LEA Block Cipher in Rust Language: Trade-off between Memory Safety and Performance

Sangwon Kim, Siwoo Eum, Minho Song, Hwajeong Seo

Hansung University, Seoul (02876), South Korea

kim3875@gmail.com, shuraatum@gmail.com, smino0906@gmail.com, hwajeong84@gmail.com

Abstract. Cryptography implementations of block cipher have been written in C language due to its strong features on system-friendly features. However, the C language is prone to memory safety issues, such as buffer overflows and memory leaks. On the other hand, Rust, novel system programming language, provides strict compile-time memory safety guarantees through its ownership model. This paper presents the implementation of LEA block cipher in Rust language, demonstrating features to prevent common memory vulnerabilities while maintaining performance. We compare the Rust implementation with the traditional C language version, showing that while Rust incurs a reasonable memory overhead, it achieves comparable the execution timing of encryption and decryption. Our results highlight Rust's suitability for secure cryptographic applications, striking the balance between memory safety and execution efficiency.

Keywords: Rust· Memory Safety· LEA· Ownership Rules

1 Introduction

LEA [1] is a 128-bit block cipher developed in C to provide confidentiality in high-speed environments such as big data and cloud, as well as in lightweight environments such as mobile devices. It excels in resource-constrained environments, providing efficient encryption and decryption while minimizing resource usage. The simplified designs of lightweight block ciphers reduce hardware footprint and manufacturing costs, making them ideal for mass-produced devices. Furthermore, these ciphers often incorporate countermeasures against side-channel attacks, enhancing their security in real-world scenarios. Their flexibility allows for adaptation to diverse applications, while standardization ensures interoperability and reliability.

Despite these advantages, LEA has vulnerabilities because of the nature of the C. This gives memory access autonomy to programmers but is error-prone. Memory safety issues in such language allows attackers to exploit memory vulnerabilities, potentially modifying the program's behavior or even taking full control of its execution flow [2]. These vulnerabilities encompass the following categories (not exhaustive):

Dangling pointers: If the program accidentally deallocates an active object, the memory manager might overwrite its contents.
Double frees: Making multiple calls to free objects that have already been deallocated can cause freelist-based allocators to malfunction.
Buffer overflows: Out-of-bound writes can damage the contents of active objects in the heap.
Heap metadata overwrites: If heap metadata is stored close to heap objects, an out-of-bound write can corrupt the metadata.
Uninitialized reads: Reading values from newly allocated or unallocated memory results in undefined behavior.

1.1 Motivation and Purpose

The motivation behind this research is to explore the potential benefits of implementing LEA in Rust. Given the critical importance of memory safety in cryptographic applications, Rust's capability to prevent common memory-related vulnerabilities presents a significant advantage. This study aims to demonstrate that Rust can offer a secure and efficient alternative to traditional C implementations of block ciphers.

The purpose of this paper is to provide a detailed implementation of LEA in Rust, compare its performance and memory usage with the original C version, and discuss the associated benefits and trade-offs. By leveraging Rust's safety features, we seek to deliver a robust and secure implementation that meets the demands of contemporary cryptographic applications.

2 Memory Safety and System Security

Memory safety is a crucial aspect of computer security that concerns the protection of memory space in computer systems from various types of errors that can lead to security vulnerabilities.

The concept revolves around safeguarding applications from accessing memory that they should not, preventing errors that could corrupt the operating system or other programs. Memory safety is vital because it defends against unauthorized access and manipulation, which can have dire consequences including data leaks, system crashes, and breaches of security. Especially low-level programming involves careful memory management, where errors can lead to significant bugs and security issues in essential code. Memory safety ensures avoiding these mistakes by ensuring memory is only accessed and released correctly. Unsafe memory practices are among the most severe software vulnerabilities today. They lead to security breaches and are common: Microsoft identified that 70% of security flaws in Windows stem from improper memory usage [3]; and in the Chromium browser, 36% of bugs result from use-after-free incidents [4], with another 33% caused by different types of unsafe memory practices [5].

2.1 Understanding Memory Safety

At its core, memory safety involves ensuring that software reliably manages memory allocation and access. This includes preventing bugs such as buffer overflows, dangling pointers, and other forms of memory corruption or mismanagement. These vulnerabilities occur when a program writes data outside the bounds of allocated memory or reads from memory locations that have been freed or not properly initialized.

Buffer overflows, for example, are among the most common and dangerous security threats in software development. They happen when a program attempts to store more data in a buffer (a sequential section of memory) than it is capable of holding. This excess data can overwrite adjacent memory, potentially altering the execution path of the application by overwriting function pointers or other critical data. Attackers can exploit such vulnerabilities to execute arbitrary code, leading to unauthorized actions by the system.

2.2 Strategies for Ensuring Memory Safety

To combat these threats, several strategies have been developed. One effective approach is the use of programming languages that enforce strict memory safety. Languages like Rust and Swift are designed with memory safety as a core feature, using ownership models and automatic memory management to prevent common security flaws.

3 Safe and Efficient System Programming with Rust

Rust achieves memory safety through its unique ownership system with rules that the compiler checks at compile time. This system eliminates common bugs like null pointer dereferencing, buffer overflows, and memory leaks without the need for a garbage collector.

3.1 Ownership Rules

In Rust, every piece of data has a single owner, and ownership can be transferred from one part of the program to another. This ownership system ensures that when the owner, such as a variable, goes out of scope, Rust automatically deallocates the associated memory, preventing memory leaks [6].

Another crucial aspect of Rust's memory management is borrowing. Rust allows data to be borrowed via references, which must have lifetimes shorter than that of the owner. This ensures that references cannot outlive the data they point to, thus avoiding dangling pointers. Moreover, Rust enforces strict rules regarding references: it allows either multiple immutable references or a single mutable reference at one time, but not both. This rule ensures that data can either be read from multiple places without being altered, or it can be altered from a single place without concurrent reads, which effectively prevents data races [7].

The Rust compiler is pivotal in enforcing these ownership and borrowing rules through its borrow checker. The borrow checker analyses all references in the code to ensure compliance with borrowing rules, checking the lifetimes of variables and their uses [8]. It ensures that no two mutable references exist simultaneously for the same data and that references do not outlive their source. This compile-time enforcement of ownership and borrowing rules means that common memory safety issues such as null pointer dereferencing and buffer overflows are almost impossible in Rust.

3.2 Immutable and Mutable References

Rust distinguishes between immutable and mutable references to enhance memory safety and concurrency control.

- **Immutable References:** An immutable reference allows you to read data without modifying it. Multiple immutable references to the same data can exist simultaneously, enabling safe concurrent reads. This is possible because the data is guaranteed not to change while these references are active, preventing data races and ensuring consistency. For example:

Listing 1.1: Immutable Reference in Rust

```rust
let x = 5;
let y = &x; // Immutable reference to x
println!("The value of y is: {}", y);
```

- **Mutable References:** A mutable reference allows you to both read and modify data. However, to maintain safety, Rust only permits one mutable reference to a particular piece of data at a time. This exclusivity ensures that no other references can access the data concurrently, thereby avoiding race conditions and ensuring that the data cannot be altered unexpectedly while it is being read or written to. For example:

Listing 1.2: Mutable Reference in Rust

```rust
let mut x = 5;
let y = &mut x; // Mutable reference to x
*y += 1;
println!("The value of x is: {}", x);
```

These capabilities allow Rust programs to achieve high levels of performance and reliability, crucial for system-level and embedded applications where both performance and safety are paramount. Rust does all this without the runtime overhead of a garbage collector, which is a significant advantage in performance-critical applications.

4 Comparative Analysis of LEA Key Scheduling in C and Rust

In the following sections, we will examine the implementation of LEA key scheduling in Rust and compare it with a C implementation. We will analyze how Rust's rules and safety features come into play and how they contribute to safer and more robust code.

The LEA used in the C code was developed by the National Security Research Institute (NSR) of Korea. It is currently distributed as open-source software by the Korea Internet & Security Agency (KISA).

4.1 LEA Key Scheduling in Rust

In Rust, separate functions are used for each key size: `round_key_gen_24`, `round_key_gen_28`, and `round_key_gen_32`, corresponding to 128-bit, 192-bit, and 256-bit keys, respectively. This separation allows each function to be tailored specifically for its respective key size without needing to check the key size at runtime. Let's take a look at `round_key_gen_24` and identify the part of this code that ensures memory safety.

Listing 1.3: `round_key_gen_24`

```rust
const KEY_CONST: [u32; 8] = [ ...
// The key schedule uses several constants for generating round keys
];
const MASTER_KEY: [u32; 8] = [ ...
/* a master key. It is denoted as a concatenation of 32-bit words. K =
    (K[0], K[1], K[2], K[3]) when Len(K) = 128; K =
    K[0]||K[1]||...||K[5] when Len(K) = 192; K = K[0]||K[1]||...||K[7]
    when Len(K) = 256. */
];

    fn round_key_gen_24(mk: &[u32; 8], erk: &mut [u32; 192], drk: &mut
        [u32; 192]){
    // Key scheduling operations for 128-bit key
    }

    fn main() {
    let mut enc_round_key: [u32; 192] = [0; 192];
    let mut dec_round_key: [u32; 192] = [0; 192];
    round_key_gen_24(&MASTER_KEY, &mut enc_round_key, &mut
        dec_round_key);
    }
```

In the declaration of this function, we can see one of the key features of Rust: immutable (`&`) and mutable (`&mut`) references. These references are fundamental to Rust's memory safety guarantees. Immutable references allow multiple parts of your code to read from the data without the risk of data being altered unexpectedly. On the other hand, mutable references enforce that only one reference to the data exists at any one time, preventing data races.

- **Data Races**: By allowing only one mutable reference or multiple immutable references at any point in time, Rust ensures that data races are prevented. This is crucial in a function like `round_key_gen_24`, where key generation involves sensitive data that must be handled carefully to avoid security vulnerabilities.
- **Borrow Checker**: Rust's compiler includes a borrow checker that enforces memory safety rules at compile time. This borrow checker ensures that references do not outlive the data they refer to, preventing the creation of dangling references. For instance, in the `round_key_gen_24` function, the borrow checker guarantees that the references to `erk` and `drk` do not persist beyond the function's scope, thereby eliminating the risk of use-after-free errors. When `round_key_gen_24` is called, `&mut enc_round_key` and `&mut dec_round_key` are passed as mutable references. The borrow checker ensures that these references are only valid within the function and cannot be used once the function returns. This mechanism prevents any unsafe access to these references outside their intended lifetime, maintaining memory safety and data integrity. Additionally, using `enc_round_key` as a mutable reference allows the function to directly modify the original array without needing to return a new array, thus improving performance and reducing memory overhead.
- **Function Signature**: The function signature itself is designed to ensure memory safety. By requiring the keys as references rather than taking ownership, the function avoids unnecessary data copying, which not only saves on performance but also reduces the complexity of memory management.

Furthermore, the use of constants like `MASTER_KEY` as immutable references means that these values can be shared safely across multiple parts of the program without duplicating data. This approach minimizes memory usage and ensures that the original data is not accidentally modified during the cryptographic operations. Thus, Rust's approach to managing memory and data access not only promotes safety and security but also enhances the performance and reliability of cryptographic functions like `round_key_gen_24`.

4.2 LEA Key Scheduling in C

Listing 1.4: `lea_set_key_generic`

```c
typedef struct lea_key_st
{
    unsigned int rk[192];
    unsigned int round;
} LEA_KEY;

void lea_set_key_generic(LEA_KEY *key, const unsigned char *mk, unsigned
    int mk_len) {
    const unsigned int* _mk = (const unsigned int*)mk;
    switch(mk_len) {
    case 16: // 128-bit key
        // Key scheduling operations for 128-bit key
        break;
    case 24: // 192-bit key
        // Key scheduling operations for 192-bit key
        break;
    case 32: // 256-bit key
        // Key scheduling operations for 256-bit key
        break;
    }
    // Further operations...
}
```

In the C code, the input `mk` is of type `const unsigned char*`. This `mk` is cast to `const unsigned int*` to access it in 4-byte chunks. This is used to set up the key schedule for the LEA encryption algorithm.

In C, it is possible to cast pointer types. If `mk` is given as `const unsigned char*`, it means accessing data in byte units. However, if it is cast to `const unsigned int*`, it accesses data in 4-byte units. The problems that can arise from this include:

- **Memory Alignment:**
  Data of type `unsigned int` must be aligned to 4-byte boundaries. However, data of type `unsigned char*` can be accessed in byte units and may not be aligned to 4-byte boundaries. If `mk` is not aligned to a 4-byte boundary, casting it to `const unsigned int*` and accessing it may result in undefined behavior.
- **Memory Access Errors:**
  If the alignment is incorrect when accessing data in 4-byte units, the CPU may raise an exception due to unaligned memory access. This is particularly problematic on some architectures.

For example, consider the following byte array for `mk`:

```
unsigned char mk[16] = { 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08,
    0x09, 0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F, 0x10 };
```

Casting this array to `unsigned int*` and accessing it would be interpreted as follows:

```
const unsigned int* _mk = (const unsigned int*)mk;
```

Here, _mk[0] would be 0x04030201 and _mk[1] would be 0x08070605. However, if mk is not aligned to a 4-byte boundary, for example:

```
unsigned char* mk = (unsigned char*)malloc(17) + 1;
```

In this case, mk is not aligned to a 4-byte boundary. The '+1' moves the pointer by one byte, making it unaligned. Casting this to unsigned int* and accessing it may result in undefined behavior because the address is not a multiple of 4.

In C, great care must be taken when accessing memory through pointer type casting [9]. Particularly, when casting a byte-pointer to a 4-byte pointer, memory alignment issues must be considered. Languages like Rust enforce memory safety to prevent such issues. In Rust, the compiler warns or errors out on such casts, helping the programmer to write safe code.

5 Benchmark

After implementing LEA in Rust, we used Jemalloc and the CPU time measurement method to compare them. And our environment for measuring the memory statistics is a MacBook Air with the following specifications:

- **Model:** MacBook Air
- **Chip:** Apple M2 (2022)
- **Memory:** 8 GB
- **Operating System:** macOS Sonoma 14.5

5.1 Jemalloc

jemalloc is a memory allocation library designed to manage memory efficiently, especially in multithreaded environments. Initially developed for the FreeBSD operating system, it has gained widespread adoption due to its superior management capabilities compared to standard memory allocators.

One of the primary advantages of jemalloc is its ability to significantly reduce memory fragmentation, ensuring more contiguous memory allocations and deallocations. This is crucial for environments requiring high concurrency [10]. Traditional memory allocators can become bottlenecks due to lock contention when threads access memory resources simultaneously [11]. jemalloc mitigates this by employing a multi-arena allocation system [12], reducing contention and improving overall performance.

Another significant feature of jemalloc is its configurability, allowing developers to tune various parameters for optimal performance in different scenarios, whether for high-load servers or memory-intensive computations. jemalloc also offers robust tools for monitoring and analyzing memory usage [13], aiding in performance tuning and leak detection.

The adoption of jemalloc in popular databases like MongoDB and Redis, as well as in programming languages such as Rust, underscores its effectiveness. These systems benefit from jemalloc by handling large numbers of simultaneous connections or operations more efficiently than traditional memory allocators [14].

5.1.1 Jemalloc Statistics jemalloc provides comprehensive statistics for optimizing and understanding memory usage. These statistics include allocated and resident memory, memory mapped from the operating system, active memory, and fragmentation metrics. These statistics offer insights into several critical aspects of memory management, enabling performance tuning, leak detection, and capacity planning. By using this statistics, we can see the differences between C and Rust. The result is shown below.

	Allocated	Active	Metadata	Resident	Mapped	Retained
C	174,720	376,832	2,616,832	2,899,968	6,668,288	0
Rust	4,227,960	5,750,784	2,645,632	8,372,224	12,124,160	0

Table 1: Jemalloc Statistics Comparison

5.1.2 Interpretation of Jemalloc Statistics jemalloc provides comprehensive statistics for optimizing and understanding memory usage. These statistics include allocated and resident memory, memory mapped from the operating system, active memory, and fragmentation metrics. Here's a detailed interpretation:

– **Allocated**: This represents the total amount of memory that the program has requested and successfully allocated from the system through Jemalloc. It indicates the cumulative size of all memory blocks that have been allocated at some point in time. This metric helps to understand how much memory the application demands during its execution.
– **Active**: Active memory refers to the amount of allocated memory that is still in use by the application. It includes all the memory currently being utilized by the program and can also encompass memory that has been fragmented but is still within allocated blocks. This number can sometimes be higher than Allocated memory due to fragmentation and internal overhead, where parts of the allocated memory are not freed properly or are kept for future use.
– **Metadata**: Metadata represents the memory used internally by Jemalloc to manage the allocation and deallocation processes. This includes data structures and tables that keep track of allocated and free memory blocks. Metadata is essential for efficient memory management but adds overhead to the total memory usage.
– **Resident**: TResident memory is the portion of the process's memory that is actually held in RAM (physical memory). It excludes any memory that has been swapped out to disk or is otherwise not present in physical memory. Resident memory gives an indication of the actual physical memory footprint of the application at any given moment.
– **Mapped**: Mapped memory refers to the total virtual memory address space that has been allocated by Jemalloc. This includes all memory regions that

the allocator has mapped, whether they are currently in use or not. Mapped memory provides insight into the total memory resources that the system has reserved for the application, which can impact the overall system memory availability.
- **Retained**: Retained memory is the amount of memory that Jemalloc has reserved for future use but is not currently allocated to the application. This memory is kept by Jemalloc to satisfy future allocation requests without needing to go back to the operating system for new memory, improving performance at the cost of higher memory usage. In this particular table, both C and Rust show 0 retained memory, meaning Jemalloc has no reserved but unused memory at the moment.

This shows that the Rust program generally uses more memory than the C program. Specifically, there is a significant difference in both active memory and resident memory. This indicates that the Rust program is allocating and using more memory overall.

5.2 CPU time measurement

CPU time measurement assesses how effectively a program uses processor resources. Unlike real (wall-clock) time, which includes waiting for I/O operations, CPU time measures the duration a CPU spends processing the actual instructions of a program [15], crucial for understanding software efficiency and performance [16].

CPU time is categorized into user CPU time and system CPU time. User CPU time is the duration the processor spends executing the program's own instructions [17]. System CPU time is the time spent executing system calls made by the program.

Various tools measure CPU time, differing by operating system [18]. On Unix/Linux, tools like ‘time‘, ‘top‘, and ‘ps‘ are commonly used [19]. The ‘time‘ command reports user CPU time, system CPU time, and real time taken, breaking down how long the program ran versus CPU usage in user and system modes.

In C, the `clock()` function measures processor time consumed by a program. Part of the C Standard Library (`<time.h>`), it returns clock ticks since the program started. To convert this to seconds, divide by `CLOCKS_PER_SEC`, representing ticks per second. The utility of `clock()` is its simplicity and directness in providing CPU time, excluding time when the program is inactive. However, it measures CPU time as a sum across all threads, not per thread, which can be a limitation in multi-threaded applications.

In Rust, tools like ‘SystemTime‘ and ‘Instant‘, and crates like ‘cpu_time::ProcessTime‘ and ‘backtrace::Backtrace‘, are used for performance measurement and debugging. These tools help optimize application performance and enhance debugging. Using these tools, we compare C and Rust implementations. The results are shown below:

These interpretations and analyses provide a basis for understanding the performance implications of jemalloc and CPU time statistics, which can guide

Key Size (bits)	Key Schedule (ms)	Encryption (ms)	Decryption (ms)
128	3.675	2.132	1.853
192	6.348	2.111	1.813
256	6.573	2.139	1.815

Table 2: LEA CPU Time in C

Key Size (bits)	Key Schedule (ms)	Encryption (ms)	Decryption (ms)
128	12.907	21.632	21.947
192	17.354	21.715	23.409
256	30.104	22.342	23.632

Table 3: LEA CPU Time in Rust

further optimizations and enhancements in system design and application development.

6 Conclusion

This study implemented the LEA block cipher in Rust, highlighting the potential safety advantages over traditional C implementations. Analyzing the LEA code revealed that using Rust could enhance memory safety due to its strict ownership and borrowing principles, which prevent common vulnerabilities such as buffer overflows and memory leaks. However, performance measurements using jemalloc and CPU time assessment indicated that Rust's implementation incurs a performance overhead compared to C. Rust's memory safety features, including bounds checking and borrow checking, introduce additional computational overhead, resulting in slower key generation and encryption/decryption processes. Despite these performance trade-offs, the enhanced security provided by Rust makes it a viable option for cryptographic applications where safety is paramount.

References

1. D. Hong *et al.*, "Lea: A 128-bit block cipher for fast encryption on common processors," in *Information Security Applications: 14th International Workshop, WISA 2013, Jeju Island, Korea, August 19-21, 2013, Revised Selected Papers*, vol. 14, Springer International Publishing, 2014.
2. L. Szekeres, M. Payer, T. Wei, and D. Song, "Sok: Eternal war in memory," in *2013 IEEE Symposium on Security and Privacy*, pp. 48–62, IEEE, 2013.
3. M. Miller, "Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape," last accessed: July 25, 2022 URL: `https://github.com/microsoft/MSRC-Security-Research/blob/master/presentations/2019_02_BlueHatIL/2019_01%20-%20BlueHatIL%20-%20Trends%2C%20challenge%2C%20and%20shifts%20in%20software%20vulnerability%20mitigation.pdf`.
4. The Chromium Projects, "Memory safety," last accessed: July 25, 2022 URL : `https://www.chromium.org/Home/chromium-security/memory-safety/`.

5. ISRG, "What is memory safety and why does it matter?," [Online]. Available: `https://www.memorysafety.org/docs/memory-safety/`. [Accessed 06. May 2022].
6. W. Crichton, G. Gray, and S. Krishnamurthi, "A grounded conceptual model for ownership types in rust," *Proceedings of the ACM on Programming Languages*, vol. 7, pp. 1224 – 1252, 2023.
7. W. Crichton, "The usability of ownership," *ArXiv*, vol. abs/2011.06171, 2020.
8. D. J. Pearce, "A lightweight formalism for reference lifetimes and borrowing in rust," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 43, pp. 1 – 73, 2021.
9. P. Li, "Safe systems programming languages," *Department of Computer and Information Science, University of Pennsylvania, October*, vol. 6, 2004.
10. A. Umayabara and H. Yamana, "Mcmalloc: A scalable memory allocator for multithreaded applications on a many-core shared-memory machine," in *2017 IEEE International Conference on Big Data (Big Data)*, pp. 4846–4848, 2017.
11. A. Gidenstam, M. Papatriantafilou, and P. Tsigas, "Nbmalloc: Allocating memory in a lock-free manner," *Algorithmica*, vol. 58, pp. 304–338, 2010.
12. W. Huang, Y. Qian, W. Srisa-an, and J. M. Chang, "Object allocation and memory contention study of java multithreaded applications," in *IEEE International Conference on Performance, Computing, and Communications, 2004*, pp. 375–382, 2004.
13. T. Ferreira, R. Matias, A. Macêdo, and L. B. de Araujo, "An experimental study on memory allocators in multicore and multithreaded applications," in *2011 12th International Conference on Parallel and Distributed Computing, Applications and Technologies*, pp. 92–98, 2011.
14. E. Berger, K. McKinley, R. Blumofe, and P. Wilson, "Hoard: a scalable memory allocator for multithreaded applications," 2000.
15. M. Ji, E. Felten, and K. Li, "Performance measurements for multithreaded programs," *Proceedings of the 1998 ACM SIGPLAN Conference*, pp. 161–170, 1998.
16. A. Gupta, J. Sampson, and M. Taylor, "Quality time: A simple online technique for quantifying multicore execution efficiency," in *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 169–179, 2014.
17. S. Chen, K. R. Joshi, M. Hiltunen, R. Schlichting, and W. Sanders, "Using cpu gradients for performance-aware energy conservation in multitier systems," *Sustain. Comput. Informatics Syst.*, vol. 1, pp. 113–133, 2011.
18. N. R. Tallent and J. Mellor-Crummey, "Effective performance measurement and analysis of multithreaded applications," *ACM SIGPLAN Notices*, vol. 44, pp. 229–240, 2009.
19. R. Bianchini and B. Lim, "Evaluating the performance of multithreading and prefetching in multiprocessors," *J. Parallel Distributed Comput.*, vol. 37, pp. 83–97, 1996.