Ops Research Arena.

References

[1] K. Ren, C. Wang, and Q. Wang, "Security challenges for the public cloud," *IEEE Internet Computing*, 2012.
[2] C. Gentry, "Fully homomorphic encryption using ideal lattices," in *Proc. ACM Symposium on Theory of Computing (STOC)*, 2009.
[3] E. Henziger and N. Carlsson, "Delta encoding overhead analysis of cloud storage systems using client-side encryption," in *Proc. IEEE Cloud Computing Technology and Science (CloudCom)*, 2019.
[4] P. Drozdowski, N. Buchmann, C. Rathgeb, M. Margraf, and C. Busch, "On the application of homomorphic encryption to face identification," in *Proc. Biometrics Special Interest Group (BIOSIG)*, 2019.
[5] S. Yaji, K. Bangera, and B. Neelima, "Privacy preserving in blockchain based on partial homomorphic encryption system for AI applications," in *Proc. High Performance Computing Workshops (HiPCW)*, 2018.
[6] M. Blatt, A. Gusev, Y. Polyakov, and S. Goldwasser, "Secure large-scale genome-wide association studies using homomorphic encryption," in *Proc. National Academy of Sciences*, 2020.
[7] M. Kim *et al.*, "Ultrafast homomorphic encryption models enable secure outsourcing of genotype imputation," *Cell systems*, 2021.
[8] A. Vizitiu, C. I. Niţă, A. Puiu *et al.*, "Applying deep neural networks over homomorphic encrypted medical data," *Computational and Mathematical Methods in Medicine*, 2020.
[9] D. Hasselquist, J. Wahlman, and N. Carlsson, "PET-Exchange: A privacy enhanced trading exchange using homomorphic encryption," in *Proc. International Conference on Privacy, Security and Trust (PST)*, 2023.
[10] D. Hasselquist, N. Johansson, and N. Carlsson, "Now is the time: Scalable and cloud-supported audio conferencing using end-to-end homomorphic encryption," in *Proc. ACM CCSW @ CCS*, 2023.
[11] S. Carpov, N. Gama, M. Georgieva *et al.*, "Privacy-preserving semi-parallel logistic regression training with fully homomorphic encryption," in *Proc. iDASH Privacy and Security Workshop*, 2020.
[12] H. Chabanne, A. de Wargny, J. Milgram, C. Morel, and E. Prouff, "Privacy-preserving classification on deep neural network," 2017.
[13] H. Chen, W. Dai, M. Kim, and Y. Song, "Efficient multi-key homomorphic encryption with packed ciphertexts with application to oblivious neural network inference," in *Proc. ACM CCS*, 2019.
[14] I. Chillotti, M. Joye, and P. Paillier, "Programmable bootstrapping enables efficient homomorphic inference of deep neural networks," in *Proc. Cyber Security, Cryptology, and Machine Learning*, 2021.
[15] R. Dathathri, B. Kostova, O. Saarikivi, W. Dai, K. Laine, and M. Musuvathi, "Eva: An encrypted vector arithmetic language and compiler for efficient homomorphic computation," in *Proc. ACM PLDI*, 2020.
[16] A. Kim, Y. Song, M. Kim, K. Lee, and J. H. Cheon, "Logistic regression model training based on the approximate homomorphic encryption," in *Proc. iDASH Privacy and Security Workshop*, 2018.
[17] H. Chen, Z. Huang, K. Laine *et al.*, "Labeled PSI from fully homomorphic encryption with malicious security," in *Proc. ACM CCS*, 2018.
[18] H. Chen, K. Laine, and P. Rindal, "Fast private set intersection from homomorphic encryption," in *Proc. ACM CCS*, 2017.
[19] Microsoft, "Password Monitor: Safeguarding passwords in Microsoft Edge," https://www.microsoft.com/en-us/research/blog/password-monitor-safeguarding-passwords-in-microsoft-edge/, 2021.
[20] R. L. Rivest, L. Adleman, and M. L. Dertouzos, "On data banks and privacy homomorphisms," *CiteSeerX*, 1978.
[21] C. Gentry, "Fully homomorphic encryption using ideal lattices," in *Proc. ACM Symposium on Theory of Computing (STOC)*, 2009.
[22] J. Fan and F. Vercauteren, "Somewhat practical fully homomorphic encryption." Cryptology ePrint Archive, Paper 2012/144, 2012.
[23] Z. Brakerski, C. Gentry *et al.*, "(leveled) fully homomorphic encryption without bootstrapping," *ACM Trans. Computing Theory (ToCT)*, 2014.
[24] J. H. Cheon, A. Kim, M. Kim, and Y. Song, "Homomorphic encryption for arithmetic of approximate numbers," in *Proc. ASIACRYPT*, 2017.
[25] "PALISADE Lattice Cryptography Library (release 1.11.2)," https://palisade-crypto.org/, 5 2021.
[26] M. Albrecht *et al.*, "Homomorphic encryption security standard," HomomorphicEncryption.org, Tech. Rep., 2018.
[27] K.-P. Vo, D. G. Korn, J. C. Mogul, and J. P. MacDonald, "The VCDIFF generic differencing and compression data format," RFC 3284, 2002.
[28] J. MacDonald, "xdelta," http://xdelta.org.
[29] Google, "open-vcdiff," https://github.com/google/open-vcdiff/.
[30] M. Powell, "The canterbury corpus," https://corpus.canterbury.ac.nz/index.html.
[31] M. Naehrig, K. Lauter, and V. Vaikuntanathan, "Can homomorphic encryption be practical?" in *Proc. ACM CCSW @ CCS*, 2011.
[32] D. Hrestak and S. Picek, "Homomorphic encryption in the cloud," in *Proc. MIPRO*, 2014.
[33] M. van Dijk, C. Gentry, S. Halevi *et al.*, "Fully homomorphic encryption over the integers," in *Proc. EUROCRYPT*, 2010.
[34] K. K. Chauhan, A. K. Sanger, and A. Verma, "Homomorphic encryption for data security in cloud computing," in *Proc. IEEE ICIT*, 2015.
[35] K. Sethi, A. Majumdar, and P. Bera, "A novel implementation of parallel homomorphic encryption for secure data storage in cloud," in *Proc. IEEE Cyber Security*, 2017.
[36] A. Baumann, M. Peinado, and G. Hunt, "Shielding applications from an untrusted cloud with haven," in *Proc. USENIX OSDI*, 2014.
[37] V. Costan and S. Devadas, "Intel SGX explained," 2016.
[38] I. Drago, E. Bocchi, M. Mellia, H. Slatman, and A. Pras, "Benchmarking personal cloud storage," in *Proc. ACM IMC*, 2013.
[39] Z. Li, C. Jin, T. Xu, C. Wilson, Y. Liu *et al.*, "Towards network-level efficiency for cloud storage services," in *Proc. ACM IMC*, 2014.
[40] E. Henziger and N. Carlsson, "The overhead of confidentiality and client-side encryption in cloud storage systems," in *Proc. IEEE/ACM Utility and Cloud Computing (UCC)*, 2019.