Scientific Research Publishing

American Journal of Computational Mathematics, 2026, 16(2), 44-79
https://www.scirp.org/journal/ajcm
ISSN Online: 2161-1211
ISSN Print: 2161-1203

Triangular Number Matrix and Application in Cryptography: A Novel Pattern-Based Approach

Nikoloz (Niko) Nadirashvili

Independent Researcher, New York, USA
Email: admin@matrixcoders.io

How to cite this paper: Nadirashvili, N. (2026) Triangular Number Matrix and Application in Cryptography: A Novel Pattern-Based Approach. *American Journal of Computational Mathematics*, **16**, 44-79.
https://doi.org/10.4236/ajcm.2026.162004

Received: August 18, 2025
Accepted: May 12, 2026
Published: May 15, 2026

Abstract

Purpose: The calculation of triangular numbers using the conventional formula $T_n = n(n+1)/2$ becomes computationally infeasible for astronomically large values of n (e.g., numbers with 10 trillion digits), due to limitations in modern computing systems. Existing methods struggle with the division and multiplication of extremely large integers, resulting in either failure or prohibitively slow processing times. This paper proposes a novel triangular number system that leverages mathematical patterns and constants to derive efficient computational shortcuts for determining high-digit triangular numbers. By circumventing the inefficiencies of traditional arithmetic operations, this approach aims to enable the rapid calculation of triangular numbers at scales previously deemed intractable. The proposed methodology synthesizes recursive pattern recognition and optimized calculation techniques to overcome the bottlenecks inherent in current computational frameworks. Experimental validation, benchmark comparisons, and theoretical analysis indicate that the system offers substantial advantages in handling ultra-large numbers, representing a significant advancement in computational number theory. The application of this pattern based analysis of triangular numbers has also been formulated and shown using TLV (Triangular layered verification) based cryptographic protocol.

Keywords

Triangular Numbers, Repetitive Digits, Multi-Base Number Systems, Pattern Symmetry, Pattern Assembly, Large Integer Generation, Catalog Based Verification, Cryptographic Protocol

1. Introduction

Triangular numbers have long attracted mathematicians because they stand at a

rare intersection of simplicity and structure. They arise from one of the most elementary constructions in arithmetic—the successive accumulation of natural numbers—yet they often reveal deeper regularities when examined through repetition, positional growth, and digit organization. In mathematics, such regularities are rarely mere visual curiosities. They often indicate that an object admits a shorter internal description than its standard formula first suggests.

This paper studies triangular numbers generated from repdigits. A *repdigit* is a number whose digits are all identical in a fixed base, such as 777_{10} or 1111_2. Although repdigits are often treated as simple notational objects, their associated triangular numbers exhibit stable constants, repeating internal blocks, and scale-dependent selectors that can be organized into a reproducible construction system.

Any n^{th} triangular number can be written in the familiar form

$$T_n = \frac{n(n+1)}{2},$$

and for ordinary values of n, direct evaluation is entirely natural. For extremely large values, however, the conventional route still depends on explicit large-integer arithmetic, especially multiplication and division, whose cost grows with the size of the representation [1]-[3]. This paper studies a different possibility: that for certain structured families of integers, the triangular number need not always be obtained by pushing the input through the full arithmetic pipeline, but may instead be assembled directly from stable internal rules already present in the number family itself.

When repdigits are examined from this viewpoint, they reveal a framework of reusable constants, directional pattern blocks, and digital-root-based selection rules. In the families studied here, these are not isolated observations but parts of a constructive system for writing triangular numbers directly from patterned components. In that sense, the Triangular Number Matrix system developed in this paper is not merely a faster implementation of a known formula. It is a structured method for repdigit triangular-number generation in which long outputs can be recovered from compact internal rules.

To provide an intuitive first view of the Triangular Number Matrix system, **Figure 1** shows a simplified example for repdigit 1. The figure highlights the synchronization between the vertical pattern, the pattern index, and the horizontal pattern as the repdigit length increases. The dark green rows mark the key cycle alignments: every ninth increase in the pattern index causes the vertical pattern to complete a full cycle and repeat, while the horizontal pattern expands by adding one more repeated block on each side. In the figure, the vertical constants are highlighted in bold red, while the horizontal pattern blocks are shown in purple on the left and brown on the right. The placement of the pattern index between the vertical and horizontal components makes this synchronization visually explicit, since it acts as the transition point linking the recycling of the vertical constants with the growth of the horizontal pattern structure. This introductory representation is deliberately simple and serves only as an entry point before the more

detailed matrix constructions introduced later.

The resulting shift may be summarized as *assembly versus compute*. Direct T_n formula methods evaluate the triangular number through explicit arithmetic on large integers. The Triangular Number Matrix system instead reduces the task to pattern selection, constant placement, scalable reduction, and structured writing. This distinction is not cosmetic. It changes the dominant workload from arithmetic growth to controlled construction. As the benchmark section later shows, this leads to substantial speedups over direct T_n formula implementations in the tested repdigit settings, while also motivating multiple generation modes tailored to different use cases, including transient in-memory generation, persistent large-catalog construction, and compact descriptor-based access.

A second contribution of the paper is that these pattern rules are not confined to isolated base-10 examples. The analysis extends into multi-base settings, where the same underlying number may expose different structured representations across bases. As discussed later in the multi-base analysis, the patterned behavior identified for repdigit triangular numbers is not confined to base 10, but also reappears in other numeral systems. Accordingly, the evidence suggests that the structural principles developed here belong to a broader multi-base framework rather than to a single-base curiosity. In this view, repetitive digits are not trivial repetitions; they behave more like a hidden language whose constants, anchoring abilities, and pattern symmetries can be organized into a computational grammar. Some numbers, at least in these families, appear to prefer to be assembled rather than computed.

The significance of this discovery is not only mathematical or computational. Modern cryptography has evolved into a broad discipline combining mathematics, computer science, and engineering to secure communication against increasingly diverse threats [4]-[6]. Within that landscape, this paper also explores a bridge from patterned triangular-number generation to a cryptographic protocol. Because the resulting triangular-number structures are dense, bounded, and selectively addressable, they admit catalog-oriented verification workflows that support the TLV (Triangular Layered Verification) protocol introduced later in the paper. The mathematics is therefore not presented here as an isolated numerical curiosity, but as the basis for a broader computational and cryptographic design space.

The remainder of the paper develops this framework in stages. We first identify the constants and repeating structures that govern repdigit triangular numbers, then show how these rules support direct construction in the Triangular Number Matrix system, followed by benchmark comparisons against direct T_n formula implementations. We then extend the discussion to multi-base behavior and finally bridge the mathematical framework to TLV, where the same pattern-based ideas support a proof-of-concept cryptographic protocol for layered verification and rolling authentication state.

The paper is organized into eight sections. Section 2 presents the methodology

used to identify recurring patterns and constants in repdigits. Section 3 develops the repdigit triangular-number calculator based on these patterns and constants for direct construction of large triangular numbers. Section 4 presents benchmark comparisons between the proposed calculator and the Python libraries sympy and gmpy2. Section 5 extends the discussion to non-repeated digit triangular-number calculations in base 10 and to multi-base representations. Section 6 connects the mathematical framework to cryptography and outlines the advantages of the Triangular Number Matrix system in the context of current and emerging cybersecurity threats. Section 7 introduces the TLV (Triangular Layered Verification) system as a proof-of-concept authentication protocol. It defines the core protocol components, including committed catalogs, per-layer operation templates, bounded argument tags, and a session nonce that refreshes per-session rules without persisting them server-side, and situates TLV within the broader threat model developed in Section 6. Section 8 concludes the paper. Supplementary A provides Supplementary Figures S1-S6, while Supplementary B presents future research directions for extending the proposed framework.

2. Methodology for Finding Patterns in Repdigits

The repdigits generated from 1 to 9, when calculating their corresponding triangular number, always have a pattern that ascends with increasing the digits of any given repdigit number. If we correctly note down constants associated with these patterns, then writing the triangular number for the next higher repdigit number is possible. We calculate the constants and associated patterns for each of these repdigits in the following sections. The side-by-side repdigit comparison Table 1 for the 3, 6, and 9 families is shown below.

Table 1. For repdigits 3, 6 and 9.

Repdigit	Triangular	Repdigit	Triangular	Repdigit	Triangular
3	6	6	21	9	45
33	561	66	2211	99	4950
333	55611	666	222111	999	499500
3333	5556111	6666	22221111	9999	49995000
33333	555561111	66666	2222211111	99999	4999950000
333333	55555611111	666666	222222111111	999999	499999500000
3333333	5555556111111	6666666	22222221111111	9999999	49999995000000
33333333	55555556111111	66666666	222222221111111	99999999	499999995000000
Triangular number series of repdigits of 3		Triangular number series of repdigits of 6		Triangular number series of repdigits of 9	

2.1. Repdigits Based on 3, 6, and 9

The simplest triangular-number patterns are related to repdigits generated from the 3, 6, and 9 series. The side-by-side table above shows the corresponding triangular numbers, and the pattern for repdigits of 3 can be written as the following theorem:

Theorem 1. *For any repdigit* $[3]_m$ *with m number of digits in it, the corresponding triangular number can be written as* $[5]_{m-1}[6]_1[1]_{m-1}$.

$$\mathbf{Trag}\left([3]_m\right)=[5]_{m-1}[6]_1[1]_{m-1} \tag{1}$$

Similarly, the same side-by-side table suggests the following statements for repdigits of 6 and 9:

Theorem 2. *For any repdigit* $[6]_m$ *with m number of digits in it, the corresponding triangular number can be written as* $[2]_m[1]_m$.

$$\mathbf{Trag}\left([6]_m\right)=[2]_m[1]_m \tag{2}$$

Theorem 3. *For any repdigit* $[9]_m$ *with m number of digits in it, the corresponding triangular number can be written as* $[4]_1[9]_{m-1}[5]_1[0]_{m-1}$.

$$\mathbf{Trag}\left([9]_m\right)=[4]_1[9]_{m-1}[5]_1[0]_{m-1} \tag{3}$$

2.2. Repdigits Based on 1, 2, 4, 5, 7, 8

The triangular number of repdigits based on 1, 2, 4, 5, 7 and 8 does not follow the simpler form as shown in the previous subsection. We now try to find the patterns and constants in this sequence using the tables attached below. A colour-coded view to identify the constants and patterns for each repdigit, as presented in **Figure 2** used as a primary sample in this research paper, along with Supplementary **Figures S1-S5**. Let's understand the constants and patterns with **Figure 2**. It consists of vertical patterns of constants, the differences between successive constants, and digits inherited from one constant to the next. Vertical patterns have left and right patterns used to assemble the triangular number of a repdigit.

The colour-coded view in **Figure 2** shows that a hidden three-digit constant vertical pattern follows in the middle of triangular numbers as it ascends. The pattern again repeats itself after every 9 triangular numbers. The symmetrical pattern starts from the second triangular number 66 and can be written in form of a set as,

$$V_{\text{const}} \in \{[660],1[216],771,1[317],882,1[438],993,1[549],104\}$$

The common feature of these vertical constants is that consecutive terms alternate between differences of 555 and 556, as shown in **Figure 2**. With further research, this regularity may enable the next constant to be computed directly, rather than determined manually. A further observed feature is that certain digits are inherited from the m^{th} triangular number when constructing the $(m+1)^{\text{th}}$ triangular number. These inherited digits form a closed loop, analogous to the constant patterns discussed above, and may provide insight toward logarithmic

derivations (or compact generation rules) for values of V_{const}. Further details are provided in Section B.

REPDIGIT	TRIANGULAR NUMBERS MATRIX		
	VERTICAL PATTERN	PATTERN INDEX	HORIZONTAL PATTERN
[1]	[1]	0	[1]
[11]	[66]	1	[66]
[111]	[6216]	2	[6216]
[1111]	[617716]	3	[617716]
[11111]	[617327 16]	4	[617327 16]
[111111]	[6172882716]	5	[6172882716]
[1111111]	[617284382716]	6	[617284382716]
[11111111]	[61728399382716]	7	[61728399382716]
[111111111]	[617283954938 2716]	8	[617283954938 2716]
[1111111111]	[61728395104938 2716]	9	[61728395104938 2716]
[11111111111]	[617283950660 49382716]	10	[617283950660 49382716]
[111111111111]	[6172839506216049382716]	11	[6172839506216049382716]
[1111111111111]	[61728395061771 6049382716]	12	[61728395061771 6049382716]
[11111111111111]	[617283950617327 16049382716]	13	[617283950617327 16049382716]
[111111111111111]	[6172839506172882716049382716]	14	[6172839506172882716049382716]
[1111111111111111]	[617283950617284382716049382716]	15	[617283950617284382716049382716]
[11111111111111111]	[617283950617283993827160493827 16]	16	[617283950617283993827160493827 16]
[111111111111111111]	[6172839506172839549382716049382716]	17	[6172839506172839549382716049382716]
[1111111111111111111]	[61728395061728395104938271604938271 6]	18	[61728395061728395104938271604938271 6]
[11111111111111111111]	[617283950617283950660493827160493827 16]	19	[617283950617283950660 049382716 049382716]

Figure 1. Simplified introductory view of the Triangular Number Matrix system for repdigit 1. The figure highlights the synchronization between vertical patterns, pattern index, and horizontal patterns. The dark green rows show cycle alignment: every ninth increase in the pattern index completes a full vertical-pattern cycle, while the horizontal-pattern composition expands by one additional repeated block on each side.

repdigit-1

INDEX	REPDIGITS	TRIANGULAR NUMBER	VERTICAL PATTERNS		
			CONSTANTS	DIFFERENCE	DIGITS INHERITED
	[1]	[1]			
1	[11]	[66]	[66+0]	556	
2	[111]	[6216]	1[216]	555	[6,6]
3	[1111]	[617716]	[771]	556	[1,6]
4	[11111]	[617327 16]	1[327]	555	[7,7,1]
5	[111111]	[6172882716]	[882]	556	[2,7]
6	[1111111]	[617284382716]	1[438]	555	[8,8,2]
7	[11111111]	[61728399382716]	[993]	556	[38]
8	[111111111]	[6172839549382716]	1[549]	555	[9,9,3]
9	[1111111111]	[61728395104938 2716]	[104]	556	[4,9]
1	[11111111111]	[617283950660 49382716]	[660]	556	[0,4]
2	[111111111111]	[6172839506216049382716]	1[216]	555	[6,6,0]
3	[1111111111111]	[61728395061771 6049382716]	[771]	556	[1,6]
4	[11111111111111]	[617283950617327 16049382716]	1[327]	555	[1,6]
5	[111111111111111]	[6172839506172882716049382716]	[882]	556	[2,7]
6	[1111111111111111]	[617283950617284382716049382716]	HORIZONTAL PATTERNS		
7	[11111111111111111]	[61728395061728399382716049382716]			
8	[111111111111111111]	[6172839506172839549382716049382716]	LEFT		RIGHT
9	[1111111111111111111]	[617283950617283951049382716049382716]	[617283950]		[049382716]

Figure 2. Triangular number series of repdigits of 1.

Apart from this three-digit vertical pattern, there is also a noticeable horizontal pattern followed by the triangular number series.

- Left Horizontal Pattern—this represents the patterns or parts of patterns that repeat horizontally on the left side of the vertical constant.

$$L_{\text{const}} = 617283950$$

- Right Horizontal Pattern—this represents the patterns or parts of patterns that repeat horizontally on the right side of the vertical constant.

$$R_{\text{const}} = 049382716$$

So far, we have clearly identified all the vertical and horizontal constants and symmetries present in them. Now, we have all the ingredients for writing a triangular number of any n^{th} repdigit $[1]_n$.

3. Triangular Number Calculator

To build the triangular number calculator, for any m^{th} digit rep number using the constants discussed in the previous section, we use the scalable reduction method (Figure 3). One of the reasons this method is scalable is that the digital root of a large repdigit can be computed by passing only a few parameters to a parallel process (e.g., repdigit = 1, length = 91,234,402), whereas for a non-repeating digit sequence, we would need to pass all 91,234,402 digits. For demonstration and maintaining the continuity of the article, we again take the example of repdigit $[1]_m$ to explain the assembly process of vertical and horizontal patterns, and the role of the digital root in the derivation of the triangular number.

INDEX	REPDIGIT	REPDIGIT LENGTH	APPLY SCALABLE REDUCTION METHOD DIGITAL ROOT	PATTERN INDEX	REPDIGIT LENGTH −2	TRIANGULAR NUMBER
1	[1]					[1]
2	[11]	2	2	1	0	[66]
3	[111]	3	3	2	1	[6216]
4	[1111]	4	4	3	2	[6177 16]
5	[11111]	5	5	4	3	[61732716]
6	[111111]	6	6	5	4	[6172882716]
7	[1111111]	7	7	6	5	[617284382716]
8	[11111111]	8	8	7	6	[61728399382716]
9	[111111111]	9	9	8	7	[6172839549382716]
10	[1111111111]	10	1	9	8	[617283951049382716]
11	[11111111111]	11	2	1	9	[61728395066049382716]
12	[111111111111]	12	3	2	10	[617283950216049382716]
13	[1111111111111]	13	4	3	11	[6172839506177160493827 16]
14	[11111111111111]	14	5	4	12	[61728395061732716049382716]
15	[111111111111111]	15	6	5	13	[617283950617288271604938271 6]
16	[1111111111111111]	16	7	6	14	[6172839506172843827160493827 16]
17	[11111111111111111]	17	8	7	15	[617283950617283993827160493827 16]
18	[111111111111111111]	18	9	8	16	[6172839506172839549382716049382716]
19	[1111111111111111111]	19	1	9	17	[6172839506172839510493827160493827 16]
20	[11111111111111111111]	20	2	1	18	[617283950617283950660493827160493827 16]
21	[111111111111111111111]	21	3	2	19	[6172839506172839506216049382716049382716]
22	[1111111111111111111111]	22	4	3	20	[61728395061728395061771604938271604938271 6]
23	11111111111111111111111	23	5	4	21	[617283950617283950617327160493827160493827 16]
24	111111111111111111111111	24	6	5	22	[6172839506172839506172882716049382716049382716]

Figure 3. Triangular number assembly methodology based on scalable reduction method, using repdigits digital root as a key for finding corresponding vertical constant.

Figure 3 shows the scalable reduction method in writing the triangular repdigit $[1]_m$. Take the example of the highlighted repdigit $[1]_{17}$. Clearly, the repdigit length is $m=17$ (Algorithm 1).

Algorithm 1. Repdigit triangular-number assembly using the scalable-reduction key.

Require: repdigit family d, repdigit length m, horizontal period P_ℓ, family constants $L_{\text{const}}, R_{\text{const}}$, and vertical-constant lookup table V_{const}

1: **Scalable-reduction step 1:** Compute the scale key N_{scale}

2: **Scalable-reduction step 2:** Use N_{scale} to select $V_{\text{const},N_{\text{scale}}}$

3: Set $N_{L/R} = m - 2$

4: Compute

$$r = N_{L/R} \bmod P_\ell, \qquad q = \frac{N_{L/R} - r}{P_\ell}$$

5: Form the left block as

$$[L_{\text{const}}]_{P_\ell \cdot q} + [L_{\text{cutoff}}]_r$$

6: Form the right block as

$$[R_{\text{cutoff}}]_{r'} + [R_{\text{const}}]_{P_\ell \cdot q}$$

where $r' = r$ unless a family-specific adjustment is required

7: Output

$$\text{Trag}([d]_m) = [L_{\text{const}}]_{P_\ell \cdot q} + [L_{\text{cutoff}}]_r + [V_{\text{const},N_{\text{scale}}}] + [R_{\text{cutoff}}]_{r'} + [R_{\text{const}}]_{P_\ell \cdot q}$$

Now, as per the scalable reduction method, sum the repdigits until arriving at a single digit. So, we add the repdigits, which will give us its digital root, denoted as

$$N_{\text{scale}} = [1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1] = 17$$
$$\rightarrow 1+7 = 8$$

This 8 serves as a key to retrieve the corresponding vertical constant; in this case, 8 → 994 in Figure 3. More detailed observations of the mappings of digital roots to vertical constants can be found in Supplementary Figure S6. While the reduction method involving the summation of large sequences of digits often presents a computational bottleneck, our approach utilizes repdigits to ensure the process remains scalable. By caching and reusing repdigit summations in discrete blocks, we significantly reduce the processing time required for large-scale numerical reduction as discussed in Section B.2. Next, let us look at the left/right horizontal pattern length, which is 9 for each side [617283950/049382716], defined as

$$P_\ell = 9.$$

We will need to pad left/right horizontal patterns, $L_{\text{const}}/R_{\text{const}}$. With a partial horizontal patterns, $L_{\text{cutoff}}/R_{\text{cutoff}}$. That is, before we append those patterns around the middle vertical constant V_{const}. To compute the length of the left/right horizontal digit blocks, we set

$$N_{L/R} = m - 2 = 17 - 2 = 15.$$

For $m=17$, this gives $N_{L/R} = 15$. Next, we calculate number of full horizontal patterns required by first calculating the length of partial patterns for $L_{\text{cutoff}}/R_{\text{cutoff}}$ by calculating remainder r,

$$r = N_{L/R} \bmod P_\ell = 15 \bmod 9 = 6$$

We determine the number of complete left/right patterns by removing the remainder r from $N_{L/R}$, so that the result is exactly divisible by the pattern length P_ℓ. Since $r = N_{L/R} \bmod P_\ell$, the quotient is

$$q = \frac{N_{L/R} - r}{P_\ell} = \frac{15-6}{9} = 1.$$

Thus, $q = 1$ gives the number of complete left/right horizontal patterns, and $r = 6$ specifies the length of the final (partial) horizontal patterns. Now it is clear how many times $L_{\text{const}}/R_{\text{const}}$ should be repeated and what length to use for $L_{\text{cutoff}}/R_{\text{cutoff}}$. A final note specific to the repdigit family $[1]^m$ is that the right cutoff length must be adjusted by one; that is, the remainder used for R_{cutoff} is $r-1$, so R_{cutoff} uses length 5 rather than 6 in this example (this adjustment is not required for the corresponding families $[2]^m$, $[4]^m$, $[7]^m$, and $[9]^m$).

Final breakdown of triangular number $[1]_{17}$ is,

$$\mathbf{Trag}\left([1]_{17}\right) = [L_{\text{const}}]_{P_\ell \cdot q} + [L_{\text{cutoff}}]_r + \left[V_{\text{const},N_{\text{scale}}}\right]_3 + [R_{\text{cutoff}}]_r + [R_{\text{const}}]_{P_\ell \cdot q} \quad (4)$$

and,

$$\mathbf{Trag}\left([1]_{17}\right) = [L_{\text{const}}]_{9\cdot 1} + [L_{\text{cutoff}}]_6 + \left[V_{\text{const},8}\right]_3 + [R_{\text{cutoff}}]_5 + [R_{\text{const}}]_{9\cdot 1} \quad (5)$$

So,

$$\begin{aligned}\text{Trag}\left([1]_{17}\right) = {} & [617283950]_9 + \left[617283\cancel{950}\right]_6 + [993]_3 \\ & + \left[\cancel{049}382716\right]_6 + [049382716]_9\end{aligned} \quad (6)$$

Finally,

$$\mathbf{Trag}\left([1]_{17}\right) = 617283950617283993827160493827 16 \quad (7)$$

Based on our analysis we can write the theorem as,

Theorem 4. *For any repdigit $[1]_m$ with m number of digits in it, the corresponding triangular number can be written as,*

$$\mathbf{Trag}\left([1]_m\right) = [L_{\text{const}}]_{P_\ell \cdot q} + [L_{\text{cutoff}}]_r + \left[V_{\text{const},N_{\text{scale}}}\right] + [R_{\text{cutoff}}]_r + [R_{\text{const}}]_{P_\ell \cdot q} \quad (8)$$

A similar strategy can be employed for m-length digits $[2]_m$. Here, for example, take $m = 18$. Supplementary **Figure S1** shows that,

$$L_{\text{const}} = 246913580, \quad R_{\text{const}} = 086419753$$

Digital roots N_{scale} mapped to their corresponding vertical constants V_{const},

$$V_{\text{const}} \in \{[253],[475],[697],[919],[141],[364],[586],[808],[030]\}$$

$$N_{\text{scale}} \in \{[\overset{\uparrow}{4}],[\overset{\uparrow}{6}],[\overset{\uparrow}{8}],[\overset{\uparrow}{1}],[\overset{\uparrow}{3}],[\overset{\uparrow}{5}],[\overset{\uparrow}{7}],[\overset{\uparrow}{9}],[\overset{\uparrow}{2}]\}$$

Also, for $m = 18$, we compute N_{scale} and its digital root, which serves as the lookup key for the corresponding vertical constant V_{const}. Since

$$N_{\text{scale}} = \underbrace{2+2+\cdots+2}_{18 \text{ terms}} = 36, \quad (N_{\text{scale}}) = (36) = 3+6 = 9,$$

the key is 9, and hence

$$9 \mapsto V_{\text{const}},\ 9 \mapsto 808\ \left(\text{i.e.}, V_{\text{const}} = 808 \text{ for key } 9\right).$$

For r and q, and with horizontal pattern length $P_\ell = 9$, we do the following,

$$N_{L/R} = m - 2 = 18 - 2 = 16,$$

$$r = N_{L/R} \bmod P_\ell = 16 \bmod 9 = 7,$$

$$q = \left(N_{L/R} - r\right) \div P_\ell = \left(16 - 7\right) \div 9 = 1$$

So, as per theorem 4, we write,

$$\begin{aligned}\mathbf{Trag}\left([2]_{18}\right) = {} & [246913580]_{9(1)} + [2469135]_7 + \left[V_{const,9}\right] \\ & + [6419753]_7 + [086419753]_{9(1)}\end{aligned} \tag{9}$$

So,

$$\mathbf{Trag}\left([2]_{18}\right) = 2469135802469135808641975308641975 3 \tag{10}$$

The triangular number $[2]_{18}$ exactly matches the one calculated using $T_n = n(n+1)/2$ the formula. The difference in both techniques is that using this *triangular number matrix system*, we are not multiplying large digits but instead writing the solution directly with patterns and constants.

Similarly, based on Supplementary **Figures S2-S5** one can write the triangular number from theorem 4 using the constants and patterns mentioned in each table.

For reproducibility and demonstration purposes, the calculator source code is available in the project repository at https://github.com/matrixcoders-io/triangular-number-matrix. An online demo version of the calculator, which will continue to receive gradual feature enhancements, is also available at https://matrixcoders.io/tnm-calculator.

As depicted in the visual calculator example, **Figure 4** shows the Triangular Number Matrix calculation together with the corresponding vertical and horizontal patterns for repdigit 1, using an incremental triangular value of 44 derived from the triangular number corresponding to the repdigit $[1]_{1000}$.

The interface renders the assembled triangular number in Pyramid view, with the vertical pattern constant $V_{\text{const},8} = 993$ at the apex and horizontal patterns $L_{\text{const}} = 617283950$ (purple) and $R_{\text{const}} = 049382716$ (amber) tiling outward symmetrically. An additive increment of 44 produces the non-repdigit value $T\left([1]_{1000}\right) + 44$; character-level fuzzy-match highlighting isolates the affected digit positions in green while undisturbed tiles retain full color. Testing with increments up to +1000 confirms that the deviation is strictly confined to the trailing digits of the left horizontal pattern, the vertical constant, and the trailing digits of the right horizontal pattern, with a symmetric correlation between the left and right boundary changes indicating a shared carry propagation rule. This boundary-confined mutation is one method being explored to extend the framework to non-repdigit triangular numbers, whereby the full deviation may be characterized as a compact correction layer over the nearest repdigit.

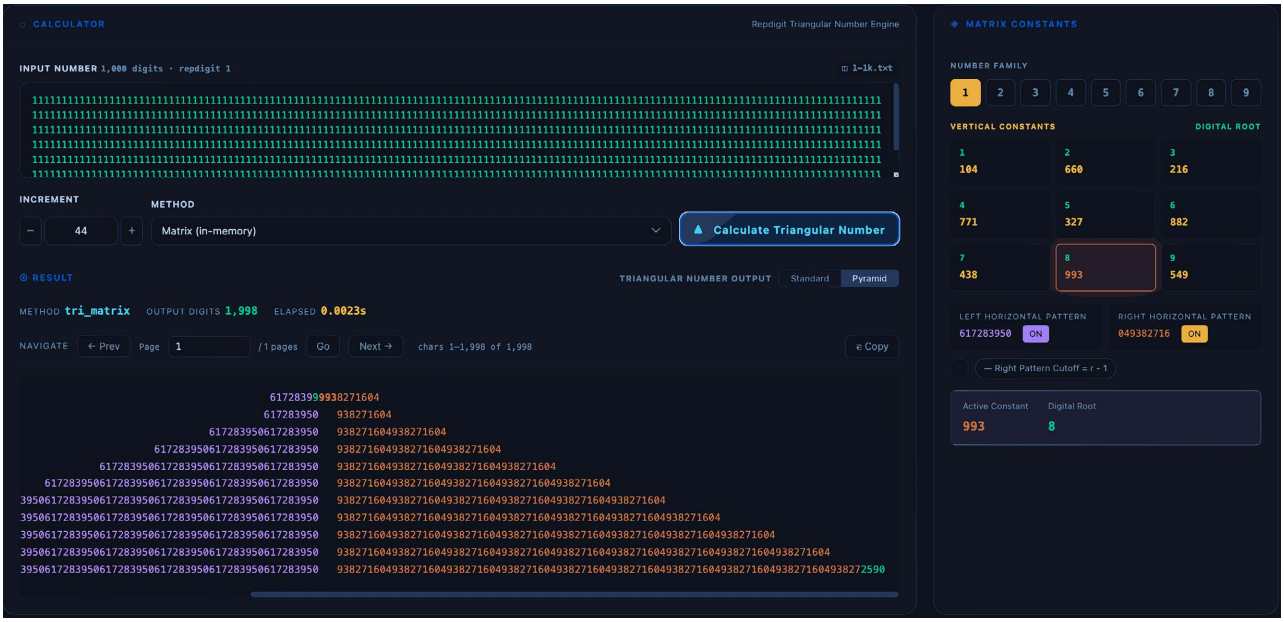

Figure 4. Triangular Number Matrix Calculator, showing the associated vertical and horizontal patterns, the repdigit length and family, color-coded changes as the incremental values vary, calculation statistics, the digital root, and related computational details.

Beyond the pattern visualization shown, the calculator provides multiple computation backends, persistent benchmarking across sessions, a pre-generated input library up to 10^{10} digits, and an integrated test suite, run history, and leaderboard; full details are available in the above mentioned repository.

4. Benchmark Evaluation of Matrix and Direct T_n Methods

To evaluate the practical performance of the proposed system, this section compares the *Direct* T_n *formula* implementations (sympy and gmpy2) against the standard *Triangular Number Matrix* method, tri_matrix. The purpose of this comparison is not to restate the mathematical derivation already established in earlier sections, but to measure how the different approaches behave as repdigit input sizes increase from millions of digits to billions of digits.

The benchmark results are organized into three parts. **Figure 4** shows a representative large-output run for the matrix method, and **Figure 5** summarizes the benchmark comparison set. Benchmark Section 1 compares sympy, gmpy2, and tri_matrix for repdigit inputs up to 100 million digits. Benchmark Section 2 extends the comparison into the billion-digit range and focuses on gmpy2 and tri_matrix, since sympy no longer remains relevant at that scale. Benchmark Section 3 introduces three additional matrix-family techniques—tri_matrix_memory, tri_matrix_stream, and tri_matrix_shortform—which were developed to address the practical difficulties of handling extremely large expanded outputs. In the benchmark tables, notation such as <5, <10, and <15 denotes measured times that remained below the indicated threshold in seconds, while N/A denotes values that were not available or were not produced under the tested conditions.

Method Family	Method Name	Comparison Group	Input / Output Long-Form (LF): Short-Form (SF):	Repdigit	Repdigit Length	Input Chars / Output Chars	File Loading Time	Repdigit Calculation Time in Seconds	Consecutive Tn Count	Total Time in Seconds
Section 1] Benchmark comparison for repdigit triangular number generation in the million digit range										
Tn formula	sympy	standard	LF/LF	1	1 million	1m/2m	<5	< 0.7	10000	< 1.1
Matrix formula	tri_matrix	standard	LF/LF	1	1 million	1m/2m	<5	< 0.003	10000	< 0.6
Direct Tn formula	sympy	standard	LF/LF	1	2 million	2m/4m	<5	< 34	10000	< 36
Matrix formula	tri_matrix	standard	LF/LF	1	2 million	2m/4m	<5	< 0.005	10000	< 2.1
Direct Tn formula	sympy	standard	LF/LF	1	5 million	5m/10m	<5	< 211	10000	< 217
Matrix formula	tri_matrix	standard	LF/LF	1	5 million	5m/10m	<5	< 0.01	10000	< 6.6
Direct Tn formula	sympy	standard	LF/LF	1	10 million	10m/20m	<5	< 838	10000	< 852
Matrix formula	tri_matrix	standard	LF/LF	1	10 million	10m/20m	<5	< 0.04	10000	< 16
Direct Tn formula	sympy	standard	LF/LF	1	100 million	100m/0	<5	N/A	10000	N/A
Matrix formula	tri_matrix	standard	LF/LF	1	100 million	100m/200m	<5	< 0.4	10000	< 167
Direct Tn formula	gmpy2	standard	LF/LF	1	100 million	100m/200m	<5	< 9	10000	< 157
Section 2] Benchmark comparison for repdigit triangular number generation in the billion digit range										
Matrix formula	tri_matrix	standard	LF/LF	1	1 billion	1b/2b	<5	< 5	0	< 5
Direct Tn formula	gmpy2	standard	LF/LF	1	1 billion	1b/2b	<5	< 119	0	> 119
Matrix formula	tri_matrix	standard	LF/LF	1	2 billion	2b/4b	<5	< 9	0	<9
Direct Tn formula	gmpy2	standard	LF/LF	1	2 billion	2b/4b	<5	< 256	0	> 256
Matrix formula	tri_matrix	standard	LF/LF	1	3 billion	3b/6b	<5	< 31	0	< 32
Direct Tn formula	gmpy2	standard	LF/LF	1	3 billion	3b/6b	<5	< 467	0	> 467
Matrix formula	tri_matrix	standard	LF/LF	1	4 billion	4b/8b	<10	< 89	0	< 89
Direct Tn formula	gmpy2	standard	LF/LF	1	4 billion	4b/8b	<10	< 963	0	> 963
Matrix formula	tri_matrix	standard	LF/LF	1	5 billion	5b/10b	<10	< 132	0	< 132
Direct Tn formula	gmpy2	standard	LF/LF	1	5 billion	5b/10b	<10	< 1208	0	> 1208
Matrix formula	tri_matrix	standard	LF/LF	1	10 billion	10b/0	<15	N/A	0	N/A
Direct Tn formula	gmpy2	standard	LF/LF	1	10 billion	10b/0	<15	N/A	0	N/A
Section 3] Advanced Triangular Number Matrix methods for high scale and window extraction purposes										
Matrix formula	tri_matrix_memory	in memory expanded output generation for transient high speed workflows	LF/LF	1	10 billion	10b/20b	<15	<9	Windows Extracted: 100000	<22
Matrix formula	tri_matrix_stream	chunked disk streamed catalog generation for memory footprint reduction and large catalog creation	LF/LF	1	10 billion	10b/20b	<15	<10	Windows Extracted: 100000	<40
Matrix formula	tri_matrix_short_form	short form descriptor based triangular number generation for reduced input output overhead	SF/SF	1	10 billion	25/<100	<0.0002	<0.2	Windows Extracted: 100000	<0.3

Figure 5. Runtime benchmarks comparing the Triangular Number Matrix method with the conventional formula $T_n = n(n+1)/2$ (implemented using sympy and gmpy2). Tests include repdigit inputs and non-repeating inputs constructed by incrementally extending repdigit-based triangular-number computations.

The first two sections show that the standard tri_matrix method maintains a strong advantage over the Direct T_n formula as input sizes increase. In Section 1, the benchmark also includes *Consecutive* T_n *Count* to show that, after calculating repdigit triangular numbers, the final non-repeating value can be extended through the additive triangular-number recurrence $T_n = T_{n-1} + n$, that is, by successive summation of consecutive natural numbers. Section 2 focuses on billion-digit comparisons and also reflects an important implementation difference: for gmpy2, *Total Time in Seconds* does not fully include the file write/output stage, since the computed big integer must still be converted back to a string and written to disk, a step that became increasingly difficult at that scale. Accordingly, the > sign for some gmpy2 totals indicates that the full end-to-end runtime would be larger once file output is completed. At the same time, the results reveal a practical boundary at very large scales: the difficulty is not only generating the triangular number itself, but also managing the memory demands of its expanded representation on limited hardware. For that reason, the third benchmark section is essen-

tial. It introduces matrix-family techniques that preserve the advantages of the Triangular Number Matrix system while adapting it to different output, storage, and access requirements.

In Benchmark Section 3, test runs do not measure output generation alone. They also include *window extraction*, meaning the retrieval of a local slice of the expanded triangular-number output by specifying a start position and an end position, or equivalently a position and length. These windows are generated randomly during the benchmark run, so the measured results reflect both number generation and localized extraction behavior rather than full-output construction alone. In the benchmark configuration, each run performs a total of 100,000 random window extractions, with extraction ranges defined in advance by index intervals such as [466,956, 467,710], [24,807,113, 24,807,238], and [48,436,213, 48,436,563]. Thus, Benchmark Section 3 evaluates not only whether the triangular-number output can be generated, but also how effectively useful local segments can be accessed during the same workflow.

The first of these techniques is tri_matrix_memory, which generates the expanded triangular number in memory without writing the result to disk. This makes it suitable for transient workflows in which the number must be produced quickly for immediate use but does not need to be stored permanently. Such a mode is useful for session-oriented generation, temporary response construction, or other in-memory operations where persistence is unnecessary.

The second is tri_matrix_stream, which is intended for cases where the expanded triangular number must be preserved as a stored artifact. Instead of holding the full output in memory, it writes the result to disk incrementally in chunks, thereby reducing memory pressure during generation. This makes it suitable for producing very large expanded catalogs that can later be accessed directly by position. In this setting, selected regions of the catalog may be retrieved through window extraction, where only a local segment is read instead of the full file. In representative access tests, the retrieved windows were randomly distributed over a large index range, with random window sizes between 100 and 200 bytes/characters and a total of 1,000,000 queries. Since 149,950,819 bytes were read overall, the average retrieved window size was approximately 149.95 bytes/characters per query. This is important because it shows that the measured access rate reflects repeated small random retrievals rather than sequential full-file scanning. The same design also suggests a broader storage role for streamed catalogs, since indexed regions may later be retrieved efficiently and, in some applications, selectively reused as structured storage locations.

The third technique is tri_matrix_shortform. This method first functions as a standalone short-form input/output system: instead of requiring a fully expanded repdigit, it accepts a compact descriptor and can return either short-form or expanded output, depending on the chosen mode. This substantially reduces the overhead associated with extremely large inputs when full expansion is unnecessary. At the same time, tri_matrix_shortform also serves as an enhancement layer

for the other matrix methods. Although benchmarked independently, the same short-form logic could later be integrated with tri_matrix_memory and tri_matrix_stream so that both methods would benefit from reduced input/output costs whenever compressed representations are acceptable.

Taken together, the three benchmark sections show that the Triangular Number Matrix system is not only fast in comparison with the Direct T_n formula, but also adaptable to multiple operational settings. The standard tri_matrix method establishes the baseline computational advantage, while tri_matrix_memory, tri_matrix_stream, and tri_matrix_shortform show how the same mathematical framework can support transient in-memory generation, persistent large-catalog construction, and compact short-form access. This flexibility is important for the later TLV discussion, where the usefulness of the method depends not only on generation speed, but also on how efficiently large triangular-number structures can be represented, stored, and selectively retrieved.

5. Multi-Base Triangular-Number Calculation and Non-Repdigit Anchors

5.1. Triangular Number of Non-Repdigits Using Shortcut Anchors

In previous sections, we discussed the ability to write the triangular number of a repdigit using constants and patterns. This raises a natural question: can the same idea be used for non-repetitive digits? The answer is yes. One approach is to use a shortcut anchor, or multiple shortcut anchors, based on a nearby repdigit in order to reduce the direct computation required.

For a lower anchor x and an offset $y \geq 0$, we use the identity

$$\mathrm{Trag}(x+y) = \mathrm{Trag}(x) + xy + \mathrm{Trag}(y),$$

where

$$\mathrm{Trag}(x) = \frac{x(x+1)}{2}.$$

Using this identity, a known triangular number can serve as an *anchor* for computing a nearby value. If x is chosen as a convenient anchor (for example, the nearest repdigit), then the triangular number of $x+y$ can be obtained by combining the anchor value $\mathrm{Trag}(x)$, the linear term xy, and the smaller triangular correction $\mathrm{Trag}(y)$. When y is small, the cost of the remaining direct computation is correspondingly reduced.

For example, if our anchor triangular number for n = 66 is 2211. What if we want to calculate the value for $n = 70$ using this novel approach? It would require the following steps: 70 − 66 = 4, hence, 2211 + (66 × 4 + **Trag**(4)) = 2211 + (264 + 10) = 2485.

In the running example, the remainder is 4. In general, if the remainder is multi-digit, we iterate the same reduction procedure until a single-digit remainder is obtained. Table 2 illustrates a case that reduces to a single-digit remainder immediately, while Table 3 shows a case that requires iteration.

Table 2. Example A: the remainder reduces to a single digit after one sub-anchor step.

Step	Example A (single-digit remainder)
1	$n = 7000$. Anchor at 6666: $7000 - 6666 = 334$. Sub-anchor at 333: $334 - 333 = 1$ (single digit).
2	$\mathbf{Trag}(334) = \mathbf{Trag}(333) + 333 \cdot 1 + \mathbf{Trag}(1) = 55611 + 333 + 1 = 55945$.
3	$\mathbf{Trag}(6666) = 22221111$, $6666 \cdot 334 = 2226444$. $\mathbf{Trag}(7000) = \mathbf{Trag}(6666) + 6666 \cdot 334 + \mathbf{Trag}(334)$ $= 22221111 + 2226444 + 55945 = 24503500.$

Table 3. Example B: the remainder is multi-digit, so the same procedure is iterated until a single-digit remainder is reached.

Step	Example B (multi-digit remainder; iterate until single digit)
1	$n = 4101$. Anchor at 3333: $4101 - 3333 = 768$ (multi-digit remainder).
2	Iterate to compute $\mathbf{Trag}(768)$: $\mathbf{Trag}(768) = \mathbf{Trag}(666) + 666 \cdot 102 + \mathbf{Trag}(102)$ $\mathbf{Trag}(102) = \mathbf{Trag}(99) + 99 \cdot 3 + \mathbf{Trag}(3) = 4950 + 297 + 6 = 5253$ $\Rightarrow \mathbf{Trag}(768) = 222111 + 67932 + 5253 = 295296$.
3	$\mathbf{Trag}(3333) = 5556111$, $3333 \cdot 768 = 2559744$. $\mathbf{Trag}(4101) = \mathbf{Trag}(3333) + 3333 \cdot 768 + \mathbf{Trag}(768)$ $= 5556111 + 2559744 + 295296 = 8411151$.

This step completes the procedure for computing triangular numbers of non-repeating digit sequences using shortcut anchor(s). The approach is amenable to parallelization: for example, in Example B ($n = 4101$), one can first enumerate the multi-digit remainders generated during the anchor reduction, compute the corresponding triangular components independently, and then aggregate the partial results as in Steps 1 - 2 of Example B. A quantitative evaluation of the runtime benefits of this parallel strategy is left for future work.

5.2. Triangular Numbers of Non-Repdigits (Multi-Base Calculator)

So far in the paper, we have learnt to effectively write the triangular number of any repetitive digit using constants and patterns. However, repdigits are not the end of the story but a step towards a new way of calculating triangular numbers. The real challenge would be to write the triangular number of any random non-repetitive digit. For the answer to this evident question, we have tried to give using our *change of base theory* in Table 4.

The premise of the idea is that every repdigit's triangular number in other base systems represents a different but also triangular number in the base 10 system.

For example, in the base-7 system, where the highest number is 6, we can easily observe these phenomena. Here are the triangular numbers in base 7 and their corresponding base-10 conversions for the number 6. Let's look at Table 4. Every repdigit of 6 in base-7 represents a non-rep triangular number in base-10. These non-repdigits in base-10, can easily be generated with a logarithm as,

Table 4. Triangular number series of repdigit number of 6 in base-7 with corresponding non-repdigits of them in base-10.

Number (base-7)	Trag number (base-7)	Number (base-10)	Trag number (base-10)
66	3300	48	1176
666	333000	342	58653
6666	33330000	2400	2881200
66666	3333300000	16806	141229221
666666	333333000000	117648	6920584776
6666666	33333330000000	823542	339111124653

- 48 next number is 342 since: 48 + 6 * 7 * 7 = 342
- 342 next number is 2400 since: 342 + 6 * 7 * 7 * 7 = 2400
- 2400 next number is 16,806 since: 2400 + 6 * 7 * 7 * 7 * 7 = 16,806

Now let's look at the triangular numbers of these non-repdigits. The triangular number of 48 in the base 10 system corresponds to 3300 in the base 7 system. We must convert the base 7 value (3300) into base 10, yielding 1176. This pattern persists; to determine the triangle number for 342, we convert the base 7 value of 333000 to base 10, yielding a triangular number of 58,653 for 342. The same applies to 2400, as evidenced by Table 4.

The base 36 system produces distinct triangular numbers using patterns that do not exist in any other base system, since the base 36 method generates different numerical sets than those produced by base 7 or base 6. In all numeral systems, we can produce unique numbers that transform into base 10 non-repeating (or repeating) digit triangular numbers. This underscores a significant point that non-repdigit values can serve as anchor points in the same manner as repdigit values in the base-10 system. For instance, the digits 48, 342, 2400, 16,806, 117,648, and 823,542 correspond to the triangular numbers 1176, 58653, 2,881,200, 141,229,221, 6,920,584,776, and 339,111,124,653, respectively. These triangular numbers, akin to repeated digits demonstrated earlier with the calculator, allow for the application of analogous reasoning.

To compute the triangle number for 49, we can utilise the known triangular number for 48, which is 1176. We only need to calculate 1176 + 49 = 1225 and continue accordingly. This practice expands a calculator to function not just with repdigits but also to utilise non-repdigit triangular numbers as anchors to stream-

line the system. Using this approach, we can speed up the calculations process since now we have more anchors than just repdigits and can more granularity access the *Triangular Matrix System* with more shortcuts.

This could be done by parallelization of the conversion process by breaking the big n-th base digit into smaller chunks, paralleling the big digit conversion, and finally adding up the sum. For example, let's take the anchor number from the table of base 6 number 333000 and break it into three chunks of 33, 00, 00, and convert these as a separate parallel task. In the base 10 system, this can be represented as:

For Chunk 1 (33 base 6):

$$= 3\times 6^1 + 3\times 6^0 = 3\times 6 + 3\times 1 = 18 + 3 = 21(\text{base } 10)$$

For Chunk 2 (30 base 6):

$$= 3\times 6^1 + 0\times 6^0 = 3\times 6 + 0 = 18 + 0 = 18(\text{base } 10)$$

For Chunk 3 (00 base 6):

$$= 0\times 6^1 + 0\times 6^0 = 0 + 0 = 0(\text{base } 10)$$

And the final formula is:

$$21\times 6^4 + 18\times 6^2 + 0\times 6^0 = 21\times 1296 + 18\times 36 + 0 = 27864$$

Multi-Base Anchors The following example demonstrates how a multi-base framework can be used to locate the closest anchor point for a non-repdigit number. Figure 6 visualizes this comparison.

Base of Anchor Point	Multi-Base Anchor Points	Converted Base-10 Anchor Points	Distance from 5760	Multi-Base Triangular Number	10-Base Triangular Number
CLOSER BASES – with the same length of rep-digits anchors can be generated from different bases.					
10	6666	6666	+906	22221111	22221111
9	8888	6560	+800	44440000	21520080
11	4444	5856	+96	975357a	17149296
9	7777	5740	-20	34002661	16476670
12	3333	5655	-105	54329b0	15992340
10	5555	5555	-205	15431790	15431790
FURTHER BASES – with different length of rep-digits anchors can generated from different bases.					
7	22222	5602	-158	250253053	15694003
6	44444	6220	+460	1530402514	19347310
3	22222222	6560	+800	1111111100000000	21520080

Figure 6. Multi-base anchors for locating the closest repdigit anchor point to a target non-repdigit number in base 10. The target value 5760 is highlighted in red, the closest anchor is highlighted in green, and the corresponding base-10 anchors are highlighted in blue.

Figure 6 illustrates the process of locating the closest anchor point for the target number **5760**. In base 10, this number lies between the anchor points **5555** and **6666**. However, by extending the search to repdigit anchors from nearby bases—such as base-**9**: **8888**, base-**9**: **7777**, base-**11**: **4444**, and base-**12**: **3333**—it is possible to identify anchors that are even closer when converted into base 10.

This makes it possible to refine the search for the nearest anchor point to **5760** by comparing the corresponding base-10 values obtained from repdigits in different bases. In Figure 6, the closest anchor point is **5740**, which gives a distance of **−20** (**5740** − **5760** = **−20**). This is the smallest absolute distance shown in the table, and it is obtained from the base-**9** repdigit anchor after conversion into base 10.

Table 5. Base-4 patterns as base-10 triangular anchors.

[4 Base]	[10 Base]
1320 = 33	15 = 120
133200 = 333	63 = 2016
13332000 = 3333	255 = 32640
1333320000 = 33333	1023 = 523776
133333200000 = 333333	4095 = 8386560
13333332000000 = 3333333	16383 = 134209536
1333333320000000 = 33333333	65535 = 2147450880
133333333200000000 = 333333333	262143 = 34359607296
13333333332000000000 = 3333333333	1048575 = 549755289600

Table 5 illustrates how a repdigit triangular-number pattern in base 4 can convert into base-10 non-repdigit anchors and their corresponding triangular numbers. To test the reliability of this process, bases 3 through 36 were examined to determine whether, for each ascending repdigit, the corresponding triangular number—after conversion to base 10—would remain a triangular number in base 10. For each base, the first 100 ascending repdigits were tested for every valid digit (for example, in base 2, digits 1 and 2 were each tested across their first 100 repdigits). The results showed that the corresponding triangular number of every tested repdigit remained triangular in base 10, supporting the reliability of the process illustrated in Figure 6.

Note: A remaining challenge is to make conversion between base systems significantly faster and to implement a software module that improves conversion speed. At present, this paper does not claim high-speed computation for non-repdigit numbers; the demonstrated speed improvements apply to repdigits only, which is sufficient for the cryptographic use case developed in this work.

6. Triangular Number Matrix to Cryptography

6.1. From Triangular Patterns to Catalog-Style Objects

Sections 2 - 5 establish the *Triangular Number Matrix* as a constructive system for

generating large triangular numbers from digit-level regularities, rather than by direct evaluation of the conventional formula $T_n = n(n+1)/2$. The central observation is that, for structured index families—most notably repdigits—the corresponding triangular numbers exhibit stable internal regularities that persist as digit-length grows. This enables an *assembly-first* view of computation: once the relevant constants and scaling rules are identified for a family, increasing the requested length does not require increasingly expensive large-integer multiplication and division, but rather the repeated application of the same pattern rules.

This view yields a protocol-relevant abstraction. Instead of treating each T_n purely as a scalar, Sections 2 - 5 implicitly define a collection of deterministic *digit objects* indexed by compact parameters. Concretely, fixing a base b and a repdigit family (or more generally, a pattern identifier), each admissible length ℓ determines a specific base-b digit string representing a triangular number instance generated by the matrix rules. In Section 7, this abstraction is formalized as *catalogs across bases*; Section 6 clarifies why the mathematical construction naturally admits such a catalog interpretation and why that interpretation is relevant under modern cybersecurity threats.

6.2. Theorems 1 - 4 as Central Catalog-Generating Results

A key ingredient for any catalog-style construction is that members can be generated reproducibly from short descriptors. In this work, Theorems 1 - 4 provide precisely the family-level determinism needed for that role.

- **Theorems 1 - 3 (closed-form repdigit families).** These theorems give explicit digit-structure formulas for repdigits based on 3, 6, and 9, each parameterized by the repdigit length m. The resulting forms specify how the triangular number representation grows with m and therefore define a direct mapping from a short descriptor (family, length) to a unique digit expansion.
- **Theorem 4 (general repdigit assembly via scalable reduction).** Theorem 4 generalizes the construction to repdigits whose triangular numbers are assembled from: 1) horizontal constants $(L_{\text{const}}, R_{\text{const}})$ that repeat around 2) a vertical constant selected by a scale-dependent key. In the notation used throughout the paper, the theorem can be summarized as a structured concatenation:

$$\text{Trag}\left([1]^m\right) = [L_{\text{const}}]^{P_\ell \cdot q} \,||\, [L_{\text{cutoff}}]^r \,||\, \left[V_{\text{const},N_{\text{scale}}}\right] \,||\, [R_{\text{cutoff}}]^r \,||\, [R_{\text{const}}]^{P_\ell \cdot q}, \quad (11)$$

where P_ℓ is the horizontal pattern length, $N_{L/R} = m-2$, $r = N_{L/R} \bmod P_\ell$, and $q = (N_{L/R} - r)/P_\ell$. Here $||$ denotes concatenation, and the bracketed repetition notation follows the conventions already established in Sections 2 - 3.

Together, Theorems 1 - 4 supply the mathematical backbone for catalog construction: they define families of digit objects whose members are determined by compact parameters (family identifier and length) and are generated by rule-driven assembly. This catalog viewpoint does not introduce new mathematics; it is a re-expression of the main results in Sections 2 - 3 in a form that is directly consumable by cryptographic protocol design (formalized later in Section 7).

6.3. Scalable Reduction as Compact Addressing and Reproducible Expansion

A second requirement for protocol use is that large objects be addressable without transmitting or computing the full expansion. The scalable reduction method (Section 3) provides this property in two complementary ways.

Compact addressing.

The scalable reduction method computes a scale-dependent key (e.g., via a digital-root style reduction) using only a small set of parameters such as the repdigit value and the repdigit length. This key is then used to select the appropriate vertical constant from V_{const}. Because the key computation depends on a compact description rather than on enumerating all digits, the selection step remains feasible even when the repdigit length is extremely large.

Reproducible expansion by assembly.

Once the vertical constant is selected, the remainder of the expansion is determined by repeating the horizontal constants and applying the appropriate cutoffs according to q and r (cf. Equation (11)). The expansion procedure is therefore deterministic and repeatable: the same descriptor yields the same digit object, and the generation process avoids full big-integer multiplication and division in favor of pattern concatenation.

This compact-address plus deterministic-expansion structure is the mathematical precursor to the short-form representations and catalog commitments introduced in Section 7. The present section does not formalize those protocol mechanisms; rather, it motivates why they arise naturally from the constructive content of Sections 2 - 5.

6.4. Multi-Base Construction and Why It Enlarges the Adversarial Search Space

Section 5 extends the matrix framework beyond base-10 repdigits by leveraging the change-of-base perspective. Two points are central for the cryptographic motivation:

1) ***Multi-base generation increases the pool of structured instances.***

Repdigits in other bases yield triangular-number representations that, when converted to base 10, correspond to distinct triangular numbers associated with non-repdigit values (Section 5). This produces a larger library of structured triangular-number instances, which can serve as additional reference points and can be generated efficiently as base-dependent patterns (e.g., the base-7 series for the digit 6 and its corresponding conversions).

2) ***Multi-base processing supports parallelizable workflows.***

Section 5 demonstrates that base conversion itself can be parallelized by chunking a large base-b number into smaller blocks, converting each block independently, and combining the results via powers of b. This property aligns with the general computational objective of decomposing large-digit operations into schedulable parallel tasks.

3) ***Multi-base representations expand the parameter space an attacker must consider.***

In the later protocol framing, operations can be expressed over multiple bases and multiple catalog families. Even when catalogs are public, multi-base diversity increases the number of plausible representations and families that may underlie an observed computation. Informally, a larger set of bases increases the combinatorial surface of admissible constructions (base choice × family choice × length choice), which expands the space of hypotheses an adversary must consider when attempting to reverse-engineer or exhaustively search over protocol-relevant parameters. **Figure 6** illustrates this concretely in the computational setting by showing that closer anchor points can be obtained by checking repdigits in nearby bases and converting back to base 10; the same multi-base diversity also underpins the protocol design surface considered in Section 7.

6.5. CyberSecurity: What Problem Are We Trying to Solve?

The preceding sections establish a mathematical engine for producing large, structured, base-dependent triangular digit objects from compact descriptors. This motivates a natural question: *why should such digit objects be considered in cryptographic design at all?* Beyond purely computational interest, the motivation is driven by practical authentication risks: large-scale credential theft and password database breaches enable offline guessing attacks that can lead to unauthorized access even when online rate limiting is effective. Section 7 therefore provides the security context for using structured, sliceable digit objects as a protocol substrate and motivates the rolling-state, layered authentication construction formalized in the same section, which is designed to mitigate the use of stolen credentials and reduce unauthorized access. In addition, Section 6 situates this direction within the broader long-horizon threat landscape, including the prospect of cryptographically relevant quantum computing and the corresponding pressure toward post-quantum security assumptions.

As advances in highly parallel hardware continue to reduce the cost of large-scale computation, cryptographic designs that rely solely on "doing the math faster" face growing pressure—both from commodity GPU acceleration and from longer-term quantum risk to widely deployed public-key primitives. A cryptographically relevant quantum computer would, in principle, undermine RSA and elliptic-curve systems via quantum algorithms, and this prospect has already motivated standardization and migration efforts in post-quantum cryptography (PQC), including NIST's finalized PQC standards (e.g., ML-KEM for key establishment) and vendor deployments that begin protecting data in transit against "harvest now, decrypt later" (HNDL) threats. In this context, the present work frames an alternative computational emphasis as *assembly versus compute*: instead of depending primarily on a small set of algebraic problems aligned with massive parallel arithmetic, the construction explored here emphasizes structured

composition—using Triangular Number Matrix patterns and base-dependent representations—so that recovering the message resembles reconstructing a constrained jigsaw (a combinatorial assembly problem) rather than optimizing a single dominant mathematical operation.

6.6. Concluding Remarks and Transition to the TLV Construction

Taken together, Theorems 1 - 4 provide deterministic pattern families that can be instantiated at scale from compact descriptors, while the multi-base processes in Section 5 enlarge the universe of available structured instances beyond base-10 repdigits. Section 7 formalizes how these families are treated as public catalogs across bases and how compact descriptors enable efficient access to structured digit material within those catalogs. The aim is not to replace standardized post-quantum public-key primitives, but to articulate a complementary design surface in which the dominant operation is structured composition over many base-dependent catalog instances—consistent with the "assembly versus compute" framing introduced above—and to present a proof-of-concept authentication construction (TLV) that leverages this structure under modern breach and offline-attack considerations.

7. Application in Cryptography: Triangular Layered Verification (TLV)

Until now, we have learned about how fast it is to calculate a triangular number using the matrix method, without directly invoking the $T_n = n(n+1)/2$ formula. Now, we would like to provide a detailed application of this method in cryptography, which we refer to as Triangular Layered Verification (TLV). Before discussing TLV, we would like to provide a brief overview of the short form [7].

7.1. Change of Base Rule in Triangular Number Calculation

Every last digit's triangular number in any base has a simple pattern that is easily reducible or can be represented in a short form as (base = 4, number = 3, length = 5). For example, (base = 4, number = 3, length = 5) would be equal to reducing 133333200000 for the expansion of number 3, that is written in base 4, and is the 5th triangular number represented as 8,386,560 in base 10, see chart below 4. That's why we can represent it as (base = 4, length = 5) which could be easily assembled back into its original form of 133333200000 based on the calculator rules presented earlier.

This pattern is not limited to just base 4, but extends to every base. For example, in Base 3, the number 2 being the highest digit, number 22's triangular number is 1100; however, we will display an expanded pattern at 20th length, like 11111111111111111111 00000000000000000000 to highlight correlations as the pattern grows in length. Similarly, we can write a pattern for different base systems as in **Table 6**.

Table 6. Expanded triangular-number patterns for the highest digit in each base.

Base	Representation
3	111111111111111111111111**0**000000000000000000000000
4	**1**333333333333333333333333**2**000000000000000000000000
5	222222222222222222222222**0**000000000000000000000000
6	**2**555555555555555555555555**3**000000000000000000000000
7	333333333333333333333333333**0**000000000000000000000000
8	**3**777777777777777777777777774**0**000000000000000000000000
…	…

7.2. Triangular Layered Verification (TLV)

TLV proposes a rolling state interactive authentication mechanism to protect against dumping passwords during a data breach when an attacker takes all the data necessary to do an offline brute force attack to crack passwords in the database. The idea is to periodically change user passwords automatically and seamlessly within a defined date threshold (ie, user password/token is older than 2 days). The authentication system precomputes, in memory, the per-layer op templates and their bounded *arg-tags* from public parameters and the committed catalogs; after the client sends the *session-nonce*, the server transiently returns this bundle to start state assembly. These *arg-tags/ops* are never persisted server-side and are discarded after the attempt; each new *session-nonce* yields fresh ops and *arg-tags*. We consider a set of committed catalogs across bases: each catalog (b, n) maps legal lengths. Below, we define the key terminology in TLV mechanisms.

1) **Per-layer op templates**: server-precomputed operation descriptors for each layer (schema: $b, n, l, L_{\text{pos}}, R_{\text{pos}}, D_{\text{len}}$) derived from public parameters and the committed catalogs, defining exactly which digit slices are read and mixed.

2) **Bounded *arg-tags***: server-computed index sets selecting a bounded subset of those op templates per layer; "bounded" ensures indices are in-range for the corresponding templates (no out-of-bounds positions); conveyed transiently and never persisted server-side.

3) ***Session-nonce***: a server-generated, high-entropy per-session nonce that the client echoes back to initiate binding and state assembly for the attempt; each new *session-nonce* triggers fresh op templates and *arg-tags* that are passed to the user once.

This builds "private user rules" and periodically syncs with legitimate users to refresh the password/token if the defined date threshold is exceeded. A legitimate user passes the "private user rules" to the server in order to undergo a layered interactive authentication process and gain access to the protected systems. The next section discusses the strategy used for TLV cryptography.

7.3. Layer and Channel Structure in TLV

To make the TLV architecture precise, we distinguish between *operation tem-*

plates, *layers*, and *channels*. This clarification separates the core interactive authentication construction from higher-level configurable protocol profiles.

Operation templates.

An operation template is a bounded catalog-derived descriptor of the form

$$\mathrm{op} = \left(b, n, \ell, L_{\mathrm{pos}}, R_{\mathrm{pos}}, D_{\mathrm{len}}\right),$$

where $\left(b, n, \ell\right)$ identifies a legal catalog member and $\left(L_{\mathrm{pos}}, R_{\mathrm{pos}}, D_{\mathrm{len}}\right)$ determines the left and right windows used in the local rip-and-mix computation. In TLV, these operation templates are precomputed by the server from public parameters and the committed catalogs, and are paired with bounded *arg-tags* that identify which templates are active for a given attempt.

Layer.

A *layer* is a bounded ordered collection of operation templates together with its associated *arg-tags* and folding rule. Intuitively, a layer represents one verification unit in which only the operations selected by the transmitted *arg-tags* are evaluated, while the remaining templates in that layer are ignored for that attempt. The outputs of the selected operations are then combined to form a single layer commitment or layer digest. Thus, a layer is not a single operation, but a bounded sequence of eligible operations from which only a transiently selected subset is activated in a given session.

Channel.

A *channel* is a use-case-specific verification profile composed of one or more layers, together with the rule by which their outputs are combined and accepted. In this paper, the primary fully described channel is the *authentication channel*. Other channels are best viewed as configurable extensions built on the same TLV structure rather than as separate cryptographic systems.

Authentication channel.

In the authentication setting, the server first precomputes the per-layer operation templates and bounded *arg-tags* from the committed catalogs and public parameters, then transiently returns these values after receiving the session nonce. The client uses the received private user rules to assemble the required per-layer responses and returns the resulting verification material. The verifier evaluates only the selected operations in each layer, forms the corresponding layer digests, and then folds these layer outputs into the final commitment checked against the stored authentication state. In this sense, the authentication channel is a layered verification profile in which each layer contributes a bounded and selectively activated portion of the overall proof.

Why layering is useful.

This structure serves two purposes. First, it keeps honest verification efficient because only a bounded subset of operations is processed per layer. Second, it increases attacker uncertainty in the offline setting because an adversary must reconstruct not only the underlying catalog-derived operations, but also the correct per-layer selection pattern across multiple layers, while learning only the final verification outcome and not any per-layer oracle. Accordingly, the role of layering

in TLV is not merely organizational; it is part of the mechanism by which selective verification, rolling state, and bounded catalog access are combined into one authentication workflow.

Scope.

For the purposes of this paper, the term *channel* should therefore be read narrowly: it denotes an application-specific arrangement of TLV layers. The authentication channel is the main proof-of-concept construction developed here. Other possible channels, such as authorization, document access, or distributed transaction verification, are more appropriately presented as configurable TLV channel profiles based on the same layer abstraction (see Section B.3).

To make this architecture more tangible, Figure 7 illustrates the conceptual composition of TLV layers and shows how selected operations contribute to the final commitment used by the verifier. Each layer contains a bounded set of operation templates together with arg-tags that select which operations are evaluated in a given session. The selected operations produce per-layer digests (H_1, H_2, $\cdots$), and the sequence of layer digests is folded to produce a final commitment H_{final} that is compared with the stored verifier state during authentication.

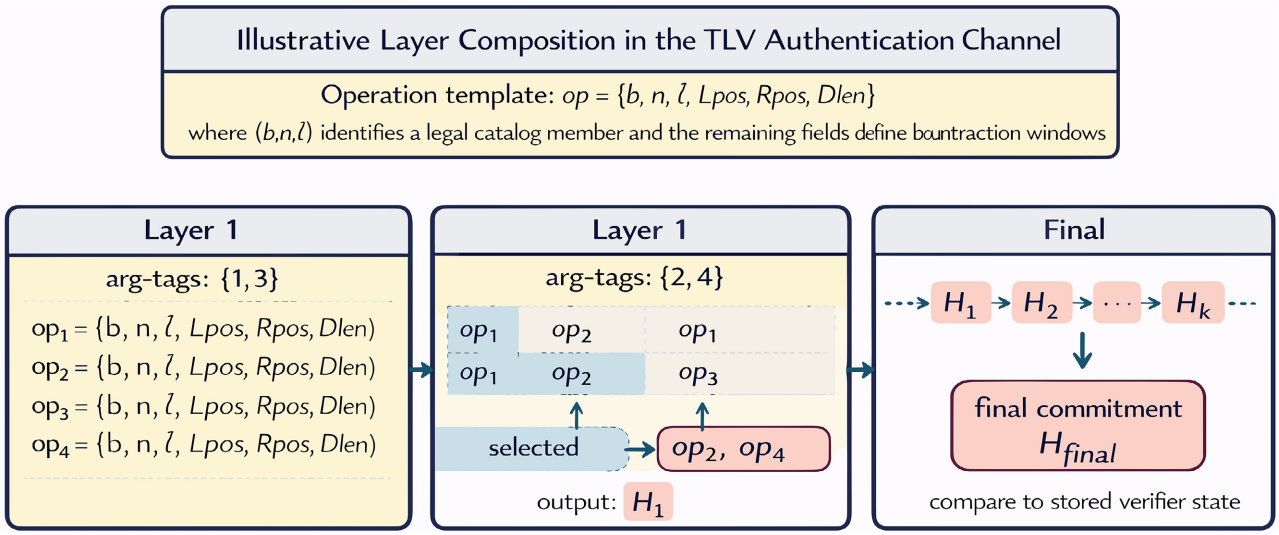

Figure 7. Illustrative TLV layer composition. Each layer contains a bounded set of operation templates of the form $(b, n, \ell, L_{\text{pos}}, R_{\text{pos}}, D_{\text{len}})$ together with arg-tags that select which operations are evaluated in a given session. The selected operations produce a layer digest (H_1, H_2, $\cdots$), and the sequence of layer digests is folded to produce a final commitment H_{final} that is compared with the stored verifier state during authentication.

7.4. Rip-and-Mix Windows for Per-Layer Computation

7.4.1. Catalog and op Model

- **Catalog.** For each base b and pattern id n, the catalog defines a *digit language* that yields, for any legal $\ell \in S_{b,n}$, a base-b digit string

$$D = D_{b,n,\ell}, \quad \ell \in \{0, \cdots, b-1\}^N, \quad N = |D|.$$

Implementations expose random access to *slices* of D (by index and length)

without materializing all N digits.

- **Op tuple.** Each operation is

$$\text{op} = \left(b, n, \ell, L_{\text{pos}}, R_{\text{pos}}, D_{\text{len}}\right),$$

with the following constraints:

- b base; n pattern id; $\ell \in S_{b,n}$ selects the specific member; together (b, n, ℓ) determine D and N.
- **Left window** start L_{pos} is 0-based from the left; **Right window** offset R_{pos} is 0-based from the right.
- **Window length** $D_{\text{len}} \geq 1$.
- **Bounds:**

$$\begin{aligned} &\text{Left}: \quad L_{\text{pos}} \geq 0,\ D_{\text{len}} \geq 0,\ L_{\text{pos}} + D_{\text{len}} \leq c, \\ &\text{Right}: \quad R_{\text{pos}} \geq 0,\ D_{\text{len}} \geq 0,\ R_{\text{pos}} + D_{\text{len}} \leq \text{len}(D) - c, \end{aligned}$$

where c is the center.

7.4.2. Window Extraction (No Full Expansion)

Given D and N, define the two windows purely by indices:

- **Left window:**

$$L\text{digits} = D\left[c - \left(L_{\text{pos}} + D_{\text{len}}\right) : c - L_{\text{pos}}\right]$$

- **Right window:**

$$R\text{digits} = D\left[c + R_{\text{pos}} : c + R_{\text{pos}} + D_{\text{len}}\right]$$

Both slices are obtained by the catalog's *slice API*; the rest of D is never expanded.

Catalog's slice API = a local function/method that returns digits $D[a:b]$ *without expanding all of* D (e.g., generator or table lookup). No external calls.

7.4.3. Converting Only the Windows

We avoid hashing a billion digits by converting only the tiny slices to decimal. Two equivalent choices (pick one for the spec):

- **Local base-b value (simple and fast):**

$$\text{val}_L = \sum_{j=0}^{D_{\text{len}}-1} L\text{digits}[j] \cdot b^{D_{\text{len}}-1-j}, \quad \text{val}_R = \sum_{j=0}^{D_{\text{len}}-1} R\text{digits}[j] \cdot b^{D_{\text{len}}-1-j}.$$

Encode each as a fixed-width decimal string (zero-padded to $\lceil D_{\text{len}} \log_{10} b \rceil$ digits).

- **Position-aware snippet (still tiny, binds to position)**

Choose a small modulus 10^t (e.g., $t = 2 \cdot D_{\text{len}}$). Let the global position of the j-th digit be P_j (0 = most significant). Compute

$$\text{valL} \equiv \sum_{j=0}^{D_{\text{len}}-1} b^{P_j} \cdot L\text{digits}[j] \pmod{10^t}, \quad \text{valR} \equiv \sum_{j=0}^{D_{\text{len}}-1} b^{P_j} \cdot R\text{digits}[j] \pmod{10^t},$$

using fast modular powers. Emit the last t decimal digits of each.

(*This captures "considering their position" without heavy big-int work.*)

Rip-and-Mix Token

Normalize the two decimal snippets to equal width and **interleave** them (zip one digit from left, one from right). Optionally permute the interleave with a per-layer salt S_i to prevent reuse:

$$\text{Lstr} = \text{zpad}\left(\text{dec}\left(\text{valL}\right), W\right)$$

$$\text{Rstr} = \text{zpad}\left(\text{dec}\left(\text{valR}\right), W\right)$$

$$\text{mix} = \text{Interleave}\left(\text{Lstr}, \text{Rstr}\right) \quad \#\text{orInterleavePerm}\left(\text{Lstr}, \text{Rstr}, S_i\right)$$

Define the op's *rip-and-mix token*:

$$t\left(op\right) = H\left(\text{"RipMix"} \| b \| n \| \ell \| Lpos \| Rpos \| Dlen \| mix \| S_i\right),$$

where H is a 256-bit hash (e.g., SHA-256/SHA3-256) and S_i is a per-layer salt such as

$$S_i = H\left(\text{"Salt"} \| E \| session\_nonce \| state_{i-1}\right).$$

per-layer index sets supplied by the client that tell the verifier which few operations to evaluate (via a RAM-bound tag) before forming the layer digest. The goal is to keep honest logins fast while forcing offline attackers to search a larger hidden space (witness and indices) with no per-layer oracle. Tag's scratch is deterministically derived from S_i.

Example (tiny numbers for illustration)

Let the catalog define $D_{6,3,5}$ as the 12-digit base-6 string:

33333 000000

- op $= \left(b = 6, n = 3, \ell = 5, L_{\text{pos}} = 4, R_{\text{pos}} = 3, D_{\text{len}} = 3\right)$
- Ldigits $= D\left[4:7\right] = 3,0,0 \rightarrow$ local decimal

$$\text{valL} = 3 \cdot 6^2 + 0 \cdot 6 + 0 = 108 \rightarrow \text{"108"}.$$

- Rdigits $= D\left[12-3-3:12-3\right] = D\left[6:9\right] = 0,0,0 \rightarrow$

$$\text{valR} = 0 \rightarrow \text{"000"}.$$

- Interleave: "108" $\perp$ "000" $\rightarrow$ "100800".
- Token:

$$t = H\left(\text{"RipMix"} \| \text{params} \| \text{"100800"} \| S_i\right).$$

You compute such tokens **only** for indices in arg_tags; the rest of the layer is unaffected.

7.5. Security and Performance Notes

- **No full expansion.** All work is on $O\left(D_{\text{len}}\right)$ digits per selected op; typical D_{len} is tiny (e.g., 3 - 16). Hash cost is fixed and small.
- **Client-held indices add search space.** Offline attackers must guess $\left(b, n, \ell, L_{\text{pos}}, R_{\text{pos}}, D_{\text{len}}\right)$ and the per-layer arg_tags_i across K layers, with *no per-layer oracle*—only the final commitment decides.
- **Salts prevent precomputation.** Including S_i in $t\left(\cdot\right)$ and T_i kills table-reuse across sessions/epochs.

- **Catalog rotation.** If catalogs change over epochs, you can bind a version V into S_i or the fold; old offline transcripts won't verify under a new V.
- **Bounds enforcement.** Reject ops whose windows fall out of range; do so in constant time and without per-layer leakage (fixed-length transcripts, padded timing).

7.6. Threat-Model Alignment

- **Server breach (but not the client's secrets).** The attacker sees only C_E (only final epoch commitment) and system parameters. To test a guess, they must run the *same* selective RAM-walks for the (unknown) per-layer indices and still only learn success/failure at the *final* check.
- **Catalog leakage.** Harmless: catalogs are public by design. Strength comes from the client-held indices and salted RAM-walks, and mathematical operations, not catalog secrecy.

7.7. Triangular vs. Other Polygonal Families (Why Triangles Win for "Assemble, Don't Compute")

Scope. By "polygonal" we refer to the standard figurate families:

- **Triangular:** $T_n = \frac{n(n+1)}{2}$.
- **Square:** $S_n = n^2$.
- **Pentagonal/Hexagonal (general *k*-gonal):** $P_k(n) = 2(k-2)n^2 - (k-4)n$.

We compare these families specifically for our *catalog + rip-and-mix windows model*, where an operation is $(b, n, \ell, L_{\text{pos}}, R_{\text{pos}}, D_{\text{len}})$ and we only ever extract tiny slices of the base-b digit string $D_{b,n,\ell}$ rather than expanding the full number.

1) Catalog Density & Short Forms Across Bases

- **Triangular:** Empirically yields the richest, dense "digit languages" across many bases (e.g., families of repeaters/structured blocks) that map to valid triangular indices for long stretches of ℓ. This yields large legal sets $S_{b,n}$ and many usable short descriptors per base. Multiple bases can be used to further increase richness.
- **Square:** Productive in a few bases (e.g., patterns that mirror squaring rules), but sparser than triangles; legal length sets $S_{b,n}$ tend to be thinner for compact patterns.
- **Pentagonal/hexagonal/general *k*-gonal:** Catalogs exist, but short-form patterns are rarer, more brittle, and less reusable across bases.

Why it matters: Denser catalogs $\Rightarrow$ more valid operations per layer without decimal work, more options for the client, and a larger hidden search space for the attacker.

2) Sliceability & Local Evaluators (Fast Windows)

- **Triangular:** Many patterns allow fetching slices $D[a:b]$ with minimal evaluators (no full expansion). L/R window extraction is cheap and uniform across

bases.

- **Square:** Local evaluation is heavier due to carry interactions from squaring; feasible but less uniform.
- **Higher *k*-gonal:** Often needs more per-digit logic or partial expansions to locate digits, slowing window extraction.

Why it matters: A rip-and-mix token touches only $O(D_{\text{len}})$ digits; triangular patterns keep this path simple and predictable.

3) Security Fit to Rip-and-Mix

- All families share the same core security principle: hardness derives from client-held choices (*witness*, *arg_tags*, *windows* $L_{\text{pos}}, R_{\text{pos}}, D_{\text{len}}$) and final-only commitment.
- **Triangles provide an edge:** their dense catalogs offer more valid operations (larger m_i) and more cross-base variants, expanding the attacker's offline search space for the same per-guess cost.

4) Operational Cost & Caching

- **Triangular:** Best cache reuse (catalog-wide, salt-agnostic) for window locators and tiny digit evaluators; highly predictable performance.
- **Square/others:** More custom evaluators; higher risk of fallback to heavy arithmetic during window extraction.
- **Performance knob:** Per-guess RAM and slice counts can be tuned. Total cost is via modular exponentiation—CPU-friendly, highly parallel, and *not* RAM-bound unless configured to be.

8. Conclusions

To conclude our findings, we have proposed an alternative and direct way of writing the triangular number of any given repetitive digit of number n and length m, $\left([N]_m\right)$, based on *the triangular number matrix* formula, without even doing any large multiplication and summation of digits. The formula involves constants and patterns hidden inside the long triangular numbers. This beautiful approach to writing triangular numbers opens a new direction for utilizing hidden symmetries and patterns, thereby avoiding direct, large mathematical operations.

Moreover, we presented *TLV*, a breach-resilient verifier that uses public triangular digit catalogs, rip-and-mix window tokens, and client-held arg_tags to keep honest work tiny while forcing offline attackers to pay a large, memory-bound cost per guess, with success revealed only at a final commitment.

Triangles excel in this setting because their catalogs are unusually dense and sliceable across bases, enabling fast, symmetric, seam-free implementations that never need to expand full numbers unless the security model explicitly demands it. Security stems from per-layer salts, selective RAM-walks, and hidden index sets—not from catalog secrecy.

Acknowledgements

This work was motivated in part by the author's longstanding fascination with

repdigits, numerical structure, and pattern formation in mathematics. The author also gratefully acknowledges Trishul Dhalia for support and assistance in preparing the manuscript.

Data Availability Statement

The data generated for this study are available within the text.

Conflicts of Interest

The author reports no conflict of interest.

References

[1] Cormen, T.H., Leiserson, C.E., Rivest, R.L. and Stein, C. (2022) Introduction to Algorithms. MIT Press. https://mitpress.mit.edu/9780262367509/introduction-to-algorithms/

[2] Karatsuba, A. (1963) Multiplication of Multidigit Numbers on Automata. *Soviet Physics Doklady*, **7**, 595-596. https://ui.adsabs.harvard.edu/abs/1963SPhD....7..595K

[3] Schönhage, A. and Strassen, V. (1971) Fast Multiplication of Large Numbers. *Computing*, **7**, 281-292. https://doi.org/10.1007/bf02242355

[4] Kessler, G.C. (2003) An Overview of Cryptography. https://garykessler.net/library/crypto.html

[5] Buchmann, J. (2004) Introduction to Cryptography, Vol. 335. Springer. https://link.springer.com/book/10.1007/978-1-4684-0496-8

[6] Qadir, A.M. and Varol, N. (2019) A Review Paper on Cryptography. 2019 7*th International Symposium on Digital Forensics and Security* (*ISDFS*), Barcelos, 10-12 June 2019, 1-6. https://doi.org/10.1109/isdfs.2019.8757514

[7] Cachin, C. and Chandran, N. (2009) A Secure Cryptographic Token Interface. 2009 22*nd IEEE Computer Security Foundations Symposium*, Port Jefferson, 8-10 July 2009, 141-153. https://doi.org/10.1109/csf.2009.7

Supplementary

A. Figures for Repdigit of 2, 4, 5, 7 and 8

Here, we present the constants and patterns in repdigits of 2, 4, 5, 7, and 8, along with the mapping from digital roots to vertical constants, in the following Supplementary Figures S1-S6.

TRIANGULAR NUMBER	VERTICAL PATTERNS		
	CONSTANTS	DIFFERENCE	DIGITS INHERITED
[3]			
[253]	[253]	223	
[24753]	[475]	222	[2,5,3]
[2469753]	[697]	222	[4,7,5]
[246919753]	[919]	222	[6,9,7]
[24691419753]	1[141]	222	[9,1,9]
[2469136419753]	[364]	223	[1,4,1]
[246913586419753]	[586]	222	[3,6,4]
[24691358086419753]	[808]	222	[5,8,6]
[246913580386419753]	1[030]	222	[8,0,8]
[2469135802530086419753]	[253]	223	[0,3,0]
[246913580247530086419753]	[475]	222	[2,5,3]
[2469135802469753086419753]	[697]	222	[4,7,5]
[246913580246919753086419753]	[919]	222	[6,9,7]
[24691358024691419753086419753]	1[141]	222	[9,1,9]
[246913580246913641975308641975 3086419753]	HORIZONTAL PATTERNS		
[2469135802469135864197530864197 53086419753]	LEFT		RIGHT
[246913580246913580308641975308641975 3086419753]	[246913580]		[086419753]

Figure S1. Triangular series of repdigit (2).

TRIANGULAR NUMBER	VERTICAL PATTERNS		
	CONSTANTS	DIFFERENCE	DIGITS INHERITED
[28]			
[3003]	1[003]	223	
[302253]	[225]	222	[0,3]
[30244753]	[447]	222	[2,5]
[3024669753]	[669]	222	[4,7]
[302468919753]	[891]	222	[6,9]
[3024691141975 3086419753]	1[114]	223	[9,1]
[302469133641975 3086419753]	[336]	222	[1,4]
[30246913558641975 3086419753]	[558]	222	[3,6]
[3024691357808641975 3086419753]	[780]	222	[5,8]
[302469135800308641975 3086419753]	1[003]	223	[8,0]
[3024691358022530864197 53086419753]	[225]	222	[0,3]
[302469135802447530864197 53086419753]	[447]	222	[2,5]
[3024691358024669753086419753]	[669]	222	[4,7]
[3024691358024689197530864197 53]	[891]	222	[6,9]
[3024691358024691141975308641975 3]	HORIZONTAL PATTERNS		
[3024691358024691336419753086419753]			
[3024691358024691355864197530864197 53]	LEFT		RIGHT
[3024691358024691357808641975308641975 3]3	[024691358]		[086419753]

Figure S2. Triangular series of repdigit (7).

TRIANGULAR NUMBER	VERTICAL PATTERNS		
	CONSTANTS	DIFFERENCE	DIGITS INHERITED
[15]			
[1540]	[540]	-111	
[154290]	[429]	-111	[5,4,0]
[15431790]	[317]	-112	[4,9]
[1543206790]	[206]	-111	[2,7]
[154320956790]	1[095]	-111	[2,0,6]
[15432098456790]	[984]	-111	[0,9,5]
[15432098734567 90]	[873]	-111	[9,8,4]
[154320987623456790]	[762]	-111	[8,7,3]
[15432098765123456790]	[651]	-111	[7,6,2]
[154320987654 0123456790]	[540]	-111	[6,5,1]
[1543209876542 90123456790]	[429]	-111	[5,4,0]
[154320987654317 90123456790]	[317]	-112	[4,9]
[15432098765432067 90123456790]	[206]	-111	[3,7]
[154320987654320956790123456790]	1[095]	-111	[2,0,6]
[1543209876543209845679 0123456790]	HORIZONTAL PATTERNS		
[1543209876543209873456790123456790]			
[154320987654320987623456790123456790]	LEFT		RIGHT
[15432098765432098765123456790123456790]	1[543209876]		[123456790]

Figure S3. Triangular series of repdigit (5).

TRIANGULAR NUMBER	VERTICAL PATTERNS		
	CONSTANTS	DIFFERENCE	DIGITS INHERITED
[36]			
[3916]	[916]	556	
[394716]	1[471]	555	[9,1,6]
[39502716]	1[027]	556	[7,1]
[3950582716]	[582]	555	[0,2,7]
[395061382716]	1[138]	556	[8,2]
[3950616938 2716]	[693]	555	[1,3,8]
[395061724 9382716]	1[249]	556	[9,3]
[39506172804 9382716]	[804]	555	[2,4,9]
[3950617283604 9382716]	1[360]	556	[8,0,4]
[395061728391604 9382716]	[916]	556	[3,6,0]
[3950617283947160 49382716]	1[471]	555	[9,1,6]
[3950617283950271 6049382716]	1[027]	556	[7,1]
[3950617283950582 716049382716]	[582]	555	[0,2,7]
[3950617283950613827 16049382716]	1[138]	556	[8,2]
[3950617283950616938 2716049382716]	HORIZONTAL PATTERNS		
[3950617283950617249 382716049382716]			
[3950617283950617280 49382716049382716]	LEFT		RIGHT
[3950617283950617283604 9382716049382716]	[395061728]		[049382716]

Figure S4. Triangular series of repdigit (8).

TRIANGULAR NUMBER	VERTICAL PATTERNS		
	CONSTANTS	DIFFERENCE	DIGITS INHERITED
[10]			
[990]	[990]	-111	
[9**8790**]	[879]	-111	[9,9,0]
[98**767**90]	[767]	-111	[8,7,9]
[987**656**790]	[656]	-111	[7,6,7]
[9876**545**6790]	[545]	-111	[6,5,6]
[98765**434**56790]	[434]	-111	[5,4,5]
[9876**5432**3456790]	[323]	-111	[4,3,4]
[98765**212**3456790]	[212]	-111	[3,2,3]
[9876543**101**23456790]	[101]	-111	[2,1,2]
[987654320**990**123456790]	[990]	-111	[0,1]
[9876543209**879**0123456790]	[879]	-111	[9,9,0]
[98765432098**767**90123456790]	[767]	-111	[8,7,9]
[987654320987**656**790123456790]	[656]	-111	[7,6,7]
[9876543209876**545**6790123456790]	[545]	-111	[6,5,6]
[98765432098765**434**56790123456790]	HORIZONTAL PATTERNS		
[987654320987654**323**456790123456790]			
[9876543209876543**212**3456790123456790]	LEFT		RIGHT
[98765432098765432**101**23456790123456790]	[876543209]		[123456790]

Figure S5. Triangular series of repdigit (4).

REP DIGIT LENGTH	CALCULATE DIGITAL ROOT OF REP DIGIT	DIGITAL ROOT TO VERTICAL CONSTANT	REP DIGIT TRIANGULAR NUMBER - VERTICAL CONSTANT COLOR CODED IN RED
1	2	2--->030	[3]
2	2+2=4	4--->253	[253]
3	2+2+2=6	6--->475	[2475**3**]
4	2+2+2+2=8	8--->697	[246**97**53]
5	2+2+2+2+2=10=1+0=1	1--->919	[246**919**753]
6	2+2+2+2+2+2=12=1+2=3	3--->141	[2469**141**9753]
7	2+2+2+2+2+2+2=14=1+4=5	5--->364	[24691**364**19753]
8	2+2+2+2+2+2+2+2=16=1+6=7	7--->586	[2469135**86**419753]
9	2+2+2+2+2+2+2+2+2=18=1+8=9	9--->808	[246913**5808**6419753]
10	2+2+2+2+2+2+2+2+2+2=20=2+0=2	2-->030	[24691358**030**86419753]
11	2+2+2+2+2+2+2+2+2+2+2=22=2+2=4	4--->253	[2469135802**53**086419753]
12	2+2+2+2+2+2+2+2+2+2+2+2=24=2+4=6	6--->475	[246913580**2475**3086419753]
13	2+2+2+2+2+2+2+2+2+2+2+2+2=26=2+6=8	8--->697	[24691358024**697**53086419753]
14	2+2+2+2+2+2+2+2+2+2+2+2+2+2=28=10=1+0=1	1--->919	[246913580246**919**753086419753]
15	2+2+2+2+2+2+2+2+2+2+2+2+2+2+2=30=3+0=3	3--->141	[2469135802469**141**9753086419753]
16	2+2+2+2+2+2+2+2+2+2+2+2+2+2+2+2=32=3+2=5	5--->364	[24691358024691**364**19753086419753]
17	2+2+2+2+2+2+2+2+2+2+2+2+2+2+2+2+2=34=3+4=7	7--->586	[246913580246913**586**419753086419753]
18	2+2+2+2+2+2+2+2+2+2+2+2+2+2+2+2+2+2=36=3+6=9	9--->808	[2469135802469135**808**6419753086419753]
19	2+2+2+2+2+2+2+2+2+2+2+2+2+2+2+2+2+2+2=38=3+8=11=1+1=2	2-->030	[24691358024691358**030**86419753086419753]

Figure S6. Mapping from digital roots to the corresponding vertical constants for repdigit (2).

B. Future Directions

B.1. Parallelization Enhancements

Both the multi-base method and the GPU-porting approach rely on dividing work into independent tasks that can be executed simultaneously. Instead of generating each triangular number via the direct formula $T_n = n(n+1)/2$, we precompute anchors across bases and assemble numbers in parallel. Key strategies include:

1) **Base-level parallelization:** employ multiple numeral systems (e.g., base 7

and base 10) concurrently to compute repdigit anchors and their triangular numbers; each base yields its own sequence (e.g., 3300, 333000, 33330000 in base 7).

2) **Task-level parallelization:** assign independent jobs to disjoint ranges of repdigit or non-repdigit inputs—one process might handle values starting at 3300 while another begins from 33333333330000000000—so anchors and subsequent additions can be generated concurrently.

3) **Pattern-iteration parallelization:** treat each pattern length (for example 3300, 333000 or 33330000) as a separate work unit that iterates the additive sequence $(1+2+3+\cdots)$ in parallel.

4) **Anchor search on GPUs:** use digit-length heuristics and binary search operations to locate the nearest repdigit anchor for a given input in a massively parallel fashion on GPU hardware.

5) **Independent pattern assembly:** construct the left and right pattern blocks separately on GPU threads, allowing simultaneous assembly of both halves of the triangular number.

6) **Parallel reduction:** break the digital-root reduction into hundreds or thousands of small computations across the GPU to obtain the scale-dependent key quickly.

7) **Block-wise base conversion:** split large base-b numbers into smaller chunks, convert each chunk independently on the GPU and reassemble them to perform base changes efficiently.

8) **Parallel non-repdigit processing:** apply logarithmic or block-based techniques to non-repdigit triangular numbers so they can be processed alongside repdigits without serial bottlenecks.

Overall, the objective is to decompose the computation of $T_n = n(n+1)/2$ into schedulable parallel tasks, including string assembly, anchor generation, reduction, and other bounded arithmetic operations that can be distributed across GPU threads. In this way, work that would traditionally be performed through a largely serial large-integer pipeline can instead be reorganized into smaller independent components better suited to multi-threaded GPU architectures. With careful implementation, triangular-number generation may therefore be expressed as a parallel workload in which non-dependent stages are executed concurrently to improve throughput and reduce computation time.

B.2. Logarithmic Enhancements

1) Auto-generating vertical constants from inherited digits (Figure 2): This approach could generate vertical constants automatically, eliminating the need to manually identify them in each base and enabling dynamic formation of these constants for pattern assembly in large-base systems (e.g., base 1000). Since the composition of the horizontal patterns is also linked to the vertical patterns, it may be possible to discover a logarithmic characterization for these relationships as well.

2) Deriving the remainder via the pattern index: This approach can reduce computation time compared with computing the remainder directly as

$r = N_{L/R} \bmod P_\ell$. The scalable reduction method yields the digital root of a large repdigit, which is mapped to a pattern index drawn from the nine repeating values $\{1,2,3,4,5,6,7,8,9\}$ (see **Figure 3**). The remainder could then be obtained by subtracting 1 from pattern index directly without additional computations.

3) Speeding up scalable reduction via caching: The scalable reduction step dominates the runtime of triangular-number generation, so accelerating it can substantially improve overall performance. In addition to parallelization (e.g., multithreading or GPUs), the repdigit setting enables an effective caching strategy keyed by digit and length. This caching approach has already been implemented, and the benchmark results reported in this paper reflect the corresponding increase in speed. For example, to compute the digital root of the repdigit $N = [2]^{140000000}$, it is unnecessary to iterate over 140,000,000 identical digits. Instead, one can cache the contribution of fixed blocks of repeated 2's—for instance $B_k = [2]^k$ for selected lengths k (e.g., $k \in \{10^3, 10^6, 10^7\}$)—and then express 140,000,000 as a sum of these cached block lengths (e.g., 140,000,000 = 10 × 14,000,000). The digital root is then obtained by combining the cached block contributions rather than summing digits individually. This optimization relies on the uniform structure of repdigits and is generally not feasible for non-repeating digit strings.

B.3. Configurable TLV Channels for Future Work

1) Authorization channel: A natural extension of the authentication channel is an authorization channel. In this setting, TLV would not only confirm that a client has established a valid session, but would also verify that the client is permitted to perform a specific protected action. Accordingly, a TLV authorization channel is best interpreted as a second verification profile layered on top of successful authentication, where the final layer commitment is bound to operation-specific or resource-specific access conditions rather than to identity alone.

2) Document access channel: A second future direction is a document access channel. Here, the layered verification process would gate retrieval of protected content, decryption material, or indexed ranges within expanded catalogs generated by methods such as tri_matrix_stream. This is a natural fit for the catalog-oriented structure developed in this paper, since TLV already supports bounded windows and selective local extraction rather than requiring full-object processing in every step. In such a setting, a successful channel verification could authorize access to a specific document segment, key fragment, or retrieval range within a stored catalog-backed object. The most realistic interpretation is therefore controlled document access and selective retrieval built on catalog storage, rather than treating TLV itself as a general-purpose document storage system.

3) Distributed transaction verification: A third future direction is a distributed transaction verification profile for blockchain or ledger-linked systems. The intended role here is not consensus replacement, but synchronized verification of transaction-related state, private payload access, or approval logic across distributed participants. In this setting, the TLV channel would act as a structured veri-

fication layer whose bounded catalog accesses and folded per-layer commitments are shared across a distributed workflow. This is especially relevant in architectures where subsets of participants require selective visibility or selective validation of protected state, since TLV naturally supports bounded local extraction and final-only acceptance.

Research direction: These examples suggest that the main reusable object in TLV is not a single authentication procedure, but a configurable layered verification profile. Future work may therefore formalize a family of TLV channels in which the same catalog and layer abstractions are retained, while the acceptance rule, protected object, and state-rotation policy are specialized to the intended application.

Towards the end of this paper, we leave with this final statement:

There is a symmetric structure in which every repdigit's triangular number in any base also corresponds to a different triangular number in another base, as demonstrated. This symmetry arises from repdigits, which are themselves symmetric. We hope that these pattern-assembly methods may inspire innovations across multiple scientific domains.