Implementation and Analysis of SHA-256 in xv6 for RISC-V

Huzaifa Ahmed Khan
Department of Computer Science
Institute of Business Administration
Karachi, Pakistan
h.khan.26485@khi.iba.edu.pk

Muhammad Bilal Adnan
Department of Computer Science
Institute of Business Administration
Karachi, Pakistan
m.adnan.27151@khi.iba.edu.pk

***Abstract*—This project explores the implementation and analysis of the SHA-256 cryptographic hash function in three environments: kernel space, user space, and system calls, using xv6 on the RISC-V architecture. The project aimed to evaluate the differences in ease of implementation, execution, and performance in each environment while analyzing security implications. The development utilized Ubuntu for the build and testing environment, with QEMU as the RISC-V emulator. Detailed benchmarking and security analysis provide insights into the trade-offs of cryptographic operations in these environments.**

I. Introdcution

Setting Up the Development Environment

- Kernel Space:
  - Environment: xv6 (RISC-V)
  - Tools: GCC cross complier, QEMU, and GDB for xv6 kernel development
  - Setup: Modified the xv6 kernel to include custom functionality.
- User Space:
  - Environment: RISC-V user-space utilities
- System Call Integration:
  - Setup: Added a custom system call to the xv6 kernel. Updated the syscall table, created a new kernel-space function.

II. Implementation

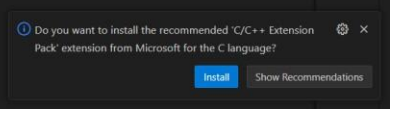
A. Kernel Space Implementation

Added a custom sha256.c file containing the implementation of the SHA-256 algorithm and a corresponding header file sha256.h. These files encapsulated the logic for message padding, scheduling, and the compression loop for SHA-256. Updated the Makefile to include the new files, ensuring they were compiled and linked into the kernel build.

Updates in the Makefile;
$K/sha256.o

```
$K/sha256.o
```

```c
static void sha256_block(struct sha256 *sha){
    uint32_t *state = sha->state;

    static const uint32_t k[8 * 8] = {
        0x428a2f98, 0x71374491, 0xb5c0fbcf, 0xe9b5dba5,
        0x3956c25b, 0x59f111f1, 0x923f82a4, 0xab1c5ed5,
        0xd807aa98, 0x12835b01, 0x243185be, 0x550c7dc3,
        0x72be5d74, 0x80deb1fe, 0x9bdc06a7, 0xc19bf174,
        0xe49b69c1, 0xefbe4786, 0x0fc19dc6, 0x240ca1cc,
        0x2de92c6f, 0x4a7484aa, 0x5cb0a9dc, 0x76f988da,
        0x983e5152, 0xa831c66d, 0xb00327c8, 0xbf597fc7,
        0xc6e00bf3, 0xd5a79147, 0x06ca6351, 0x14292967,
        0x27b70a85, 0x2e1b2138, 0x4d2c6dfc, 0x53380d13,
        0x650a7354, 0x766a0abb, 0x81c2c92e, 0x92722c85,
        0xa2bfe8a1, 0xa81a664b, 0xc24b8b70, 0xc76c51a3,
        0xd192e819, 0xd6990624, 0xf40e3585, 0x106aa070,
        0x19a4c116, 0x1e376c08, 0x2748774c, 0x34b0bcb5,
        0x391c0cb3, 0x4ed8aa4a, 0x5b9cca4f, 0x682e6ff3,
        0x748f82ee, 0x78a5636f, 0x84c87814, 0x8cc70208,
        0x90befffa, 0xa4506ceb, 0xbef9a3f7, 0xc67178f2,
    };

    uint32_t a = state[0];
    uint32_t b = state[1];
    uint32_t c = state[2];
    uint32_t d = state[3];
    uint32_t e = state[4];
    uint32_t f = state[5];
    uint32_t g = state[6];
    uint32_t h = state[7];

    uint32_t w[16];

    int i, j;
    for (i = 0; i < 64; i += 16){
        update_w(w, i, sha->buffer);

        for (j = 0; j < 16; j += 4){
            uint32_t temp;
            temp = h + step1(e, f, g) + k[i + j + 0] + w[j + 0];
            h = temp + d;
            d = temp + step2(a, b, c);
            temp = g + step1(h, e, f) + k[i + j + 1] + w[j + 1];
            g = temp + c;
            c = temp + step2(d, a, b);
            temp = f + step1(g, h, e) + k[i + j + 2] + w[j + 2];
            f = temp + b;
            b = temp + step2(c, d, a);
            temp = e + step1(f, g, h) + k[i + j + 3] + w[j + 3];
            e = temp + a;
            a = temp + step2(b, c, d);
        }
    }

    state[0] += a;
    state[1] += b;
    state[2] += c;
    state[3] += d;
    state[4] += e;
    state[5] += f;
    state[6] += g;
    state[7] += h;
}
```



sha256.c

```c
#ifndef SHA256_H
#define SHA256_H

#include <stddef.h>
#include <stdint.h>

#define SHA256_HEX_SIZE (64 + 1)
#define SHA256_BYTES_SIZE 32

/*
 * Compute the SHA-256 checksum of a memory region given a pointer and
 * the size of that memory region.
 * The output is a hexadecimal string of 65 characters.
 * The last character will be the null-character.
 */
void sha256_hex(const void *src, size_t n_bytes, char *dst_hex65);

void sha256_bytes(const void *src, size_t n_bytes, void *dst_bytes32);

typedef struct sha256 {
    uint32_t state[8];
    uint8_t buffer[64];
    uint64_t n_bits;
    uint8_t buffer_counter;
} sha256;

/* Functions to compute streaming SHA-256 checksums. */
void sha256_init(struct sha256 *sha);
void sha256_append(struct sha256 *sha, const void *data, size_t n_bytes);
void sha256_finalize_hex(struct sha256 *sha, char *dst_hex65);
void sha256_finalize_bytes(struct sha256 *sha, void *dst_bytes32);

#endif
```

sha256.h

B. User Space Implementation

It works similarly to the kernel space implementation by adding a custom sha256.c file containing the implementation of the SHA-256 algorithm and a corresponding header file sha256.h. These files encapsulated the logic for message padding, scheduling, and the compression loop for SHA-256. The only change exists while updating the Makefile to include the new files, ensuring they are compiled and linked into the user build.

Updates in the Makefile;
$U/_sha256\

```
$U/_sha256\
```

C. System Call Implementation

User space;

- The changes are made as follows first we created a hello.c file in user space, which demonstrates the use of the custom hash256 system call. The program defines a string ("HELLLLLOOO") and computes its SHA-256 hash using the hash256 system call. The hash is then printed out in hexadecimal format.

```c
#include "kernel/types.h"
#include "kernel/stat.h"
#include "user/user.h"

int main(void) {
    printf("hello world!\n");
    //printf("my sys calls: %d\n", sysCallCount());
    const char *text = "HELLLLLOOO";

    char hex[65];

    hash256(text, hex);

    printf("THE SHA-256 SUM OF \"%s\" is:\n\n", text);
    printf("%s\n\n", hex);

    exit(0);
}
```

- Also updated user.h to include the declaration for the hash256 system call.

```c
int hash256(const char* inpt, char* output_hash);
```

- Modified the usys.pl file to generate the user-space wrapper for the hash256 system call by adding the following entry.

```
entry("hash256");
```

- Added the hello.c file to the user-space build by updating the Makefile with the following.

```
$U/_hello\
```

Kernel space;

- The changes are made as follows first we added a function, begin(), in main.c that computes the SHA-256 hash of a string and prints it.

```c
void begin(){
    /* Input text. */
    const char *text = "Hello, World!";

    /* Char array to store the hexadecimal SHA-256 string. */
    /* Must be 65 characters big (or larger). */
    /* The last character will be the null-character. */
    char hex[SHA256_HEX_SIZE];

    /* Compute SHA-256 sum. */
    sha256_hex(text, strlen(text), hex);
    /* Print result. */
    printf("The SHA-256 sum of \"%s\" is:\n\n", text);
    printf("%s\n\n", hex);
}
```

- Also updated the syscall.c by adding a declaration of the sys_hash256 function and also registered the new system call by adding sys_hash256 to the syscall table.

```c
extern uint64 sys_hash256(void);
[SYS_hash256] sys_hash256
```

- Defined a new system call constant for hash256 in the syscall.h file.

```c
#define SYS_hash256 22
```

- In sysproc.c we have implemented the sys_hash256 system call. This function takes a string as an input, computes its SHA-256 hash using the sha256_hex function and copying the resulting hash to the user space.

```c
#include "kernel/sha256.h"

uint64
sys_hash256(void)
{
  char inpt[1000];
  uint64 out_addr;
  char hash[SHA256_HEX_SIZE];

  if (argstr(0, inpt, 1000) < 0)
    return -1;

  argaddr(1, &out_addr);

  sha256_hex(inpt, strlen(inpt), hash);

  if(copyout(myproc()->pagetable, out_addr, (char*)hash, SHA256_HEX_SIZE) < 0)
    return -1;

  return 0;
}
```

How does it all work together?

In user space, the program (hello.c) calls the hash256 function, which is a wrapper for the system call. The user provides the string input and a buffer (hex) to store the resulting hash. The system call hash256 is invoked to compute the hash. When the system call is invoked, the kernel checks the arguments: the string input and the address of the output buffer.The input string is passed to the sha256_hex function to compute the SHA-256 hash. The resulting hash is then copied back to the user space using the copyout function. The user space program calls hash256(), which eventually triggers the sys_hash256 system call in the kernel. The kernel processes the input, computes the hash, and returns the result to the user space. The user space program receives the hash and displays it.

III. Performance benchmarking

In the benchmarking set-up the following variables are considered:

Inputs: Strings of varying length and large files which are tested under different hashing scenarios.

Metrics: The time it takes to complete the hashing operation, measured in milliseconds.

Tools: Time measurement

Discussion of Results:

- User Space:
  - The user space implementation is the second fastest. This is because, in user space, cryptographic operations can be performed with some overhead and the need for switching between user and kernel contexts especially when handling larger inputs.
- Kernel Space:
  - Kernel space is the fastest as it provides the advantage of security and control over system resources, it doesn't need to introduce any additional context switching and the kernel's involvement in the hashing process. Direct access to optimized algorithms allows for efficient hashing.
- System Call:
  - The system call implementation is the slowest across both small inputs and large files. This is primarily due to the added time required for the user-kernel boundary crossing. Every time a system call is invoked, the program transitions from user mode to kernel mode and back, which adds latency. The impact of this overhead becomes more apparent with larger datasets or repeated function calls.

IV. Security considerations

Kernel Space

Pros:

- Operates in a secure environment, isolated from user applications.
- Minimal exposure to vulnerabilities like memory leaks

Cons:

- Any bug could compromise the entire system.
- Limited flexibility for non-administrative users.

User Space

Pros:

- Easier to develop and debug.
- Errors are contained within the user process.

Cons:

- Vulnerable to library-specific exploits.
- Less secure in multi-user environments.

System Call

Pros:

- Leverages kernel security for the hashing operation.
- Controlled exposure of kernel functionality.

Cons:

- Increased attack surface due to added syscall.
- Performance impact from boundary crossing.

V. Challenges faced

While implementing and testing SHA-256 in the xv6 operating system, several challenges emerged, particularly around managing large inputs, efficient memory use, and ensuring smooth interaction between user and kernel spaces. Below are the key challenges encountered in user mode, kernel mode, and during system call execution.

A. User Mode

- Handling Large Strings: xv6 allocates buffers in 4KB pages, so strings larger than this size could not be directly processed. To address this, a chunking method was used, breaking the input into 4KB segments. Each segment was sequentially appended to the SHA-256 context using sha_append, and once all chunks were processed, the final hash was computed with sha_finalize_hex.

- File Hashing: For file hashing, the read system call was used to load file data into memory in 4KB or 3KB chunks. Each chunk was appended to the SHA-256 context. After processing all chunks, the final hash was computed and converted to hexadecimal. Special care was required for handling the partial chunks at the end of a file.

Solutions: The chunking strategy allowed for the processing of both strings and files of arbitrary sizes while staying within xv6's buffer limits. This approach enabled efficient memory usage while ensuring accurate hash computation in user mode.

B. Kernel Mode

- Handling Large Strings: Like user mode, kernel space had to process large strings in chunks due to the 4KB buffer limit. Each chunk was sequentially appended to the SHA-256 context,

and after all chunks were processed, the final hash was computed.

- File Hashing: Without access to user-space system calls, the kernel had to use xv6's inode structures to read file data in chunks, requiring careful navigation through the file system. Edge cases, such as files smaller than the chunk size or buffers that weren't filled, added complexity.

- Copying Data to User Space**:** Directly writing the computed hash to a user-space address wasn't allowed due to security restrictions. To handle this, the memout function was used to securely transfer the hash from kernel space to the user-space output.

Solutions: Using xv6's inode structures for file reading and the memout function for secure data transfer helped the kernel implementation stay within xv6's constraints while maintaining reliable hashing functionality.

C. System Calls

- Secure Data Transfer: Transferring data from user space to kernel space required careful use of functions like copyin, while copying the computed hash back to user space involved memout. Ensuring data integrity and preventing buffer overflows or memory corruption during these transfers was a critical challenge.

- Handling Large Inputs: Large strings required the system call to process data in 4KB chunks, appending each chunk to the SHA-256 context in the kernel. Tracking chunk processing progress and handling edge cases like incomplete chunks or malformed strings were essential considerations.

- File Hashing via System Calls: File hashing via system calls meant managing file reading in the kernel using inode structures, which added complexity compared to user-space file reading. Ensuring that the full file was read and processed correctly required thorough implementation and testing.

Solutions: By employing chunking and securely transferring data with copyin and memout, the system call implementation ensured both reliability and security. Extensive testing across various input cases validated the functionality.

D. Overall Challenges

Across all implementations, several common challenges required coordinated efforts to resolve:

- Memory Constraints: xv6's limited buffer size necessitated the use of chunking for both strings and file data, which added complexity to the implementation.
- Integration Efforts: Integrating SHA-256 functionality into both user and kernel spaces while maintaining compatibility with xv6's existing structure demanded careful attention to detail.

- Testing and Debugging: Verifying the output against known test vectors and debugging in the low-level xv6 environment proved challenging due to the lack of modern debugging tools.

CLOSING COMMENTS

In conclusion, our project successfully implemented a SHA-256 hashing program within xv6 for the RISC-V architecture. The program's functionality and accuracy highlight xv6's potential as a platform for educational and lightweight application development. Future work could focus on integrating more complex cryptographic operations, enhancing user interface features, and optimizing performance for large datasets. Additionally, exploring multi-threaded implementations and hardware acceleration for cryptographic tasks could further enhance the system's capabilities.

REFERENCES

[1] Lions, J. Commentary on UNIX 6th Edition.

[2] xv6 Documentation: https://pdos.csail.mit.edu/6.1810/

[3] RISC-V Toolchain: https://github.com/riscv/riscv-gnu-toolchain

[4] National Institute of Standards and Technology (NIST): FIPS PUB 180-4 (Secure Hash Standard).

[5] [1]"SHA-256 and SHA-3," GeeksforGeeks, Mar. 20, 2024. https://www.geeksforgeeks.org/sha-256-and-sha-3/