Implementing SHA-256 in XV6 Operating System

Syed Muhammad Deebaj Haider Kazmi†, Ikhlas Ahmed Khan‡
† 26012, ‡ 27096
School of Mathematics and Computer Science
Institute of Business Administration
Karachi, Pakistan

***Abstract*—This research presents a comprehensive implementation and performance analysis of the SHA-256 cryptographic hashing algorithm within the XV6 educational operating system. By developing three distinct implementations - in user space, kernel space, and through a system call interface - we demonstrate the architectural trade-offs and security implications of cryptographic operations across different privilege boundaries. Our findings provide insights into optimizing cryptographic operations in minimal operating systems and highlight the performance impacts of crossing privilege boundaries.**

***Index Terms*—SHA-256, hashing, cryptography, XV6**

I. Introduction

This project explores the implementation and analysis of the **SHA-256 cryptographic hash function** within the *XV6 operating system environment* across three distinct architectural levels. The implementation is realized through three different approaches:

- A **user space implementation** that operates entirely within the user process boundary.
- A **system call implementation** that bridges user space and kernel space.
- A **kernel space implementation** integrated directly into the XV6 kernel.

These varied implementations enable a comprehensive examination of how cryptographic operations perform and behave at different privilege levels within a minimal operating system environment. Through this exploration, we aim to understand the trade-offs in **performance**, **security**, and **complexity** that each implementation approach presents within the educational context of XV6's simplified architecture.

II. The SHA-256 Algorithm

A. Cryptographic Hash Functions

Cryptographic hash functions serve as fundamental building blocks in modern computer security systems, transforming arbitrary input data into fixed-size output values. These functions act as digital fingerprinting mechanisms, taking data of any size and producing a unique, fixed-length string that represents the original input. The transformation is designed to be **deterministic**, ensuring that identical inputs always produce identical hash values, while even minimal changes in the input result in significantly different output values.

```
$ usha hello
SHA-256 hash of 'hello': 2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824
$ usha hallo
SHA-256 hash of 'hallo': d3751d33f9cd5049c4af2b462735457e4d3baf130bcbb87f389e349fbaeb20b9
```

Fig. 1. Example of avalanche effect

The security of cryptographic hash functions relies on several essential properties:

- **One-way functions**: It is computationally infeasible to derive the original input from its hash value. This property, known as *preimage resistance*, ensures that hash functions can securely store sensitive information such as passwords.
- **Avalanche effect**: A slight modification in the input produces a dramatically different hash value (Fig.1), making it impossible to predict output changes based on input modifications.
- **Collision resistance**: It is extremely difficult to find two different inputs that produce the same hash value.

In practical applications, cryptographic hash functions play crucial roles in data integrity verification, digital signatures, password storage, and message authentication codes. They provide a way to verify data integrity without storing or transmitting the original data, making them invaluable in scenarios where security and efficiency are paramount.

B. SHA-256 Overview and Applications

SHA-256 stands as one of the most widely adopted cryptographic hash functions in the modern digital landscape. As a member of the SHA-2 family developed by the *National Security Agency (NSA)*, SHA-256 produces a 256-bit (32-byte) hash value, typically represented as a 64-character hexadecimal string. The algorithm's name derives from its output size, reflecting the substantial security margin it provides against various cryptographic attacks.

The strength of SHA-256 lies in its carefully designed structure and mathematical properties. The algorithm combines multiple rounds of complex operations, including bitwise functions, modular addition, and data mixing steps, creating a sophisticated cryptographic construct that has withstood extensive cryptanalysis. Its 256-bit output space provides a sufficiently large domain to resist *birthday attacks*, while its

internal state size and number of processing rounds offer protection against other known cryptographic attack vectors.

SHA-256 has found widespread adoption across diverse applications in modern computing:

- **Blockchain technology**: Serves as the core hashing algorithm for mining and transaction verification in numerous cryptocurrencies, including Bitcoin.
- **Digital signature schemes**: Creates message digests for signing and verification.
- **Software distribution systems**: Verifies package integrity to ensure that downloaded files haven't been tampered with during transmission.
- **Password storage systems**: Provides secure password hashing capabilities when combined with salting techniques.

The algorithm's appeal extends beyond its security properties. Its relatively efficient implementation characteristics, particularly on modern processors with specialized instructions for cryptographic operations, make it practical for both high-performance and resource-constrained environments. The standardization of SHA-256 through various national and international bodies has further cemented its position as a trusted cryptographic primitive in security-critical applications.

C. *The Algorithm*

Constants and Functions The SHA-256 algorithm utilizes several predefined constants and functions, and processes them according to a set of steps.

1) **Initial Hash Values** The square roots of the first 8 square numbers are used as initial hash values H_0 through H_7:

$$H_0 = 0x6a09e667, \quad H_1 = 0xbb67ae85$$
$$H_2 = 0x3c6ef372, \quad H_3 = 0xa54ff53a$$
$$H_4 = 0x510e527f, \quad H_5 = 0x9b05688c$$
$$H_6 = 0x1f83d9ab, \quad H_7 = 0x5be0cd19$$

```
ctx->state[0] = 0x6a09e667;
ctx->state[1] = 0xbb67ae85;
ctx->state[2] = 0x3c6ef372;
ctx->state[3] = 0xa54ff53a;
ctx->state[4] = 0x510e527f;
ctx->state[5] = 0x9b05688c;
ctx->state[6] = 0x1f83d9ab;
ctx->state[7] = 0x5be0cd19;
```

Listing 1. Initial hashes implemented in code

2) **Round Constants** The cube roots of the first 64 prime numbers are used as round constants K_0 through K_{63}. Here are the first few:

$$K_0 = 0x428a2f98, \quad K_1 = 0x71374491$$
$$K_2 = 0xb5c0fbcf, \quad K_3 = 0xe9b5dba5$$
$$\vdots$$

```
uint k[64] = {
    0x428a2f98, 0x71374491, 0xb5c0fbcf,
    0xe9b5dba5, 0x3956c25b, 0x59f111f1,
    0x923f82a4, 0xab1c5ed5, 0xd807aa98
    ...
};
```

Listing 2. A snippet of round constants in code

3) **Basic Operations** The algorithm defines the following basic bitwise operations:

$$\text{ROTR}(n)(x) = (x \gg n) \,|\, (x \ll (32-n)) \quad \text{(Right Rotate)}$$
$$\text{SHR}(n)(x) = x \gg n \quad \text{(Right Shift)}$$

4) **Core Functions** The core logical functions used in SHA-256 are:

$$\text{Ch}(x, y, z) = (x\&y) \oplus (\sim x\&z) \quad \text{(Choice function)}$$
$$\text{Maj}(x, y, z) = (x\&y) \oplus (x\&z) \oplus (y\&z) \quad \text{(Majority function)}$$
$$\Sigma_0(x) = \text{ROTR}^2(x) \oplus \text{ROTR}^{13}(x) \oplus \text{ROTR}^{22}(x) \quad \text{(Upper sigma 0)}$$
$$\Sigma_1(x) = \text{ROTR}^6(x) \oplus \text{ROTR}^{11}(x) \oplus \text{ROTR}^{25}(x) \quad \text{(Upper sigma 1)}$$
$$\sigma_0(x) = \text{ROTR}^7(x) \oplus \text{ROTR}^{18}(x) \oplus \text{SHR}^3(x) \quad \text{(Lower sigma 0)}$$
$$\sigma_1(x) = \text{ROTR}^{17}(x) \oplus \text{ROTR}^{19}(x) \oplus \text{SHR}^{10}(x) \quad \text{(Lower sigma 1)}$$

5) **Processing Steps**

a) **Padding**
The message is padded to ensure its length is a multiple of 512 bits. This involves:

- Appending a single '1' bit to the end of the message
- Adding '0' bits until the length is congruent to 448 modulo 512
- Appending the original message length as a 64-bit big-endian integer

For example, for the message "abc" (length = 24 bits), the final block will be:

$$\text{abc} + 1 + 423 \text{ zeros} + 24 \quad \text{(as a 64-bit number)}$$

b) **Message Block Decomposition**
Each 512-bit block is divided into sixteen 32-bit words M_0 through M_{15}:

$$\text{Message} = M_0, M_1, \ldots, M_{15}$$

Each word is processed in big-endian format.

c) **Message Schedule Creation**
Expand 16 words into 64 words for processing:

$$W_0, W_1, \ldots, W_{15} \quad \text{(original words)}$$

For $t = 16$ to 63:

$$W[t] = \sigma_1(W[t-2]) + W[t-7] + \sigma_0(W[t-15]) + W[t-16]$$

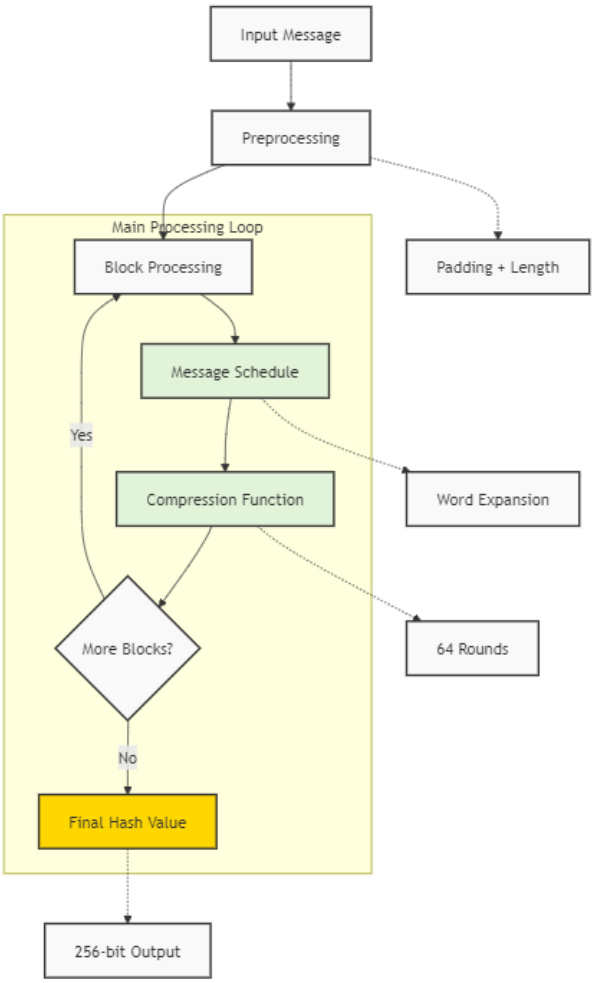

Fig. 2. Processing logic flowchart

d) **Working Variables Initialization**
Eight working variables are initialized with the current hash values:

$$a = H_0, b = H_1, c = H_2, d = H_3, e = H_4, f = H_5,$$
$$g = H_6, h = H_7$$

e) **Compression Function** For each round $t = 0$ to 63:

$$T_1 = h + \Sigma_1(e) + \text{Ch}(e, f, g) + K[t] + W[t]$$
$$T_2 = \Sigma_0(a) + \text{Maj}(a, b, c)$$

Then update the working variables:

$$h = g, \quad g = f, \quad f = e, \quad e = d + T_1$$
$$d = c, \quad c = b, \quad b = a, \quad a = T_1 + T_2$$

f) **Hash Value Update**
After processing each block, update the hash values:

$$H_0 = H_0 + a, \quad H_1 = H_1 + b, \quad \ldots, \quad H_7 = H_7 + h$$

The final hash value is obtained by concatenating $H_0, H_1, \ldots, H_7$.

D. Summary

The SHA-256 algorithm processes input data through several steps: padding, message scheduling, compression, and updating hash values. By performing complex bitwise operations, it produces a secure, irreversible hash value used in various applications, including data integrity verification, digital signatures, and blockchain technology.

III. Implementation in XV6

A. User Space

The user space implementation of SHA-256 in XV6 consists of a complete algorithmic implementation accessible through the command line interface. This implementation is structured as a standalone program that can be executed like any other user program in XV6.

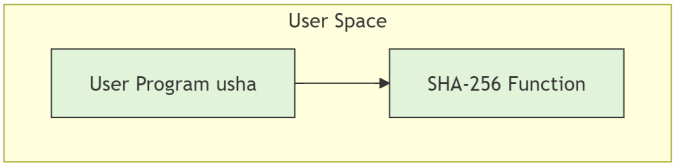

Fig. 3. Flow of program in user space

*1) **Implementation Structure**:* The implementation is organized into two main source files. The primary algorithmic implementation resides in `usha.c`, which contains the complete SHA-256 implementation including all helper functions and core algorithmic components. Integration with the XV6 system was accomplished by adding the program to the list of user programs in `Makefile.c`, ensuring it would be compiled and linked with other user programs in the system.

*2) **Helper Functions**:* The implementation relies on several utility functions to handle input processing and data manipulation. The `myStrcat` is a custom function that provides string concatenation capabilities, essential for handling input preparation where multiple strings need to be combined into a single buffer, and for output where the multiple hash strings have to be concatenated into one. For output formatting, the `toHex` function converts individual bytes into human-readable hexadecimal representation, producing a two-character string for each byte.

Data type conversion is handled by the `convertToUCharArray` function, which transforms standard C-style strings into unsigned character arrays required by the SHA-256 algorithm. The `givemestr` function serves as an input aggregator, combining multiple command-line arguments into a single coherent string while preserving proper spacing between arguments.

*3) **Core Algorithm Implementation**:* The SHA-256 algorithm is implemented through four primary functions that work together to process input data and produce the final hash:

- The process begins with `SHA256Init`, which establishes the initial state of the hashing context. This function sets up the initial hash values as specified in the SHA-256 standard and prepares the internal state variables for data processing. These initial values are crucial as they serve as the starting point for the iterative hashing process.
- Data processing is handled by `SHA256Update`, which manages the incremental processing of input data. This function operates by breaking the input into 64-byte blocks and maintaining internal state between successive calls. It can handle arbitrary input lengths and ensures that data is properly buffered when block boundaries don't align with input boundaries.
- The core transformation of the data occurs in `SHA256Transform`. This function implements the main SHA-256 computation, processing each 512-bit block through the series of operations defined by the SHA-256 specification. It performs the message schedule expansion and the compression function that iteratively updates the hash state using the defined mathematical operations.
- The hashing process is concluded by `SHA256Final`, which handles the padding of the input data according to the SHA-256 specification and produces the final hash value. This function ensures that the message length is properly encoded and appended to the data, and performs the final transformation to generate the 256-bit hash output.

```
char* myStrcat(char* dest, const char* src);
void toHex(char byteValue, char hexString[3]);
void convertToUCharArray(const char* str,
    unsigned char* ucharArray);
void SHA256Transform(SHA256_CTX *ctx, uchar
    data[]);
void SHA256Init(SHA256_CTX *ctx);
void SHA256Update(SHA256_CTX *ctx, uchar data
    [], uint len);
void SHA256Final(SHA256_CTX *ctx, uchar hash[])
    ;
char* SHA256(char* data);
```

Listing 3. Function definitions

*4) **User Interface**:* The implementation provides a straightforward interface through the `SHA256` function, which encapsulates the entire hashing process into a single call. This function coordinates the initialization, update, and finalization steps, providing a simple way to generate a hash from an input string. The main function serves as the entry point for user interaction, accepting command-line arguments and producing hexadecimal output. It utilizes the `givemestr` function to process command-line arguments and displays the resulting hash in a user-friendly hexadecimal format.

This user space implementation provides a complete and self-contained SHA-256 hashing capability within XV6, allowing users to compute hashes of arbitrary input strings through a simple command-line interface. The modular design of the implementation allows for easy maintenance and potential future enhancements while maintaining compatibility with the XV6 operating system environment. The program that is responsible for this is called usha.c and is present in the user space (Listing 4).

```
int main(int argc, char *argv[]) {
    char *res = givemestr(argc, argv);//
        extracts string from command line
    char *hashStr = "";
    hashStr = SHA256(res);
    printf("SHA-256 hash of '%s': %s\n", res,
        hashStr);

    return 0;
}
```

Listing 4. Main function of the usha.c file

B. System Call

The system call implementation of SHA-256 in XV6 represents an intermediate approach between user space and kernel space implementations, leveraging the operating system's privilege mechanisms to provide hashing functionality through kernel services. This implementation demonstrates the integration of cryptographic operations within the operating system's system call interface.

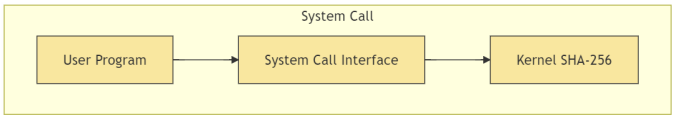

Fig. 4. Flow of a system call

*1) **Implementation Structure**:* The implementation spans multiple components of the XV6 kernel infrastructure, requiring modifications to core system files and the addition of new functionality. The SHA-256 algorithm implementation resides in `sha.c` within the kernel directory, providing the cryptographic operations at the kernel level. The integration with XV6's system call mechanism is achieved through careful modification of the system's call table and related structures. For the user to be able to call the system call, the system call hash has been added to the usys.pl file and the user.h file.

*2) **Kernel Integration**:* The integration process begins with the addition of the hash system call declaration in `syscall.h`, establishing the system call's presence within the kernel's namespace. This declaration enables the system call to be recognized and accessed throughout the kernel's execution environment. The system call table in `syscall.c` is extended to include the mapping between the hash system

call identifier and its implementation function, establishing the bridge between user requests and kernel functionality.

The core implementation of the system call resides in `sysproc.c`, where the interface between user space and kernel space is managed. This implementation handles the transition of data from user space to kernel space, ensures proper memory management, and coordinates the execution of the SHA-256 algorithm within the kernel's privileged context.

*3) **Memory Management**:* Memory management in the system call implementation requires careful attention to the boundary between user and kernel space. The implementation utilizes `argaddr` to safely fetch string arguments from user space, ensuring that memory access violations are prevented and that user-provided data is properly validated before processing.

The implementation maintains memory safety through proper allocation and deallocation practices. Memory resources are managed using the kernel's memory allocation facilities, with explicit cleanup operations using `kfree` to return resources to the system's free-page list. This careful memory management ensures that the system call implementation does not introduce memory leaks or resource exhaustion vulnerabilities.

*4) **Build System Integration**:* The integration of the SHA-256 functionality into the kernel required modifications to the build system. The `Makefile` was updated to include `sha.o` in the kernel's compilation process, ensuring that the SHA-256 implementation is properly built and linked with the kernel. Header files, including `sha.h`, were created to provide appropriate function declarations and ensure proper compilation across the kernel's source files.

*5) **System Call Interface**:* The system call provides a straightforward interface for user programs to access the SHA-256 functionality. User programs can invoke the hash system call with a string argument, and the implementation handles the computation and output of the hash value. The system call maintains the expected behavior of XV6 system calls, properly handling error conditions and returning appropriate status codes to the calling process.

This system call implementation demonstrates the integration of cryptographic functionality within the operating system's privileged execution environment while maintaining the security and stability requirements of kernel-level code. The implementation serves as an educational example of proper system call development practices within the XV6 environment.

C. Kernel Space

The kernel space implementation of SHA-256 in XV6 represents the deepest level of integration, embedding the hashing functionality directly within the kernel's initialization process. This implementation demonstrates how cryptographic operations can be integrated into the core operating system functionality, operating at the highest privilege level during system startup.

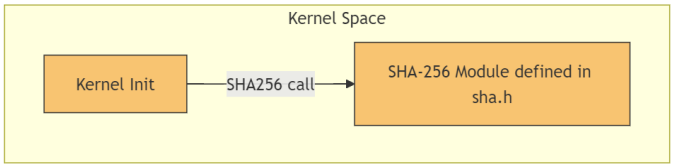

Fig. 5. Flow of hashing at start up in kernel

*1) **Implementation Structure**:* The kernel space implementation integrates the SHA-256 functionality directly into the kernel's initialization sequence through modifications to `main.c`. This approach differs fundamentally from both the user space and system call implementations by making the hashing capabilities an intrinsic part of the kernel's startup process. The implementation requires careful consideration of the kernel's initialization sequence and proper handling of kernel-level resources.

*2) **Kernel Integration**:* Integration begins with the inclusion of the SHA-256 functionality through `sha256.h` in `main.c`. The implementation leverages the kernel's initialization process to perform hashing operations during system startup. This integration is specifically designed to execute when the master core (identified by CPU ID 0) performs its initialization sequence, ensuring that the hashing operation occurs exactly once during the system's boot process.

*3) **Execution Context**:* The kernel space implementation operates in a privileged context distinct from both user space and system call implementations. By executing during kernel initialization, the implementation has direct access to kernel resources and operates without the need for context switches or privilege level transitions. This direct execution in kernel space provides maximum efficiency but requires careful handling of kernel resources and state.

*4) **Resource Management**:* Memory management in the kernel space implementation must adhere to strict kernel-level resource handling practices. The implementation utilizes kernel memory allocation functions directly, with explicit deallocation through `kfree` to maintain proper resource management. This direct management of kernel resources requires careful attention to prevent memory leaks that could persist throughout the system's runtime.

*5) **Architectural Differences**:* The kernel space implementation presents several key architectural differences from the system call implementation. While the system call implementation creates a bridge between user space and kernel space through a dedicated hash system call, the kernel space implementation operates solely within kernel space, bypassing the need for system calls or user interaction.

The implementation executes in the early stages of kernel initialization, leveraging the highest privilege level to perform its work before other processes are allowed to run. This approach ensures that the hashing operation occurs as part of the system's secure startup sequence.

*6) **Challenges and Benefits**:* The primary challenge of the kernel space implementation lies in its early execution during the boot process. The system must be carefully designed to handle potential errors or failures that could occur before the kernel has fully initialized. Additionally, the need to operate with kernel-level privileges requires strict attention to security and resource management.

However, the benefits of the kernel space implementation are clear. By integrating directly with the kernel, the SHA-256 hashing functionality becomes an intrinsic part of the system's core infrastructure, ensuring its availability at all times during system operation. Furthermore, executing in kernel space provides maximum performance and minimizes the overhead typically associated with context switching between user and kernel spaces.

*7) **Build System Integration**:* The kernel space implementation was seamlessly integrated into the kernel's build system through modifications to the `Makefile`. As with other kernel components, the `sha256.o` object file is compiled and linked into the kernel binary during the build process. This ensures that the SHA-256 implementation is present during kernel initialization and is available as soon as the kernel begins execution.

The build system modifications also ensure that the necessary kernel headers, such as `sha256.h`, are included and that the kernel code can reference the hashing functionality during startup. This integration reflects the close coupling of the SHA-256 implementation with the kernel's initialization and early execution processes.

IV. Performance Evaluation

The SHA-256 algorithm presents a fascinating study in computational complexity, with distinct characteristics in both time and space requirements that reflect its cryptographic design principles. This analysis examines the algorithm's performance characteristics across its core computational stages.

A. Time Complexity

The time complexity of the SHA-256 algorithm is fundamentally consistent across different input sizes, exhibiting a linear relationship with the input length. For an input of length n bits, the algorithm demonstrates $O(n)$ time complexity. This linear characteristic stems from the algorithm's structured processing of input data in fixed-size blocks of 512 bits.

The computational process can be broken down into the following key stages, each contributing to the overall time complexity:

1) **Preprocessing:** This involves padding the input message to ensure it meets the 512-bit block requirement. The process operates in $O(n)$ time, where n represents the input length. Padding adds a single '1' bit, followed by '0' bits, and concludes with a 64-bit representation of the original message length.
2) **Message Schedule Creation:** The initial 16 32-bit words are transformed into 64 words using a series of bitwise and modular operations. This expansion process occurs in constant time for each block, contributing $O(1)$ complexity per 512-bit block of input.
3) **Core Compression Function:** This is the most computationally intensive stage, performing a fixed number of 64 rounds of cryptographic transformations. Each round involves constant-time operations, including bitwise, rotate, and modular addition operations, maintaining $O(1)$ complexity per input block.

B. Space Complexity

The space complexity of SHA-256 is notably consistent and memory-efficient, requiring $O(1)$ additional space. The algorithm utilizes a fixed-size state regardless of input length, allowing it to process inputs of arbitrary size without exponential memory growth.

The implementation maintains several key data structures:

1) A 64-byte input buffer for block processing.
2) Eight 32-bit state variables.
3) Auxiliary computation variables.
4) A 64-element constant array for round constants.

```
typedef struct {
    uchar data[64];
    uint datalen;
    uint bitlen[2];
    uint state[8];
} SHA256_CTX;

uint k[64] = {
    0x428a2f98,0x71374491,
    0xb5c0fbcf,0xe9b5dba5,
    ...
};
```

Listing 5. Data structures used for the algorithm

These structures consume a constant amount of memory, independent of the input size. Incremental processing avoids the need to store the entire input in memory simultaneously.

C. Algorithm Scaling Analysis

To evaluate scaling, we used the implementation of an atomic instruction that reads the control and status register and returns the raw cycles elapsed of the CPU that was defined in `r_time()` function defined in `riscv.h`:

```
asm volatile("csrr %0, time" : "=r" (time));
```

Listing 6. The assembly instruction responsible for giving time

We measured the time required to compute hashes by invoking the assembly instruction before and after 1000 hash computations, and then took the time obtained after subtracting the start time from the end time. This process was repeated for

the kernel space program, user space program, and system call implementation.

```
uint64 start;
uint64 end;

asm volatile("csrr %0, time" : "=r" (start) );
for(int i=0;i<1000;i++){
    hashStr = SHA256(result);
}
asm volatile("csrr %0, time" : "=r" (end) );
printf("Time in user space: %lu\n", (end-start)
    );
printf("SHA-256 hash of '%s': %s\n", result,
    hashStr);
```

Listing 7. Code used for analysing time

Initial testing in user space revealed the algorithm's scaling characteristics with different input sizes. The implementation was tested with SHA cycle counts ranging from 1 to 10 loops. The results demonstrated a linear relationship between the number of cycles and execution time, with each additional SHA cycle consistently adding between 20,000 and 30,000 machine cycles. A graphical analysis confirmed this predictable scaling behavior across 10 input sizes (Fig.6 on next page).

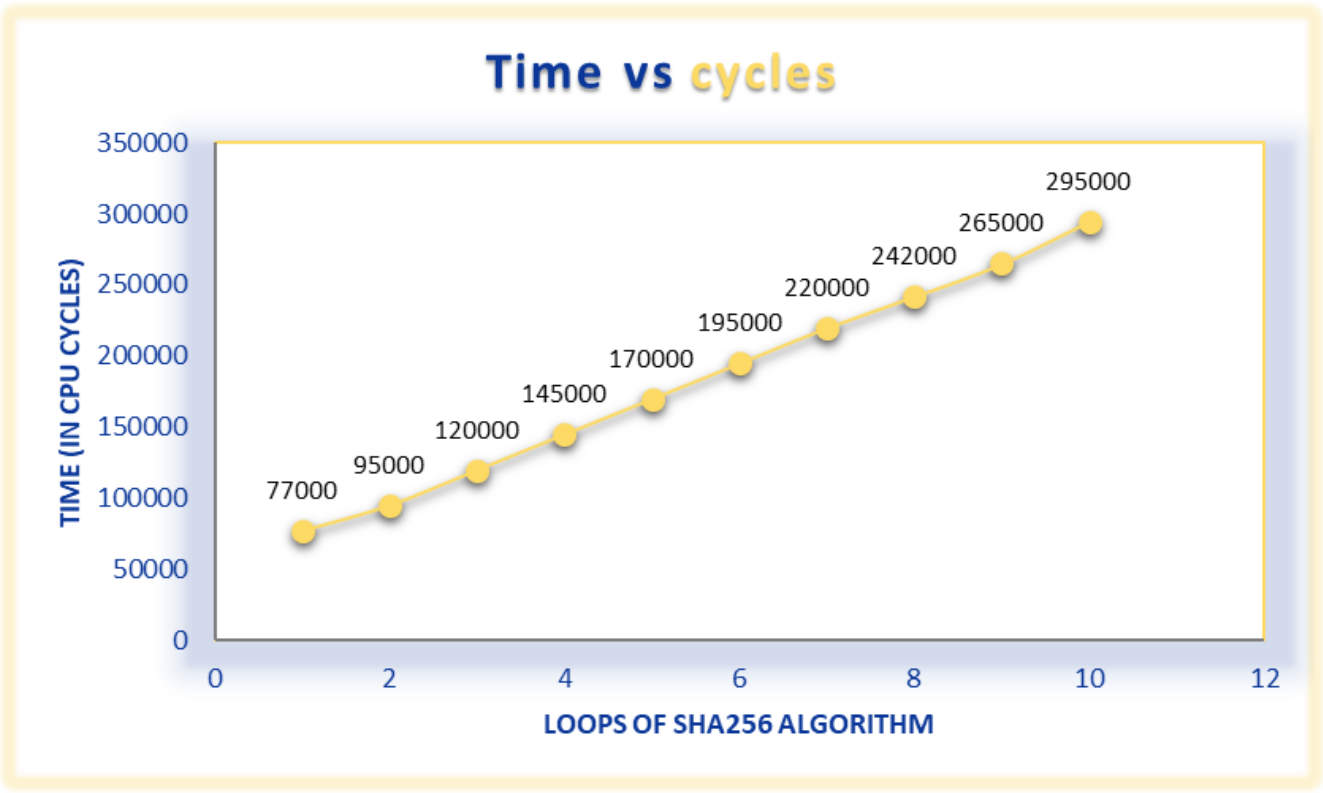

Fig. 6. **Findings from time analysis: CPU cycles vs Loops in algorithm**

D. Cross-Environment Performance Comparison

We evaluated the execution time across the three different implementations of the algorithm:

```c
int main(int argc, char *argv[]) {
    char *result = "hello";
    char *hashStr = "";
    uint64 start;
    uint64 end;

    //Measuring time for 1000 user function calls
    asm volatile("csrr %0, time" : "=r" (start) );
    for(int i=0;i<1000;i++){
        hashStr = SHA256(result);
    }
    asm volatile("csrr %0, time" : "=r" (end) );
    printf("Time in user space: %lu\n", (end-start));

    //Measuring time for 1000 system calls
    asm volatile("csrr %0, time" : "=r" (start) );
    for(int i=0;i<1000;i++) hash(result);
    asm volatile("csrr %0, time" : "=r" (end) );
    printf("Time in system call: %lu\n", (end-start));

    return 0;
}
```

Listing 8. Implementation in user space and through a system call

```
$ usha h
Time in user space: 81045
SHA-256 hash of 'hello': 2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824
Time in system call: 665347
$ usha h
Time in user space: 64026
SHA-256 hash of 'hello': 2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824
Time in system call: 646224
$ usha h
Time in user space: 93883
SHA-256 hash of 'hello': 2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824
Time in system call: 657360
$ usha h
Time in user space: 65033
SHA-256 hash of 'hello': 2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824
Time in system call: 667542
$ usha h
Time in user space: 77401
SHA-256 hash of 'hello': 2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824
Time in system call: 664242
$ usha h
Time in user space: 65988
SHA-256 hash of 'hello': 2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824
Time in system call: 744026
```

Fig. 7. Sample times from user and system call implementation

```c
void main()
{
  if(cpuid() == 0){
    consoleinit();
    ...
    uint64 start;
    uint64 end;
    asm volatile("csrr %0, time" : "=r" (start) );
    for(int i=0;i<1000;i++) SHA256("hello");
    asm volatile("csrr %0, time" : "=r" (end));
    printf("Time in kernel space(startup): %lu\n", (end-start));
    ...
  }
```

Listing 9. Implementation in kernel space

TABLE I
PERFORMANCE COMPARISON ACROSS EXECUTION ENVIRONMENTS

Environment	Average Execution Time (Cycles)
User Space	70,000
Kernel Space	230,000
System Call	650,000

The user space implementation demonstrated the best performance among all environments, while the system call implementation exhibited the highest execution time due to significant system call overhead. The kernel space implementation showed intermediate performance, though it was unexpectedly slower than the user space implementation.

E. Analysis of Performance Disparities

The significant performance differences observed across environments can be attributed to several factors:

1) **System Call Overhead:** The system call implementation exhibited the highest execution time (650,000 cycles), due to:
   - Trap handling mechanisms.
   - Context switching between user and kernel modes.
   - Security and parameter validation processes.
   - Return procedure overhead.
2) **Unexpected User Space vs. Kernel Space Performance:** The kernel space implementation (200,000 cycles) performed slower than the user space implementation (70,000 cycles), which was unexpected. Potential reasons include:
   - Variations in core execution location affecting timing measurements.
   - Different cache behavior and memory access patterns between kernel and user space.
   - Varying interrupt handling mechanisms across execution contexts.

   Further investigation is required to definitively identify the primary factors contributing to this discrepancy.

V. SECURITY CONSIDERATIONS

The implementation of SHA-256 across different execution environments presents varying security implications that must be carefully considered. This analysis examines the security ramifications of implementing the hash function in both user space and kernel space environments, with particular attention to access control, vulnerability risks, and potential security breaches.

```
xv6 kernel is booting

Time in kernel space(startup): 220709
hart 1 starting
hart 2 starting
init: starting sh
```

Fig. 8. Sample time from kernel space

A. *User Space Implementation*

In the user space implementation, the SHA-256 algorithm operates within the confines of standard user privileges, providing inherent security benefits through process isolation. This environment naturally restricts access to system resources and memory segments, effectively containing any potential security breaches within the user space boundary. The isolation mechanism ensures that compromised processes cannot directly impact kernel operations or other user processes, significantly reducing the potential attack surface.

However, this implementation remains susceptible to user-level attacks and potential interference from other user processes, necessitating robust application-level input validation and security checks.

B. *Kernel Space Implementation*

The kernel space implementation, while offering enhanced performance capabilities, introduces more significant security considerations due to its privileged operation status. Operating at the kernel level provides the implementation with full system access and direct memory access capabilities, which, while beneficial for performance, substantially increases the potential impact of security vulnerabilities.

Any bugs or security flaws in kernel-level code can lead to system-wide failures or compromise, making this implementation particularly sensitive from a security perspective. The elevated privileges of kernel space operations also present increased risks for privilege escalation attacks, requiring exceptionally thorough input validation and security measures.

C. *Access Control and Vulnerability Management*

Access control considerations differ significantly between these implementations. The user space version benefits from operating system-provided access controls and naturally implements the principle of least privilege, reducing the risk of privilege escalation. Conversely, the kernel space implementation requires careful permission management and must implement robust access controls to prevent misuse of elevated privileges.

Vulnerability management also varies between the two environments. While user space vulnerabilities typically remain contained within the process scope, kernel space vulnerabilities can have system-wide implications. This distinction affects both the severity of potential security breaches and the approaches required for mitigation. Regular security audits and testing are crucial for both implementations, but the kernel space version requires additional scrutiny due to its potential for system-wide impact.

D. *Best Practices and Recommendations*

When considering security best practices, the user space implementation generally offers advantages in terms of isolation and containment, making it easier to update, patch, and maintain from a security perspective. The kernel space implementation, while potentially offering better performance, carries greater security responsibilities and requires more complex security implementations.

For optimal security, it is recommended to implement SHA-256 in user space unless kernel-level access is specifically required by the application requirements. This approach minimizes potential security risks while maintaining adequate functionality for most use cases. Regardless of the chosen implementation environment, robust input validation, secure memory handling practices, and regular security testing should be maintained as standard practice. Furthermore, proper documentation, regular code reviews, and comprehensive incident response planning should be established to ensure long-term security maintenance.

E. *Conclusion*

These security considerations highlight the importance of carefully evaluating the implementation environment for cryptographic functions like SHA-256, where the choice between user space and kernel space implementation can significantly impact the overall security posture of the system. The decision should be based on a thorough assessment of security requirements, performance needs, and risk tolerance levels specific to the application context.

VI. Proposed Research Directions

One promising direction for future research involves implementing kernel integrity verification during the `XV6` startup process. This could be achieved through a hash-based integrity checking mechanism that operates in two phases.

In the first phase, hash values of critical kernel files would be pre-computed and stored in a secure location within the filesystem. During kernel startup, these files would be hashed again, and the results compared against the stored values. This approach would provide a robust method for detecting unauthorized modifications to kernel files, enhancing the overall security of the operating system.

The implementation would require careful consideration of:

- Performance impacts during boot time.
- Secure storage of the reference hash values.

By integrating this mechanism, `XV6` could serve as a testbed for exploring modern kernel integrity verification techniques, providing insights that could be applied to more complex operating systems.

VII. Conclusion

The implementation and analysis of the SHA-256 algorithm across various execution environments—user space, kernel space, and system calls—highlight the intricate balance between performance, security, and design trade-offs in operating systems. While the user space implementation demonstrates superior performance and inherent security benefits through process isolation, the kernel space and system call implementations offer unique challenges and opportunities for optimization and secure design.

The performance evaluation reveals a clear hierarchy in execution efficiency, with user space outperforming kernel space and system calls due to reduced overhead. However, the increased security responsibilities in kernel space and

system calls underscore the importance of robust security mechanisms, including careful privilege management, input validation, and vulnerability mitigation.

Security considerations further emphasize the significance of tailoring implementation choices to specific use cases, balancing the trade-offs between performance gains and potential risks. Kernel space operations, while powerful, require exceptional care to prevent system-wide vulnerabilities.

Finally, proposed future research directions, such as implementing hash-based kernel integrity verification during the `XV6` startup process, underscore the potential for improving operating system security. This approach not only provides a method for detecting unauthorized modifications but also positions `XV6` as a platform for exploring innovative security mechanisms.

In conclusion, the study of SHA-256 implementations and their environments offers valuable insights into the broader challenges and opportunities in operating system design. By addressing performance and security considerations holistically, developers can create systems that are both efficient and resilient against emerging threats.