PALISADE

HOMOMOPHIC ENCRYPTION FOR PALISADE USERS: TUTORIAL WITH APPLICATIONS

Applications of Homomorphic Encryption over Integers

Dr. David Bruce Cousins, Director Duality Labs

dcousins@dualiltytech.com

AGENDA

- Role of integers in encrypted applications
- Basic Examples from the PALISADE distribution
- Encrypted SubString Search Application

Role of Integers in Encrypted Applications

A quick review

PALISADE Supports Schemes with Three Classes of Encrypted Operations

- Boolean operations with unlimited depth — Previous lecture
  - FHEW, TFHE
- Integer operations with limited depth — This lecture
  - BGV, BFV, and their RNS variants
  - RNS → Residue Number System – breaks rings of large bit-width integers into a parallel set of rings using < 64 bit residues, allowing very efficient computation on 64-bit CPU architectures
- Approximate "floating point" with limited depth — Next lecture
  - CKKS

Which Scheme You Use Will Depend on the Form of Your Data

- Typical applications well suited for integer schemes:
  - Strings of Characters are just integers
  - Private information retrieval using Integer ID fields
  - Private set intersection [privately joining encrypted data sets based on common fields]
- Before CKKS was available many HE applications used a block scaling approach to approximate floating-point arithmetic.
  - Multiply all inputs by a large constant
    - 3.1415 x 10000 = 31415
  - Requires numerical analysis of the problem to determine how big the scale factor should be
    - Affects roundoff error, saturation error
    - Need to keep track of increase of scale during multiplies.
      - a x 10000 x b x 10000 = c x 100000000 etc…
- Don't do this anymore – use CKKS instead!

Limitations on Integer Homomorphic Encrypted Operations

- Some common Integer software operations cannot be done easily with HE
  - Examples are division, comparison
- Often, we need to recast our problem in order to craft a HE solution
  - An example we will see today is determining if two encrypted numbers are equal
- Packed encoding allows us to take advantage of SIMD (Single Instruction Multiple Data) operations that can provide a great efficiency
  - However not all problems map to this structure well
  - SIMD comes with complexity – efficient code can be difficult to understand

Basic Integer Examples

From the PALISADE distribution

C++ Examples of Integer Operations Provided in the PALISADE release

- Sample executables are in public key encryption area **${root}/build/bin/examples/pke**
- C++ source code for these examples are in **${root}/src/pke/examples**
    - **depth-*** : examples of variations on performing chained multiplication for BGVrns, BFVrns and BFVrns-b.
    - **simple-integers** : examples of addition, multiplication and rotation using packed vector encoding for BFVrns [we reviewed this in the first part of this webinar]
    - **simple-integers-bgvrns** : same for BGVrns
    - **simple-integers-serial*** : how to serialize (save to disk) the components of a crypto-system (various keys and ciphertext) for BGVrns and BFVrns
- Source code for sample benchmarks are in **${root}/benchmark/src/compare-bfvrns-vs*.cpp**

Encrypted SubString Search Application

From the PALISADE integer examples repository

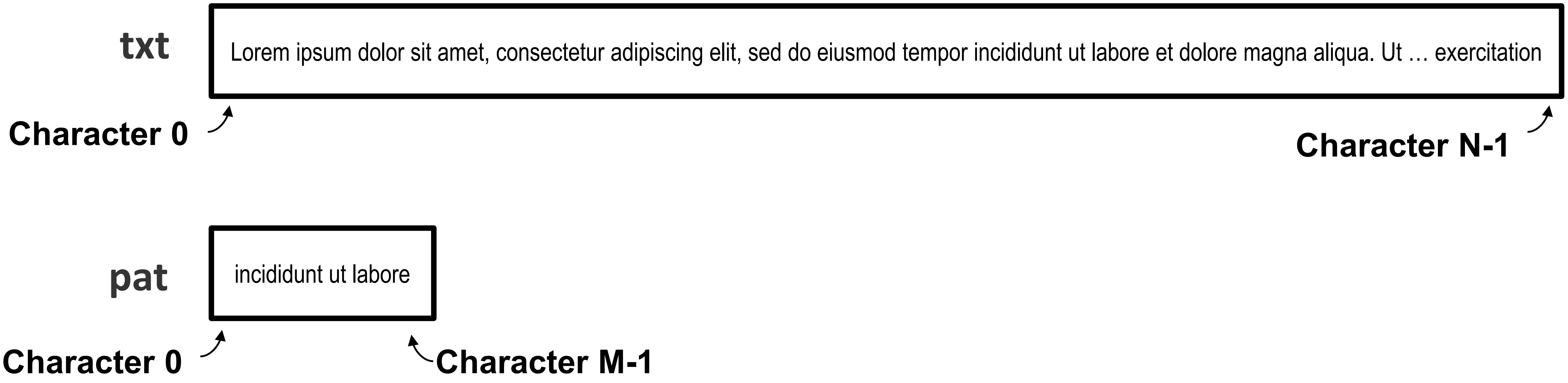
PALISADE Encrypted SubString Search

- GitLab repo: **https://gitlab.com/palisade/palisade-integer-examples**
  - Build instructions are in README.md, requires you to install PALISADE development edition
- Contains prototype C++ code that shows a BFVrns application:
  - **strsearch_enc_1.cpp:** Perform a plaintext string search (no wildcards) using the Rabin-Karp method modified for homomorphic encrypted computation, and compare with an encrypted version, one character per ciphertext
  - **strsearch_enc_2.cpp:** The same algorithm but searching on a very large text by using packed vectors of character per ciphertext

PALISADE

Plaintext SubString Search

- search for substring **pat** of length M in string **txt** of length N



PALISADE

Plaintext SubString Search Algorithms

- Brute force substring search for substring **pat** of length M in string **txt** of length N
- For each offset **i** into **txt**
  - For each offset **j** into **pat**
    - Compare **pat[j] == txt[j+i]**
    - Return true if all comparisons for this value of **i** are true.
- Lots of repeated comparisons.

- There are more efficient approaches that use a **rolling hash**
  - The rolling hash of **pat** is computed once.
  - The rolling hash of substring of **txt** at offset i=0, length M is computed once
  - The hashes are compared for equality (if == then substring matches)
  - For each offset into **txt** the rolling hash is update by removing one old character and adding one new character.

PALISADE

Plaintext Rolling hash - initialization

- Calculation of initial hash for a given modulus **p**
- Note **d** is the size of the alphabet (here 256 bits or one **char**)

```
int ph = 0;  // hash value for pattern
int th = 0; // hash value for txt
int h = 1;



// The value of h would be "pow(d, M-1)%p"
for (i = 0; i < M-1; i++) {
  h = (h*d)%p;
}
// Calculate the hash value of pattern and first window of text
for (i = 0; i < M; i++) {
  ph = (d * ph + pat[i]) % p;
  th = (d * th + txt[i]) % p;
}
```

Plaintext Rolling hash – update (sliding through **txt**)

- Update of txt hash for new offset

```
// Calculate hash value for next window of text: Remove leading digit,
// add trailing digit
if ( i < N - M ) {
  th = (d * (th - txt[i] * h) + txt[i + M]) % p;

  // We might get negative value of t, converting it to positive
  if (th < 0) {
    th = (th + p);
  }
```

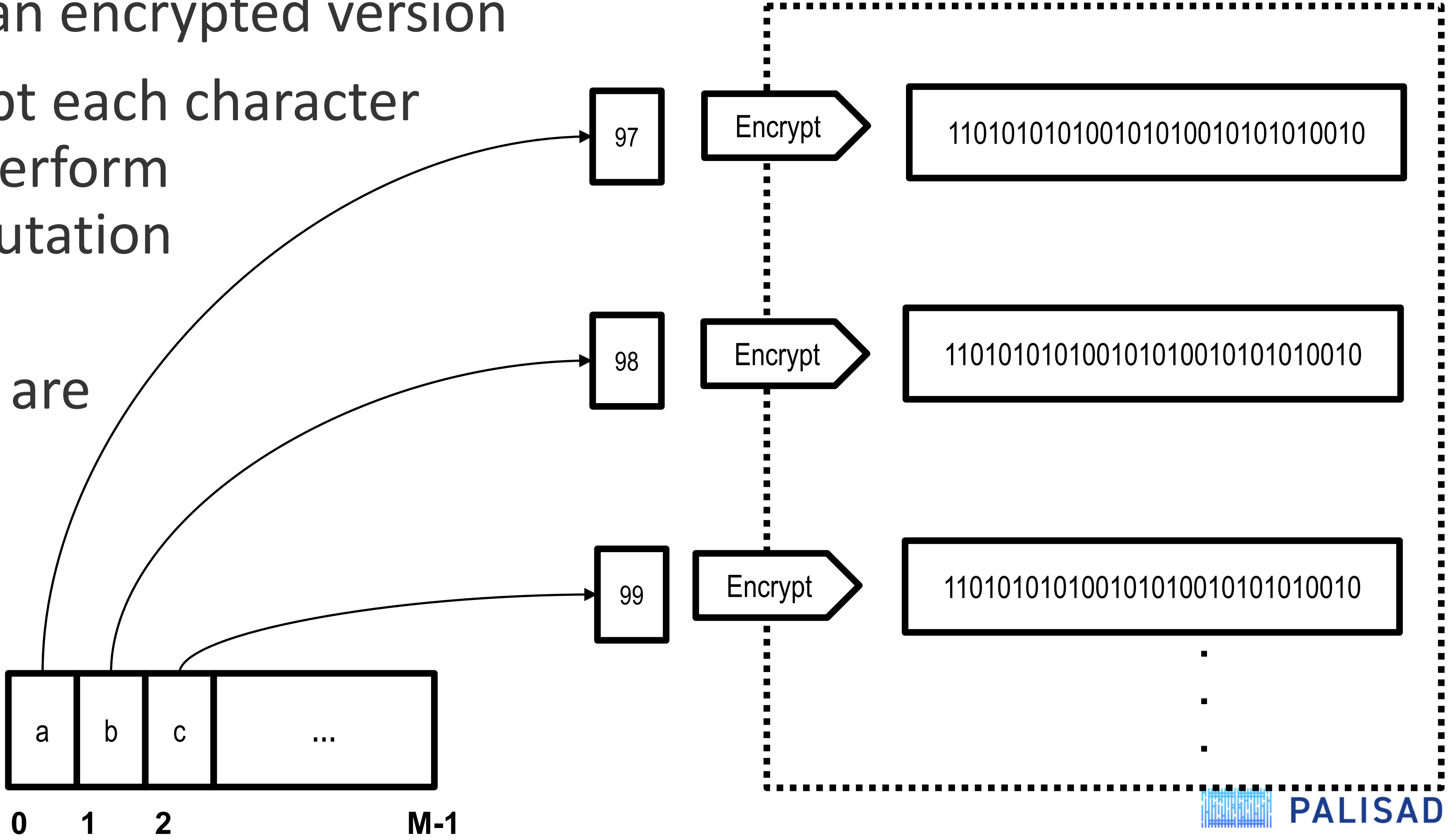
- We see there is one multiply by **h** each update, and one multiplication by **d**
- Since d is a power of 2, we can replace it with a set of repeated additions to save on multiplicative depth

Plaintext Rolling Hash - Converting to Encrypted Form

- This version of the Rabin-Karp rolling hash was selected because it was amenable to integer HE operations:
  - No divisions only +,- and *
  - Math was performed modulo some large prime
  - Limited multiplications (one per character of **pat**, and one for each update into **txt**)
  - Multiplication by a power of 2 constant (can be implemented with a binary tree of additions in order to reduce multiplicative depth)
  - Hash comparisons can be done with encrypted subtraction. Zero values correspond to ==
  - We support mod of negative numbers so we've found that the test is not necessary.

PALISADE Encrypted Substring Search V1

- We implement our plaintext strings as vector<char> vs std::string to simplify writing an encrypted version
- Version 1, encrypt each character separately and perform encrypted computation in PALISADE
- Both **txt** and **pat** are encrypted into a vector of ciphertexts



PALISADE Encrypted Rolling hash

- Calculation of initial encrypted hash for a given plaintext modulus **p**

```
CT phct = encrypt_repeated_integer(cc, pk, 0, nrep);  // hash value for patter

CT thct = encrypt_repeated_integer(cc, pk, 0, nrep);  // hash value for txt



// The value of h would be "pow(d, M-1)%p"
long h = 1;
for (i = 0; i < M-1; i++) {
  h = (h*d)%p;
}
CT hct = encrypt_repeated_integer(cc, pk, h, nrep);  // encrypted h

DEBUG("encrypting first hashes" );
// Calculate the hash value of pattern and first window of text
for (i = 0; i < M; i++) {
  auto tmp = encMultD(cc, phct);
  phct = cc->EvalAdd(tmp, epat[i]);

  tmp = encMultD(cc, thct);
  thct = cc->EvalAdd(tmp, etxt[i]);
}
```

Two helper functions defined on next slide

PALISADE Encrypted Rolling hash - Initialize

- Two helper functions are defined:

```
CT encrypt_repeated_integer(CryptoContext<DCRTPoly> &cc, LPPublicKey<DCRTPoly> &pk,  int64_t in, size_t n){

  vecInt v_in(0);
  for (auto i = 0; i < n; i++){
    v_in.push_back(in);
  }
  PT pt= cc->MakePackedPlaintext(v_in);
  CT ct = cc->Encrypt(pk, pt);

  return ct;
}
```

Packs an integer **in** into a packed encrypted vector by duplicating it **n** times.

Note that while we could have used a single integer encoding, we will use this function to optimize the algorithm in the next version

```
CT encMultD(CryptoContext<DCRTPoly> &cc, CT in){
   auto tmp(in);
   for (auto i = 0; i< 8; i++ ){
     tmp = cc->EvalAdd(tmp, tmp);
   }

   return(tmp);
```

Multiplies by 256 via binary tree of repeated addition

Note: typically noise growth due to addition is very small vs multiplication, but here we are adding a ciphertext with *itself multiple times*, so the noise grows faster than adding independent ciphertexts. The growth is not as fast as multiplication of two ciphertexts but use this approach with caution.

PALISADE Encrypted Rolling hash - Update

- Update of encrypted txt hash for new offset

```
vecCT eres(0);
// Slide the pattern over text one by one
DEBUG("sliding" );
for (i = 0; i <= N - M; i++) {
  cout<<i<< '\r'<<flush;

  // Check the hash values of current window of text and pattern
  // If the hash values match then only check for characters on by one
  // subtract the two hashes, zero is equality
  DEBUG("sub" );
  eres.push_back(cc->EvalSub(phct, thct));

  // Calculate hash value for next window of text: Remove leading digit,
  // add trailing digit
  if ( i < N - M ) {
    DEBUG("rehash" );
    //th = (d * (th - txt[i] * h) + txt[i + M]) % p;

    auto tmp = encMultD(cc,
                    cc->EvalSub(thct,
                                cc->EvalMult(etxt[i], hct)
                                )
                    );
    thct = cc->EvalAdd(tmp, etxt[i+M] );
```

Comparison result stored in **eres**

Update rolling hash

PALISADE

Output Processing

- We decrypt the output **encres**. Any zero entries indicate the hashes match
- Since these are hashes, there is a very small probability of a hash collision, so the result should be considered a "highly likely match"
- If we were concerned with leaking any information about the encrypted **pat** or **txt**, we could multiply each entry in **encres** by an encrypted random number which then randomizes the non-zero entries

PALISADE Encrypted SubString Search V2 – SIMD processing

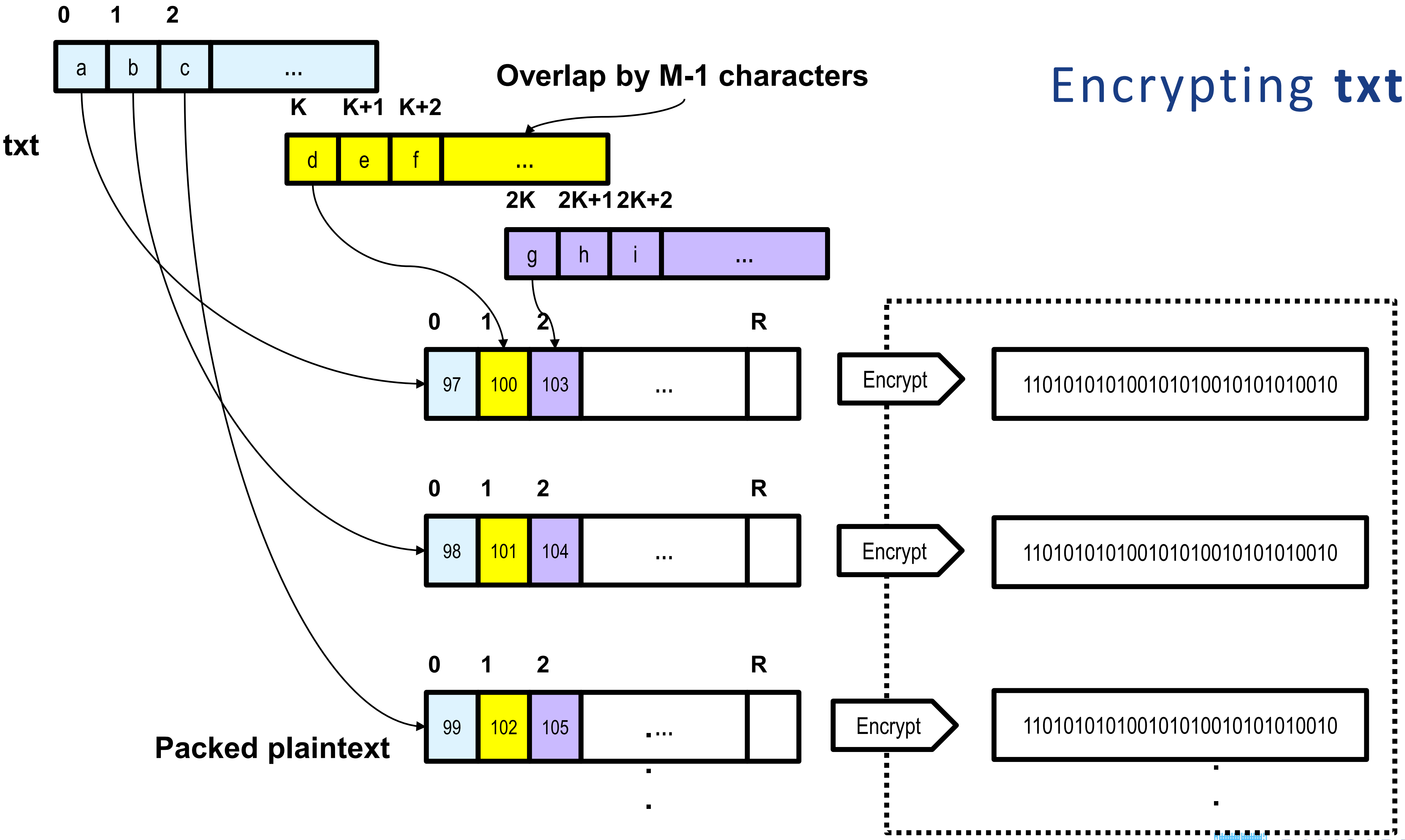
- The major limitation of V1 is that it is not very efficient.
  - We need a ciphertext for each character in **txt**
  - We need multiplicative depth equal to length of **txt**
  - **Both** limit the practical size of **txt** that can be searched
- **Solution:** use packed encoding of vectors and SIMD operations
- **Strategy**: Slice the text into batches, and pack them into Encrypted vectors to enable SIMD searching of each batch in parallel
  - If we pack the ciphertexts right, we can use the same code to do R=ring-size comparisons in parallel. Remember R is O(32k→64k)

PALISADE

Encrypting **txt**

- We vectorize **txt** in batches, so each vector has every **K**th character
- We then create a packed plaintext of each vector and encrypt it, resulting in **L** ciphertexts (note **L** is approx. **N**/ring-size, but must be adjusted to account for overlap *and* must be > **M** so we can still generate full hashes for comparison)
- Choose **K** and **L** to provide an overlap in the batches so that there are no gaps in searching for **pat** in the batched **txt**

Encrypting txt



PALISADE

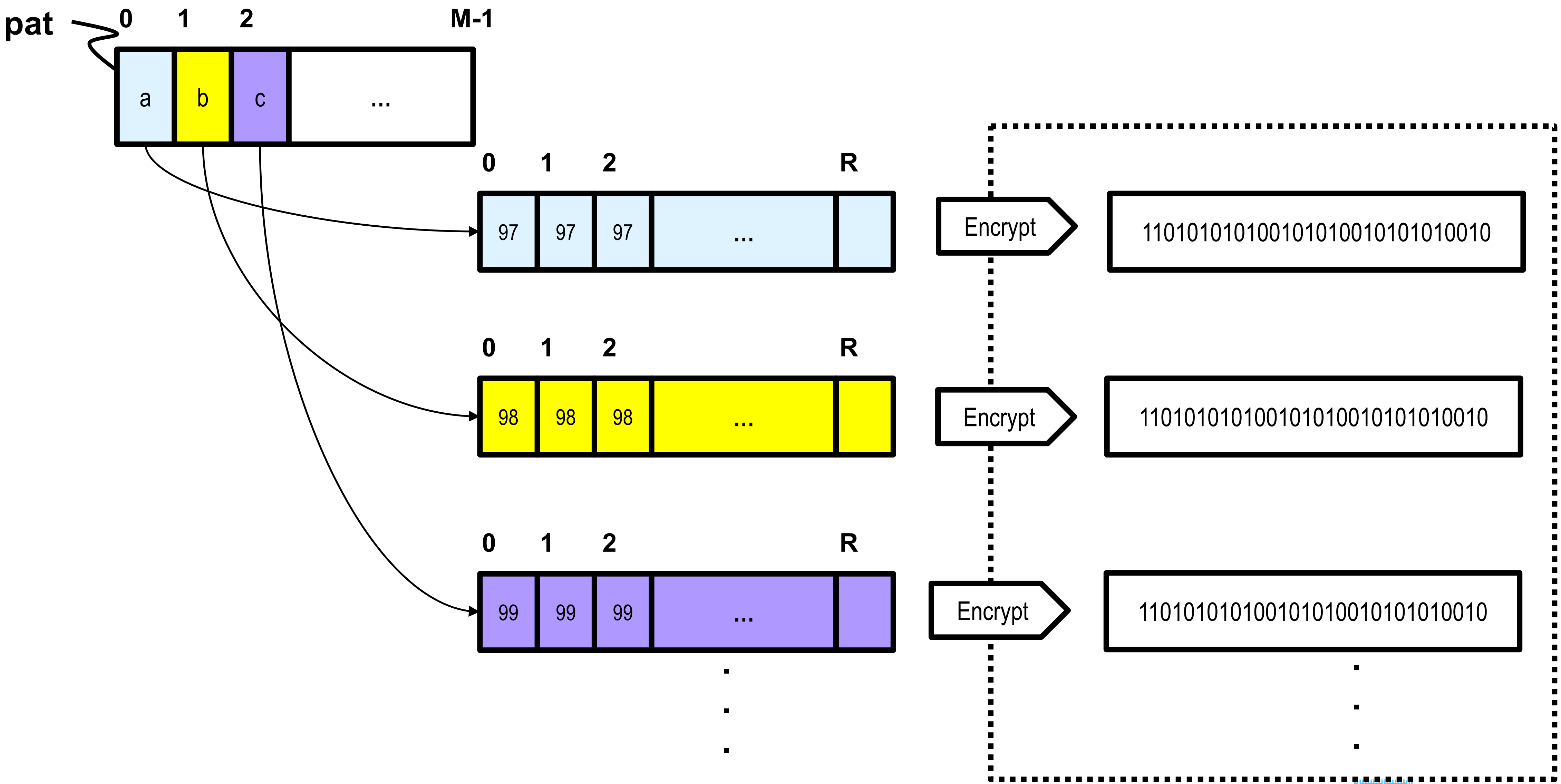
Selecting batch size K and # Ciphertexts L

- K is selected based on the ciphertext ringsize **R**, and the lengths of **M** and **N** so that a large size **txt** can be encrypted
  - Ringsize is determined by the system based on plaintext modulus **P**, depth, and security
    - Can be read via **cc->GetRingDimension()**
- The actual values of **K** and **L** can be found with a simple iteration. See the code for details on the computation.
- Note there is overlap in the batches, i.e. characters in **txt** are encoded with an overlap in order to allow ‘sliding’ the hash over **L** ciphertexts without skipping any characters in the original **txt**
- Note **K** must be less than the specified multiplicative depth to guarantee decryption

Encrypting **pat**

- We vectorize **pat** the same way as before, except we use the helper function encrypt_repeated_integer() to repeat each integer **R** times.
- Now we can use the same code as V1 to compute every hash operation (generation, update, comparison) in parallel over all the batches of the **txt**

- We vectorize **pat** the same way as before, except we use **encrypt_repeated_integer()** to repeat each integer ringsize times.



PALISADE

Decryption

- Decryption: each zero entry in each output ciphertext vector now provides an indication of a match within that batch.
- We compute an offset into each batch and use that to generate the overall offset of the match in **txt**
  - The offset into each batch is **batch_index *(L –M+1)**
    -- see code for details

PALISADE Timing results (24 processor server)

- You can find the code in the repository. It has hardwired values for **txt** and **pat** but you can modify the code and play around.
- Version 1 and 2 perform both plaintext and encrypted search and compare the results.
- Encrypted v1: Search for "Anna" in text of 32 characters takes **18.3** sec (one occurrence)
- Encrypted v2: Looking for "Anna" in the entire text of "Anna Karenina" (1666846 characters) takes **16.5** sec (825 occurrences)
- Why is v2 faster than v1?
  - The algorithm used determined that the entire text will fit in 29 batches vs the 32 of v1, so there are fewer updates of the hash.
  - For M=4 ("A,n,n,a") V1 →32-4 = 28 hash updates vs V2→ 29-4 = 25 hash updates

PALISADE Practical Observations for Building Integer Systems

- Manually setting parameters, i.e. hardcoding correct values for plaintext modulus **p** for a large depth (> 20) can be tricky.
  - PALISADE can throw exceptions that are not easy to understand
  - For example: during development we tried using p = 65537 and depth 32 which caused an exception in deep the math layer (shift overflow)
    - An internal computation during parameter computation overflowed the maximum big integer bit-width specified in the default MATHBACKEND 2.
    - Increasing the maximum bit-width at library build time or using one of the dynamically sized backends (4 or 6) would avoid this.
    - p = 7864433 worked well for our example.
- Write your code incrementally, to find values of **p**, depth and **R**, you may need some trial and error to find good values.
- Multiplicative depth for BFVrns is always approximate (though generous)
  - You can often get away with a few more multiplies than depth dictates, but at a risk of failing to decrypt.
  - Very large numbers of additions may reduce the overall depth as well.

Questions?

THANK YOU

dcousins@dualtitytech.com