Department of Physics

WEP Encryption/Decryption on GPUs
(Programming for Performance and Safety - C++ vs. Rust)

Seminar Talk: Physics on GPUs

Presented By: Rajpreet Singh

Focus of this talk

- W.E.P. – Wired Equivalent Privacy or Wireless Encryption Protocol
- Treat the encryption and decryption as black box algorithms and profile its complexity
- Discuss the advantages of RUST over C++
- Try to Implement the WEP Encryption/Decryption with **CUDA C++** as well as **rustacuda**
- Compare the performance of the same code between C++ and Rust

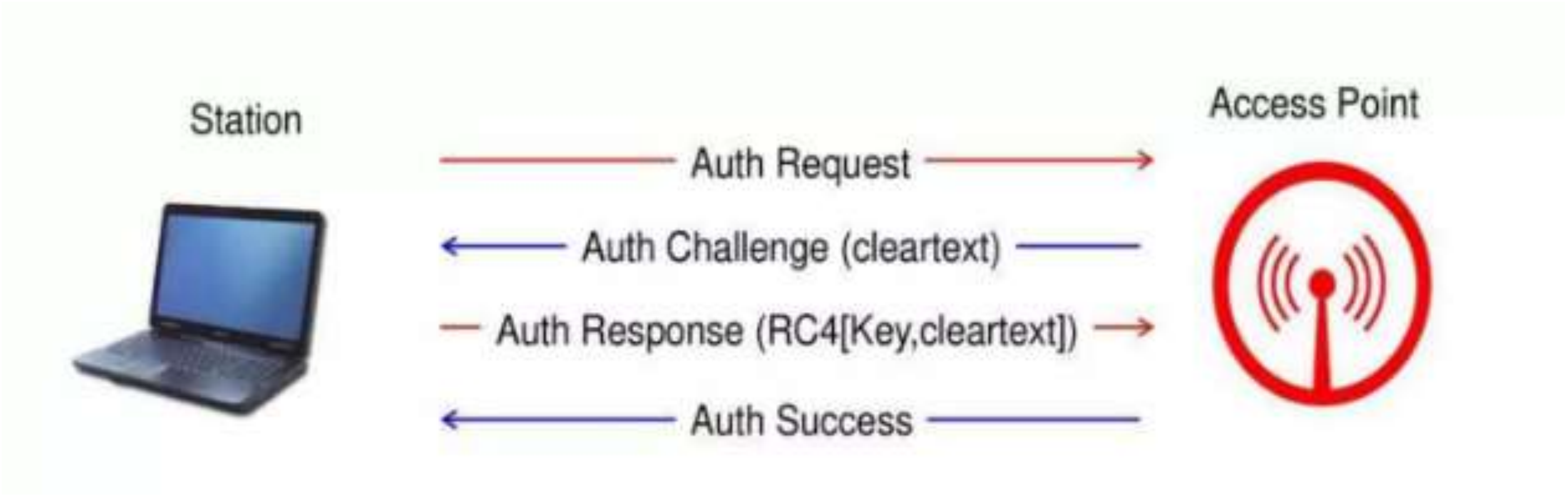
Motivation

- W.E.P. – Wired Equivalent Privacy or Wireless Encryption Protocol
- This protocol was introduced with the original 802.11 standard as a means to provide authentication and encryption to wireless LAN implementations
- There are two methods of authentication used with WEP: Open System Authentication and Shared Key Authentication

Motivation

- W.E.P. Key is used for authentication in a four-step challenge-response handshake
  1. The Client sends an authentication request to the Access Point
  2. The Access Point replies with a clear-text challenge
  3. The Client encrypts the challenge-text using the configured WEP key and sends it back in another authentication request
  4. The Access Point decrypts the response. If this matches the challenge text, the Access Point sends back a positive reply.
- After the authentication and association, the pre-shared WEP key is also used for encrypting the data frames using RC4

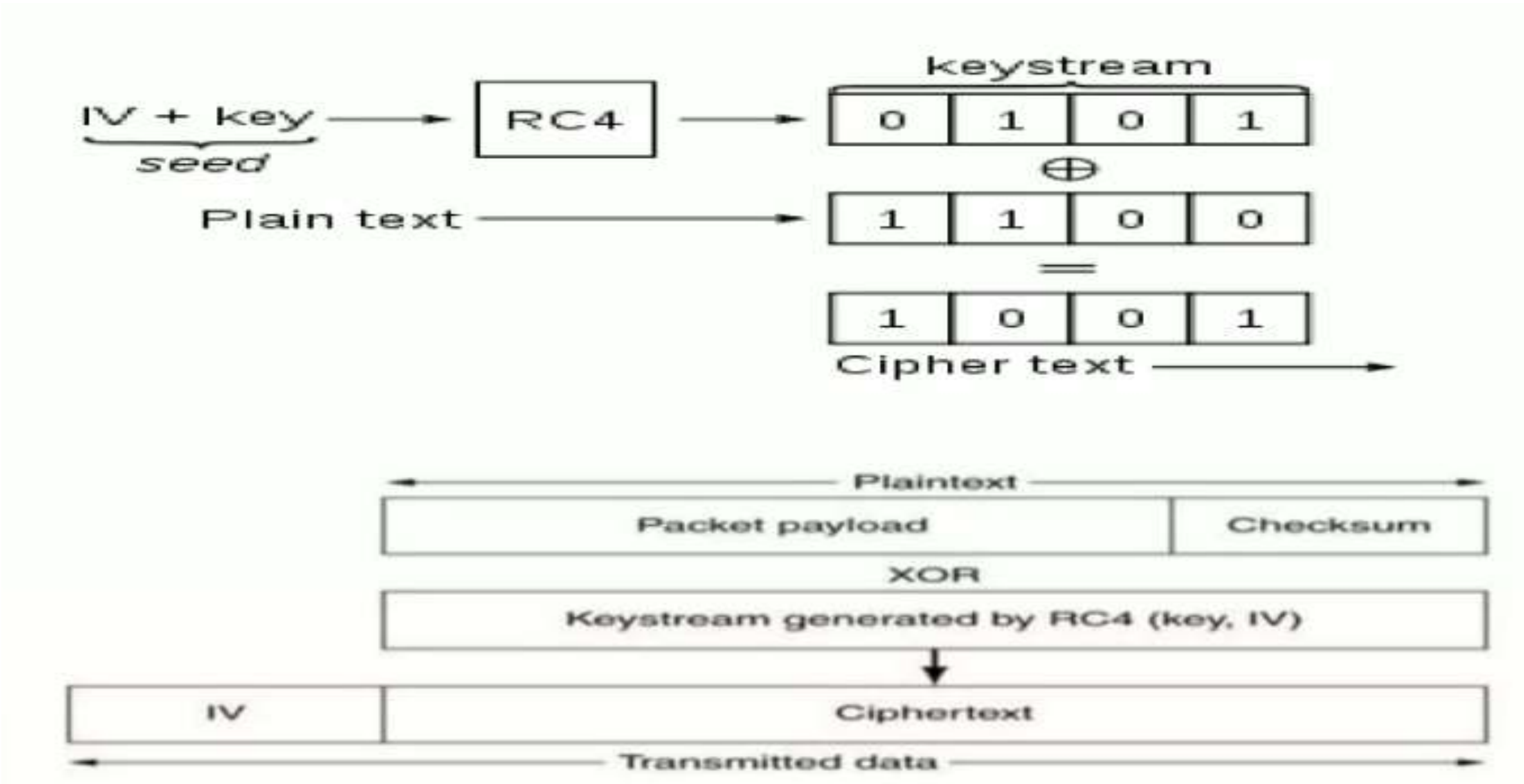
WEP Authentication



WEP Encryption

- Based on the Rivest Cipher 4 (RC4) stream cypher with a Pre-shared Secret Keys (PSK) of 40 or 104 bits, depending on the implementation. A 24-bit pseudorandom initialization Vector (IV) is concatenated with the pre-shared key to generate the pre-packet keystream used by RC4 for the actual encryption and decryption process. Thus, the resulting keystream could be 64 or 128 bits long
- In the Encryption phase, the keystream is encrypted with the XOR cypher with the plaintext data to obtain the encrypted data. While in the decryption phase, the encrypted data is XOR-encrypted with the keystream to obtain the plaintext data.

WEP Encryption



Weakness of WEP

- No key management
- IV is just 24 bits and transmitted as clear text
- IV values can be reused
- No standard procedure for IV generation
- First few key streams bytes are predictable in RC4 algorithm with weak IVs

About Rust

- Rust is a systems programming language sponsored by Mozilla Research
- It's a "Safe, Concurrent, Practical" open-source language supporting functional and imperative-procedural paradigms
- Rust is syntactically similar to C++, but it provides better memory safety while still maintaining performance
- **MEMORY SAFETY**
    - The system is designed to be memory safe, and it does not permit null pointers, dangling pointers, or data races in safe code
    - Rust code library provides an option type, which can be used to test if a pointer has Some value or None
    - Rust also introduces additional construct to manage lifetimes, and the compiler reasons about these through its borrow checker.

About Rust

- Rust is a systems programming language sponsored by Mozilla Research
- **MEMORY MANAGEMENT**
  - Rust does not use an automated garbage collection system like those used by Go, Java or .NET Framework
  - Resources are managed through resource acquisition is initialization (RAII)
  - Rust also favours stack allocation of values and does not perform implicit boxing
- **OWNERSHIP**
  - Rust has an ownership system where all values have a unique owner where the scope of the value is the same as the scope of the owner
  - Values can be passed by immutable reference using &T, by mutable reference using &mut T or by value using T
  - At all times, there can be either be multiple immutable references or one mutable reference
  - The Rust compiler enforces these rules at compile time and also checks that all references are valid

About Rust

- Rust is a systems programming language sponsored by Mozilla Research
- **TYPES and POLYMORPHISM**
  - The type system supports a mechanism similar to type classes, called "Traits", inspired directly by the Haskell Language
  - The implementation of Rust generics is similar to the typical implementation of C++ templates
  - The object system within Rust is based around implementations, traits and structured types
- **Who use RUST ??**

ATLASSIAN

Atlassian : We use Rust in a service for analyzing petabytes of source code.

coursera

Coursera : Programming Assignments in secured Docker containers.

moz://a

Mozilla : Building the Servo browser engine, integrating into Firefox, other projects.

Dropbox

Dropbox : Optimizing cloud file-storage.

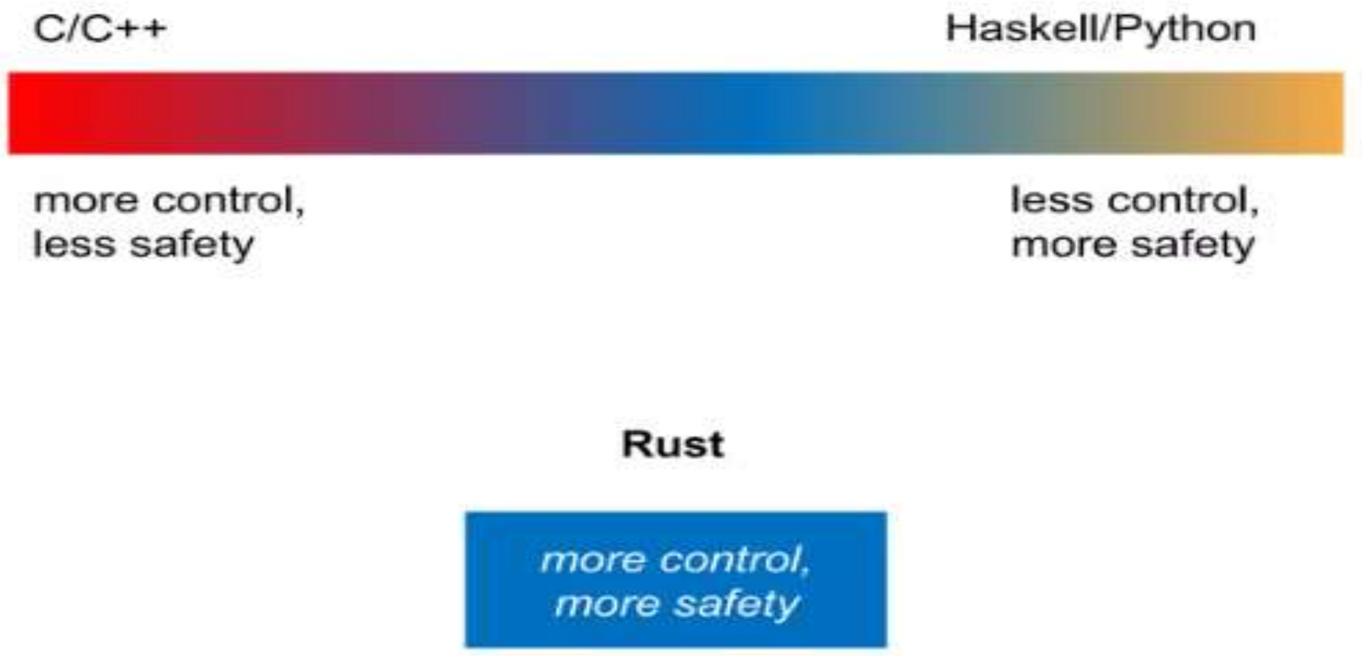
SAMSUNG SmartThings

SmartThings : Memory-safe embedded applications on our SmartThings Hub and supporting services in the cloud.

npm

npm, Inc : Replacing C and rewriting performance-critical bottlenecks in the registry service architecture.

Classification of Rust



Classification of Rust

	Type safety	Type expression	Type checking	Garbage Collector
C	unsafe	explicit	static	No
C++	unsafe	explicit	static	No
Rust	safe	implicit/ explicit	static/ dynamic	No*

Classification of Rust

C++
+ Speed, no overhead
+ Community?
+ Commitee
- Missing package manager, safety

Rust

+ Package manager, Community
+ Tooling, Documentation, Concurrency
+ Speed, no overhead?
+ RFC process
- Syntax?
- Borrow checker;)
- Packages

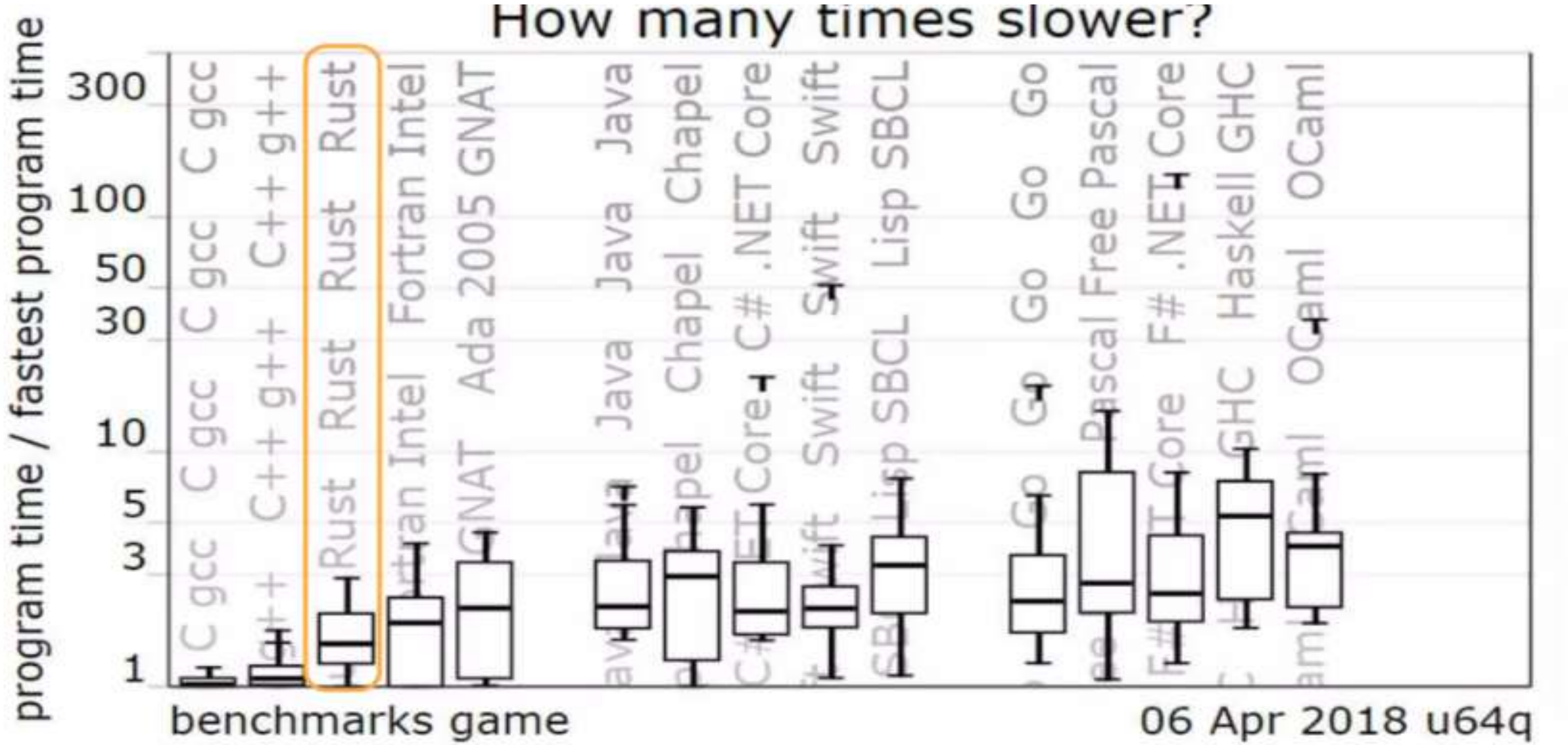
Stack Vs. Heap - Rust

```rust
fn main() {
  let y: i32 = 1; //allocated on the stack
  let x: Box<i32> = Box::new(10); //allocated on the heap

  println!("Heap {}, Stack {}", x, y);
}

//cargo run
// Heap 10, Stack 1
```

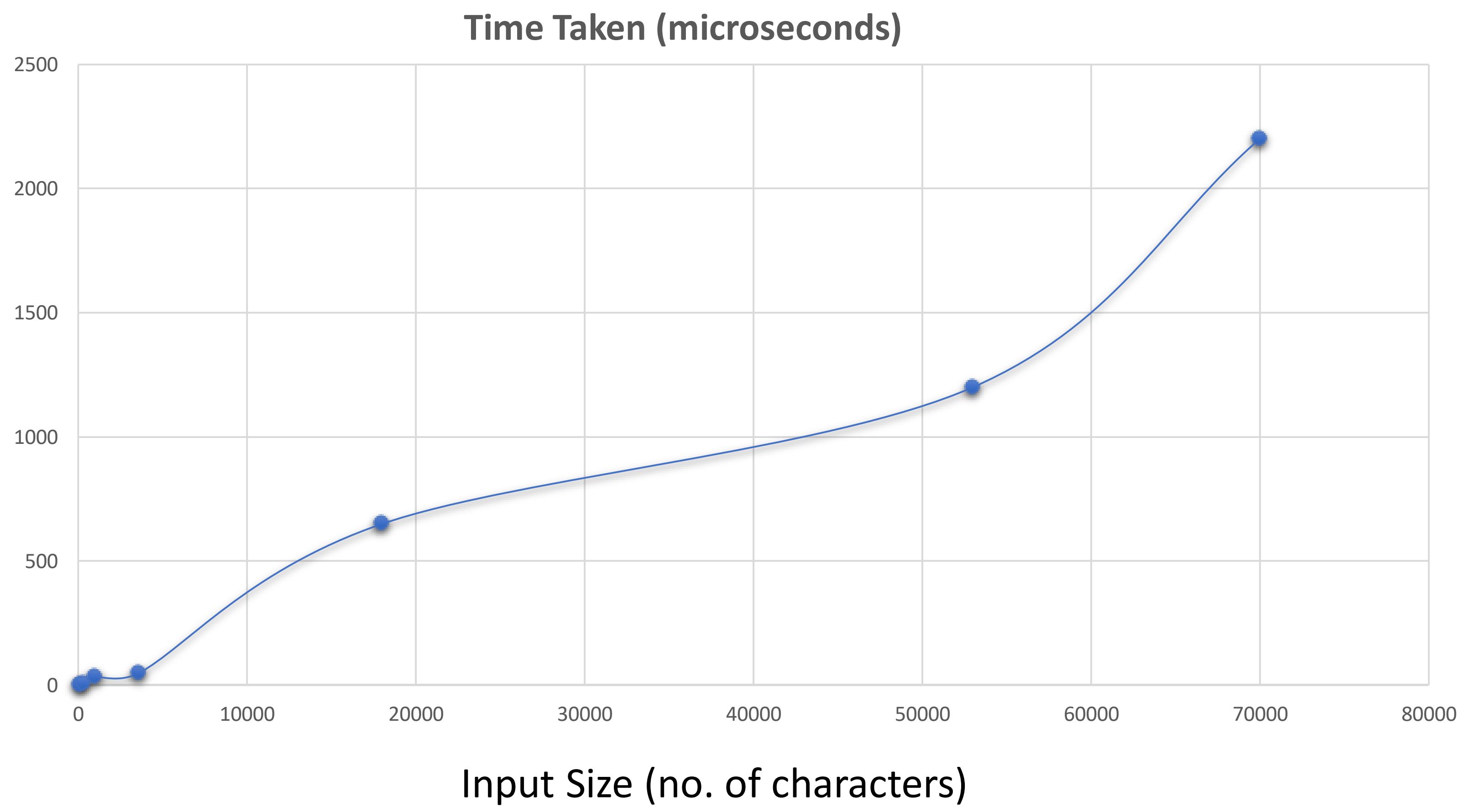
Benchmarks


How many times slower?
program time / fastest program time
300
100
50
30
10
5
3
1
C gcc
C++ g++
Rust
Fortran Intel
Ada 2005 GNAT
Java
Chapel
C# .NET Core
Swift
Lisp SBCL
Go
Free Pascal
F# .NET Core
Haskell GHC
OCaml
benchmarks game
06 Apr 2018 u64q

WEP Encryption Algorithm (C++)

```
int countCharacters(const std::string& str) {
    return str.size();
}
template<typename Duration, typename Function, typename... Args>
Duration time_taken_by_the_function(const std::string& function_name, Function&& function, Args&&... args) {
    auto Tstart = std::chrono::high_resolution_clock::now();
    std::forward<Function>(function)(std::forward<Args>(args)...);
    auto Tend = std::chrono::high_resolution_clock::now();
    auto time_duration = std::chrono::duration_cast<Duration>(Tend - Tstart);
    std::cout << "Executing the '" << function_name << "' took " << time_duration.count() << " microseconds" << std::endl;
    return time_duration;
}
std::string wep_encrypt(const std::string& plaintext, const std::vector<uint8_t>& wep_key) {
    std::string ciphertext;
    size_t key_length = wep_key.size();
    for (size_t i = 0; i < plaintext.size(); i++) {
        char encrypted_byte = plaintext[i] ^ wep_key[i % key_length];
        ciphertext += encrypted_byte;
    }
    return ciphertext;
}
std::string to_binary(const std::string& input) {
    std::string binary;
    for (const auto& byte : input) {
        binary += std::bitset<8>(byte).to_string();
    }
    return binary;
}
int main() {
    std::string plaintext = "we present how to learn regression models on Lie groups\n"
                            "and apply our formulation to visual object tracking tasks. Many transformations used\n"
                            "order approximation to the geodesic error";
    int Characters_Count =  countCharacters(plaintext);
    std::cout << "Number of characters: " << Characters_Count << std::endl;
    std::vector<uint8_t> wep_key = { 0xAA, 0xBB, 0xCC, 0xDD, 0xEE };
    auto duration = time_taken_by_the_function<std::chrono::microseconds>("wep_encrypt", wep_encrypt, plaintext, wep_key);
    std::string ciphertext = wep_encrypt(plaintext, wep_key);
    return 0;
}
```

WEP Encryption Algorithm (C++)

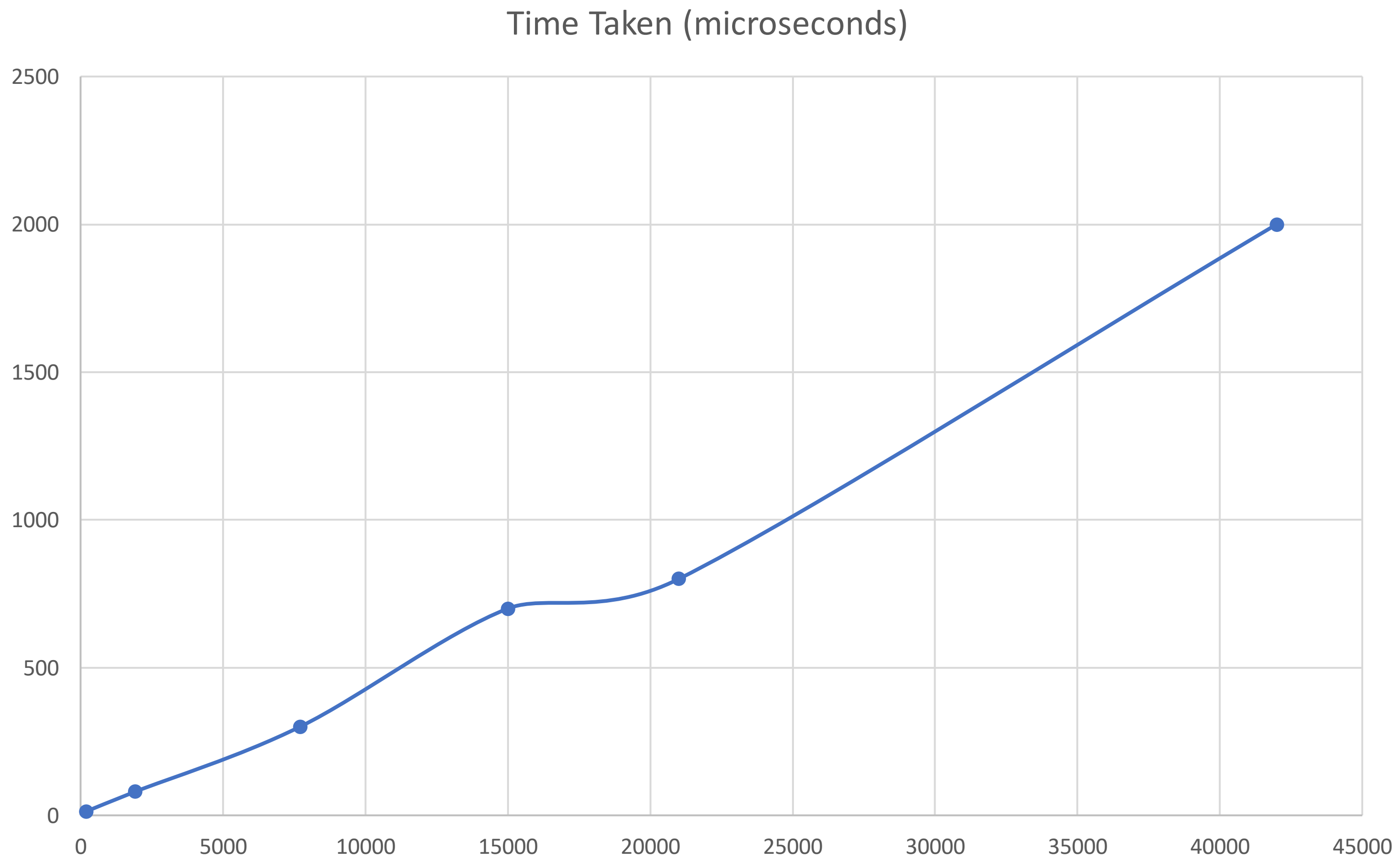


WEP Decryption Algorithm (C++)

```
int countCharacters(const std::string& str) {
    return str.size();
}
template<typename Duration, typename Function, typename... Args>
Duration time_taken_by_the_function(const std::string& function_name, Function&& function, Args&&... args) {
    auto Tstart = std::chrono::high_resolution_clock::now();
    std::forward<Function>(function)(std::forward<Args>(args)...);
    auto Tend = std::chrono::high_resolution_clock::now();
    auto time_duration = std::chrono::duration_cast<Duration>(Tend - Tstart);
    std::cout << "Executing the '" << function_name << "' took " << time_duration.count() << " microseconds" << std::endl;
    return time_duration;
}
std::string wep_decrypt(const std::string& ciphertext, const std::vector<uint8_t>& wep_key) {
    std::string plaintext;
    size_t key_length = wep_key.size();

    // Remove spaces from the binary ciphertext
    std::string binary_ciphertext = "";
    for (char c : ciphertext) {
        if (c != ' ') {
            binary_ciphertext += c;
        }
    }
    // Decrypt each byte from the binary ciphertext
    for (size_t i = 0; i < binary_ciphertext.size(); i += 8) {
        std::string byte_str = binary_ciphertext.substr(i, 8);
        uint8_t encrypted_byte = static_cast<uint8_t>(std::bitset<8>(byte_str).to_ulong());
        char decrypted_byte = encrypted_byte ^ wep_key[i / 8 % key_length];
        plaintext += decrypted_byte;
    }
    return plaintext;
}
int main() {
    std::string ciphertext = "11011101 11011110 11101100 10101101 10011100 11001111 11001000 10101001 10110011 10011010 10001010 11010011 10100011 10101010 11001110 11011110
    std::vector<uint8_t> wep_key = { 0xAA, 0xBB, 0xCC, 0xDD, 0xEE };
    std::string decrypted_text = wep_decrypt(ciphertext, wep_key);
    int Characters_Count =  countCharacters(ciphertext);
    std::cout << "Number of characters: " << Characters_Count << std::endl;
    std::cout << "Ciphertext (binary): " << ciphertext << std::endl;
    std::cout << "Decrypted Text: " << decrypted_text << std::endl;
    auto duration = time_taken_by_the_function<std::chrono::microseconds>("wep_decrypt", wep_decrypt, ciphertext, wep_key);
    return 0;
}
```

WEP Decryption Algorithm (C++)

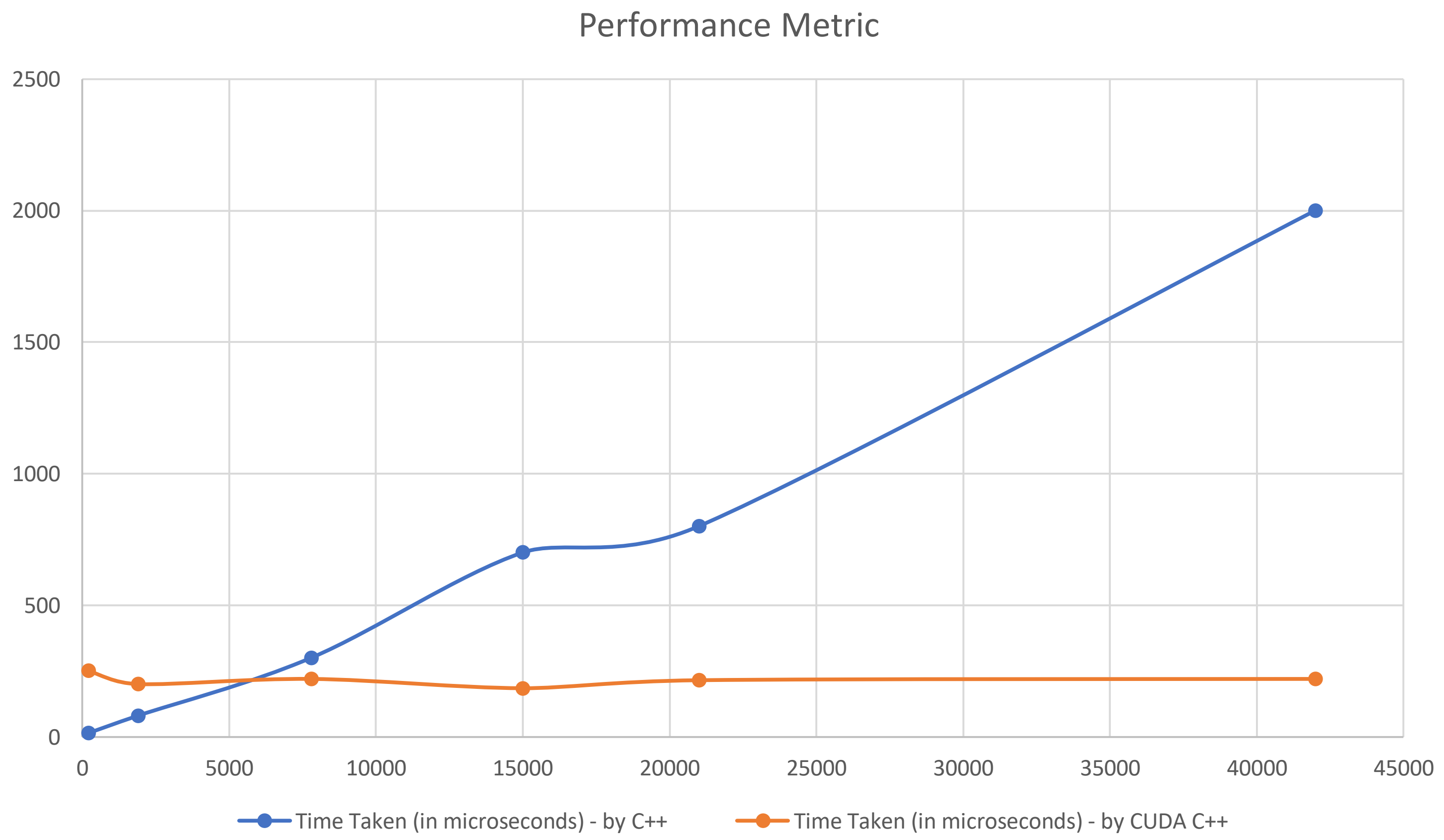


Input Size (no. of characters)

WEP Encryption Algorithm (Rust)

```rust
use std::time::Instant;
fn count_characters(s: &str) -> usize {
    s.chars().count()
}
fn time_taken_by_the_function<F, R, Args>(function_name: &str, function: F, args: Args) -> R
where
    F: FnOnce(Args) -> R,
{
    let t_start = Instant::now();
    let result = function(args);
    let t_end = Instant::now();
    let time_duration = t_end - t_start;
    println!(
        "Executing the '{}' took {} microseconds",
        function_name,
        time_duration.as_micros()
    );
    result
}
fn wep_encrypt(plaintext_and_key: (&str, &[u8])) -> String {
    let (plaintext, wep_key) = plaintext_and_key;
    let key_length = wep_key.len();
    let mut ciphertext = String::new();
    for (i, byte) in plaintext.bytes().enumerate() {
        let encrypted_byte = byte ^ wep_key[i % key_length];
        ciphertext.push_str(&format!("{:08b}", encrypted_byte));
    }
    ciphertext
}
fn to_binary(input: &str) -> String {
    let mut binary = String::new();
    for byte in input.bytes() {
        binary.push_str(&format!("{:08b}", byte));
    }
    binary
}
fn main() {
    let plaintext = "we present how to learn regression models on Lie groups\n\
                     we present how to learn regression models on Lie groups\n\
                     order approximation to the geodesic error";
    let characters_count = count_characters(plaintext);
    println!("Number of characters: {}", characters_count);
    let wep_key = vec![0xAA, 0xBB, 0xCC, 0xDD, 0xEE];
    let _duration = time_taken_by_the_function("wep_encrypt", wep_encrypt, (plaintext, &wep_key));
    let ciphertext = wep_encrypt((plaintext, &wep_key));
    println!("Ciphertext (binary): {}", ciphertext);
}
```

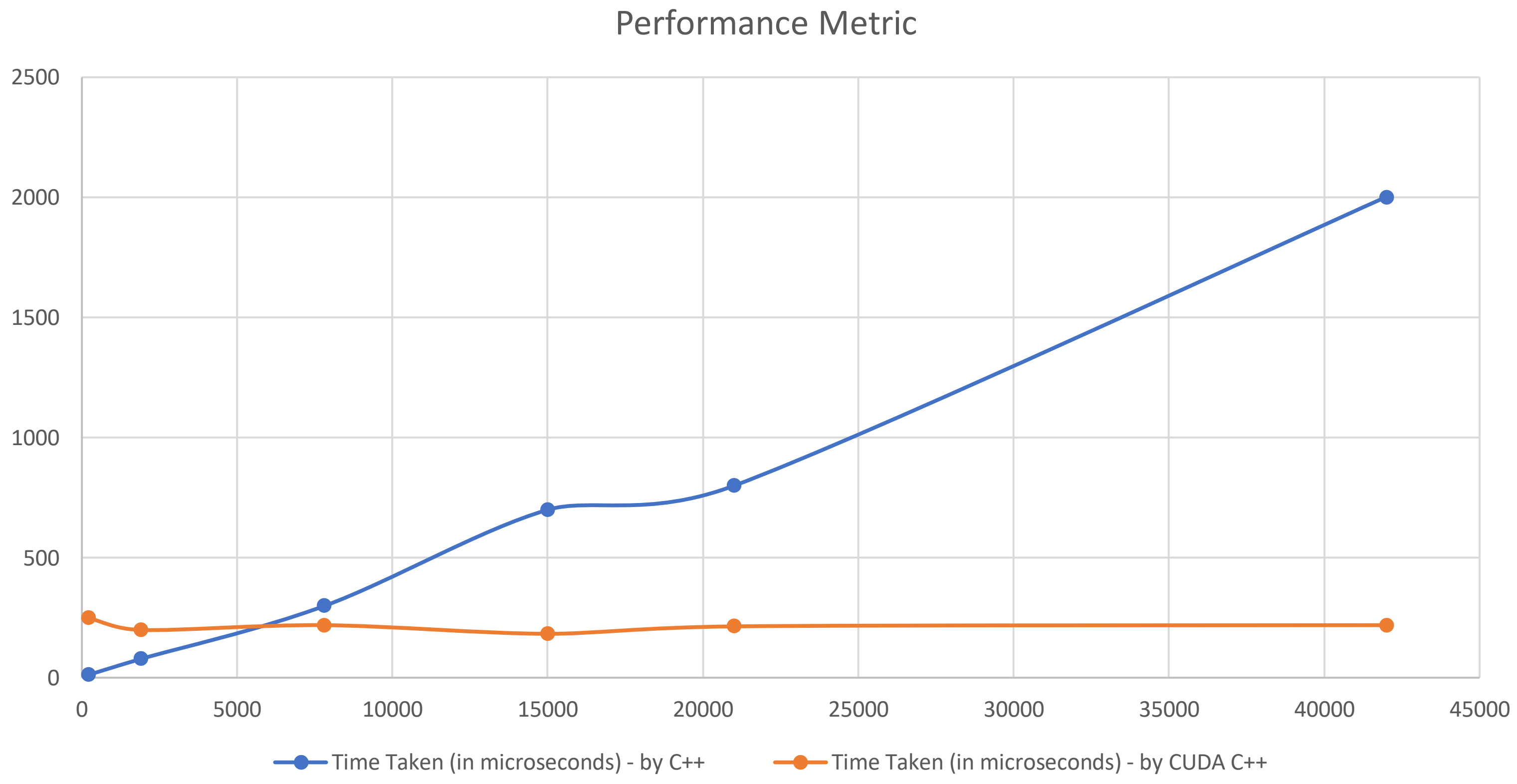
WEP Encryption Algorithm (Rust Vs. C++)



WEP Decryption Algorithm (Rust)

```rust
use std::time::Instant;
fn count_characters(s: &str) -> usize {
    s.chars().count()
}
fn time_taken_by_the_function<F, R, Args>(function_name: &str, function: F, args: Args) -> R
where
    F: FnOnce(Args) -> R,
{
    let t_start = Instant::now();
    let result = function(args);
    let t_end = Instant::now();
    let time_duration = t_end - t_start;
    println!(
        "Executing the '{}' took {} microseconds",
        function_name,
        time_duration.as_micros()
    );
    result
}
fn wep_decrypt(ciphertext_and_key: (&str, &[u8])) -> String {
    let (ciphertext, wep_key) = ciphertext_and_key;
    let binary_ciphertext = ciphertext.replace(" ", "");
    let mut plaintext = String::new();
    let key_length = wep_key.len();
    for chunk in binary_ciphertext.as_bytes().chunks(8) {
        let byte_str = std::str::from_utf8(chunk).unwrap();
        let encrypted_byte = u8::from_str_radix(byte_str, 2).unwrap();
        let decrypted_byte = encrypted_byte ^ wep_key[plaintext.len() % key_length];
        plaintext.push(decrypted_byte as char);
    }
    plaintext
}
fn main() {
    let ciphertext = "11011101 11011110 11101100 10101101 10011100 11001111 11001000 10101001 10110011 10011010 10001010 11010011 10100011 10101010 11001110 11011110
    let wep_key = vec![0xAA, 0xBB, 0xCC, 0xDD, 0xEE];
    let decrypted_text = wep_decrypt((&ciphertext, &wep_key));
    let characters_count = count_characters(&ciphertext);
    println!("Number of characters: {}", characters_count);
    println!("Ciphertext (binary): {}", ciphertext);
    println!("Decrypted Text: {}", decrypted_text);
    let _duration = time_taken_by_the_function("wep_decrypt", wep_decrypt, (&ciphertext, &wep_key));
}
```

WEP Decryption Algorithm (Rust Vs. C++)



Huge Gain in Performance for Decryption of the Ciphertext

WEP Decryption Algorithm (CUDA C++)

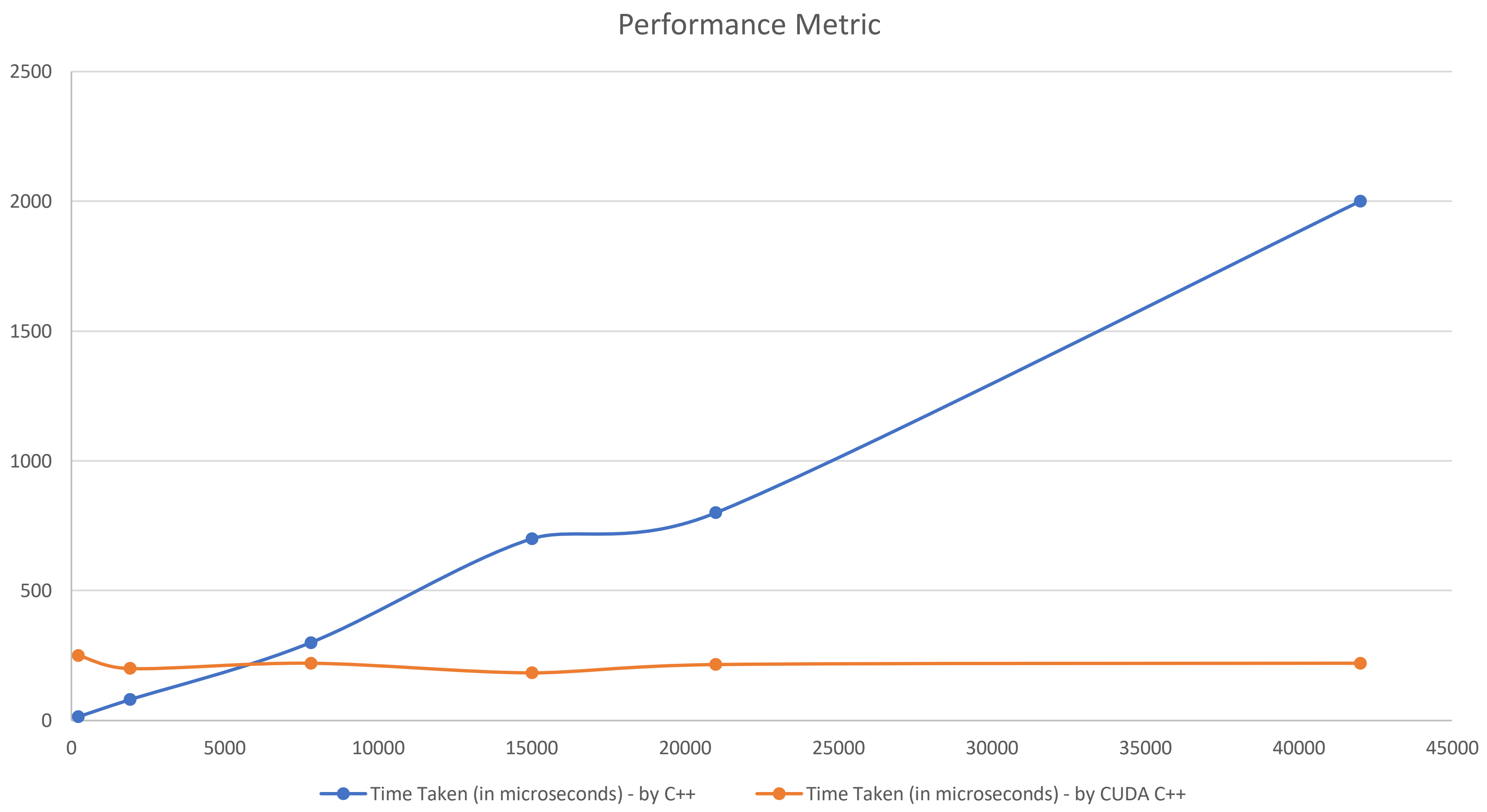
```
// CUDA kernel to perform decryption on the GPU
__global__ void wep_decrypt_kernel(const char* ciphertext, size_t ciphertext_length, const uint8_t* wep_key, size_t key_length, char* decrypted_text) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    int stride = blockDim.x * gridDim.x;

    for (int i = tid; i < ciphertext_length; i += stride) {
        if (ciphertext[i] != ' ') {
            // Convert 8-bit binary representation to an 8-bit integer
            uint8_t encrypted_byte = 0;
            for (int j = 0; j < 8; j++) {
                if (ciphertext[i + j] == '1') {
                    encrypted_byte |= (1 << (7 - j));
                }
            }
            decrypted_text[i] = encrypted_byte ^ wep_key[i / 8 % key_length];
        }
    }
}
```

WEP Decryption Algorithm (CUDA C++)

```
std::string wep_decrypt(const std::string& ciphertext, const std::vector<uint8_t>& wep_key) {
    std::string plaintext;
    size_t key_length = wep_key.size();
    // Allocate memory on the host
    char* host_decrypted_text = new char[ciphertext.size() + 1];
    // Allocate memory on the device (GPU)
    char* device_ciphertext;
    char* device_decrypted_text;
    uint8_t* device_wep_key;
    size_t ciphertext_size = ciphertext.size();
    cudaMalloc((void**)&device_ciphertext, (ciphertext_size + 1) * sizeof(char));
    cudaMalloc((void**)&device_decrypted_text, (ciphertext_size + 1) * sizeof(char));
    cudaMalloc((void**)&device_wep_key, key_length * sizeof(uint8_t));
    // Copy data from host to device
    cudaMemcpy(device_wep_key, wep_key.data(), key_length * sizeof(uint8_t), cudaMemcpyHostToDevice);
    cudaMemcpy(device_ciphertext, ciphertext.c_str(), (ciphertext_size + 1) * sizeof(char), cudaMemcpyHostToDevice);
    // Launch the kernel on the GPU
    int num_threads_per_block = 256;
    int num_blocks = (ciphertext_size + num_threads_per_block - 1) / num_threads_per_block;
    wep_decrypt_kernel<<<num_blocks, num_threads_per_block>>>(device_ciphertext, ciphertext_size, device_wep_key, key_length, device_decrypted_text);
    // Copy the decrypted text back to the host
    cudaMemcpy(host_decrypted_text, device_decrypted_text, (ciphertext_size + 1) * sizeof(char), cudaMemcpyDeviceToHost);
    host_decrypted_text[ciphertext_size] = '\0';
    // Clean up memory on the device
    cudaFree(device_ciphertext);
    cudaFree(device_decrypted_text);
    cudaFree(device_wep_key);
    // Clean up memory on the host
    plaintext = std::string(host_decrypted_text);
    delete[] host_decrypted_text;
    return plaintext;
}
```

WEP Decryption Algorithm (CUDA C++)



WEP Decryption Algorithm (rustacuda)

Technische Universität München

```
use std::time::Instant;
use rustacuda::prelude::*;
use rustacuda::launch;
use rustacuda::memory::DeviceBox;
fn count_characters(s: &str) -> usize {
    s.chars().count()
}
fn time_taken_by_the_function<F, R, Args>(function_name: &str, function: F, args: Args) -> R
where
    F: FnOnce(Args) -> R,
{
    let t_start = Instant::now();
    let result = function(args);
    let t_end = Instant::now();
    let time_duration = t_end - t_start;
    println!(
        "Executing the '{}' took {} microseconds",
        function_name,
        time_duration.as_micros()
    );
    result
}
fn wep_decrypt(ciphertext_and_key: (&str, &[u8])) -> String {
    let (ciphertext, wep_key) = ciphertext_and_key;
    let binary_ciphertext = ciphertext.replace(" ", "");
    let mut plaintext = String::new();
    let key_length = wep_key.len();
    for chunk in binary_ciphertext.as_bytes().chunks(8) {
        let byte_str = std::str::from_utf8(chunk).unwrap();
        let encrypted_byte = u8::from_str_radix(byte_str, 2).unwrap();
        let decrypted_byte = encrypted_byte ^ wep_key[plaintext.len() % key_length];
        plaintext.push(decrypted_byte as char);
    }
    plaintext
}
```

WEP Decryption Algorithm (rustacuda)

```rust
fn main() {
    // Initialize the CUDA API
    rustacuda::init(CudaFlags::empty()).expect("Failed to initialize CUDA");
    // Get the first available device
    let device = Device::get_device(0).expect("Failed to get CUDA device");
    let _context = Context::create_and_push(ContextFlags::MAP_HOST | ContextFlags::SCHED_AUTO, device)
        .expect("Failed to create CUDA context");
    // Allocate memory on the GPU
    let ciphertext = "11011101 11011110 11101100 10101101 10011100 11001111 11001000 10101001 10110011 10011010 10001010 11010011 10100011 10101010 11001110
    let wep_key = vec![0xAA, 0xBB, 0xCC, 0xDD, 0xEE];
    let ciphertext_len = ciphertext.len() + 1; // Including null terminator
    let mut device_ciphertext = DeviceBox::new(ciphertext_len).expect("Failed to allocate memory on GPU");
    let mut device_decrypted_text = DeviceBox::new(ciphertext_len).expect("Failed to allocate memory on GPU");
    let mut device_wep_key = DeviceBox::new(wep_key.as_slice()).expect("Failed to allocate memory on GPU");
    // Copy data from host to device
    let ciphertext_bytes = ciphertext.as_bytes();
    device_ciphertext.copy_from(ciphertext_bytes).expect("Failed to copy ciphertext to GPU");
    // Launch the CUDA kernel
    let num_threads_per_block = 256;
    let num_blocks = (ciphertext_len + num_threads_per_block - 1) / num_threads_per_block;
    let stream = Stream::new(StreamFlags::NON_BLOCKING, None).expect("Failed to create CUDA stream");
    let params = (device_ciphertext.as_ptr(), ciphertext_len, device_wep_key.as_ptr(), wep_key.len(), device_decrypted_text.as_mut_ptr());
    unsafe {
        launch!(wep_decrypt_kernel<<<num_blocks, num_threads_per_block, 0, stream>>>(
            params
        )).expect("Failed to launch CUDA kernel");
    }
    // Copy the decrypted text back to the host
    let mut host_decrypted_text = vec![0u8; ciphertext_len];
    device_decrypted_text.copy_to(host_decrypted_text.as_mut_slice()).expect("Failed to copy decrypted text from GPU");
    // Clean up memory on the device
    device_ciphertext.free().expect("Failed to deallocate memory on GPU");
    device_decrypted_text.free().expect("Failed to deallocate memory on GPU");
    device_wep_key.free().expect("Failed to deallocate memory on GPU");
    // Clean up CUDA API
    rustacuda::deinit();
    let decrypted_text = String::from_utf8(host_decrypted_text).unwrap();
    let characters_count = count_characters(&ciphertext);
    let _duration = time_taken_by_the_function("wep_decrypt", wep_decrypt, (&ciphertext, &wep_key));
```

Learnings

- Speedup $\approx$ 8, but only for Decryption of Ciphertext
- Rust provides safer code, but only in the Host (CPU) **NOT** on the Device (GPU)
- Rust performs far better in Serial Implementation for Decryption part (see plots)
- GPU implementation of the decryption code with "rustacuda" is expected to perform insanely good
- Industry is expected to make a transition from **C++** to **Rust**
- Vectors in C++ gives optimized performance as compared to arrays
- W.E.P. is not safe (not recommended at all)
- If you want to see a state of the art in Cryptographic World: *See* => *Homomorphic Encryption*