RSAB
Final Report

Jaykar Nayeck
Emily Pakulski
Adam Incera
Noah Stebbins

Overview

For our project, we built an RSA Cryptographic accelerator. We covered all the components of RSA encryption, including key generation, encryption, and decryption, which are explained in depth below.

Generating Keys

Below are the steps to generate the private and public keys, \texttt{d} and \texttt{e}, respectively.

- We generate two distinct 64-bit prime numbers: p and q in software. To do this we do “focused" (i.e. rapid) random guessing, explained in the subsequent section. We then quickly verify if this is prime using a Miller-Rabin Primality Test.
- We then multiply the prime numbers, yielding n ($n = pq$). We use a built-in multiplier in SystemVerilog to do this. Then we need to multiply p-1 and q-1, to compute phi; to do this, we just use the same multiplier we used to compute n.
- Next, we find a positive integer that is less than $(p-1)(q-1)$ and is coprime with $(p-1)(q-1)$, a value called e. We hardcode this value to be 65537. This does not affect the overall security of our system as this is not part of the private key.
- Finally, we determine d, the multiplicative inverse of $e(mod((p-1)(q-1))$. This is done efficiently using the extended Euclid’s algorithm, which is coded in software.

As the end-user communicating with the FPGA, you are aware of all other people's (e, n) but d is kept secret.

Encrypting Messages

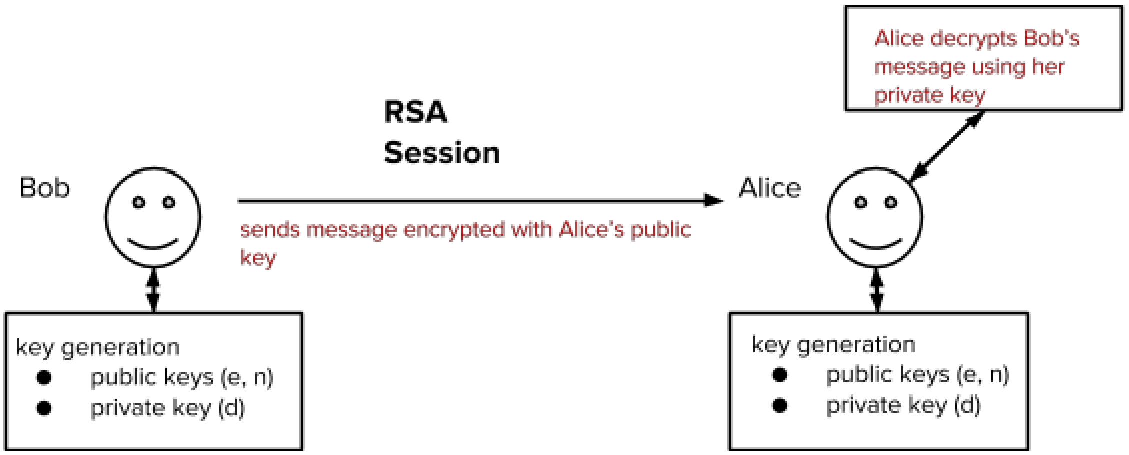
To encrypt a message m, another person would compute c, the ciphertext, using the (e, n) values published by means of the operation $c = m^e(mod(n))$. This operation is computed efficiently using modular exponentiation. We used a modified (improved) version of Blakley's algorithm for modular exponentiation. We determined this to be the most expensive operation of the entire encryption scheme, so we implemented this in hardware.

Decrypting Messages

To decrypt a message c, the person would compute $m = c^d(mod(n))$. This is also done efficiently using modular exponentiation, using the same technique mentioned above.

Putting it All Together

Public keys unique to a user (e & n) are shared across all users to allow encryption with a specific user in mind. The private key (d) is used along with the n value to decrypt the message.



Project Design

Software

In software, we computed the components of RSA that were not the costliest, namely key generation, since this only needs to be computed one time. We relegated the implementation of modular exponentiation to hardware, since it needs to be used every time we encrypt or decrypt a message.

Prime Number Generation

The first step of RSA encryption is to generate two large prime numbers: p and q. To generate prime numbers, we initially used random number generation with added optimizations to generate two 64 bit primes. First you randomly produce a number close to 2^{64} in magnitude, and test for primality using Miller-Rabin probabilistic primality test. If not prime, try a slightly smaller number slightly more times (*linear backoff*). The motivation behind this is you generate two large primes relatively quickly. We implemented this and found it produced numbers slightly larger than 2^{32}, which was far too small.

Thus, to keep RSA secure and prevent the prime numbers from being guessed correctly, we simply hardcoded an array of 50 prime numbers.

Extended Euclid's Algorithm

We used Extended Euclid's algorithm to compute the private key. The costliest component of RSA is encryption/decryption, not the actual key generation, so we chose to implement this in software.

We implemented Extended Euclid's algorithm in Python, due to its ability to handle large numbers easily. To use Python with C, we created a pipe within the C for parent and child (standard one way communication). The child executes (`execlp`) the Python program with five 32-bit arguments (1 for e = 32 bits and 4 for d = 128 bits). The Python consolidates these inputs, computes a large number output and sends it as four ints back to the C program.

The C program then uses these four ints (combined together to generate d), for decryption.

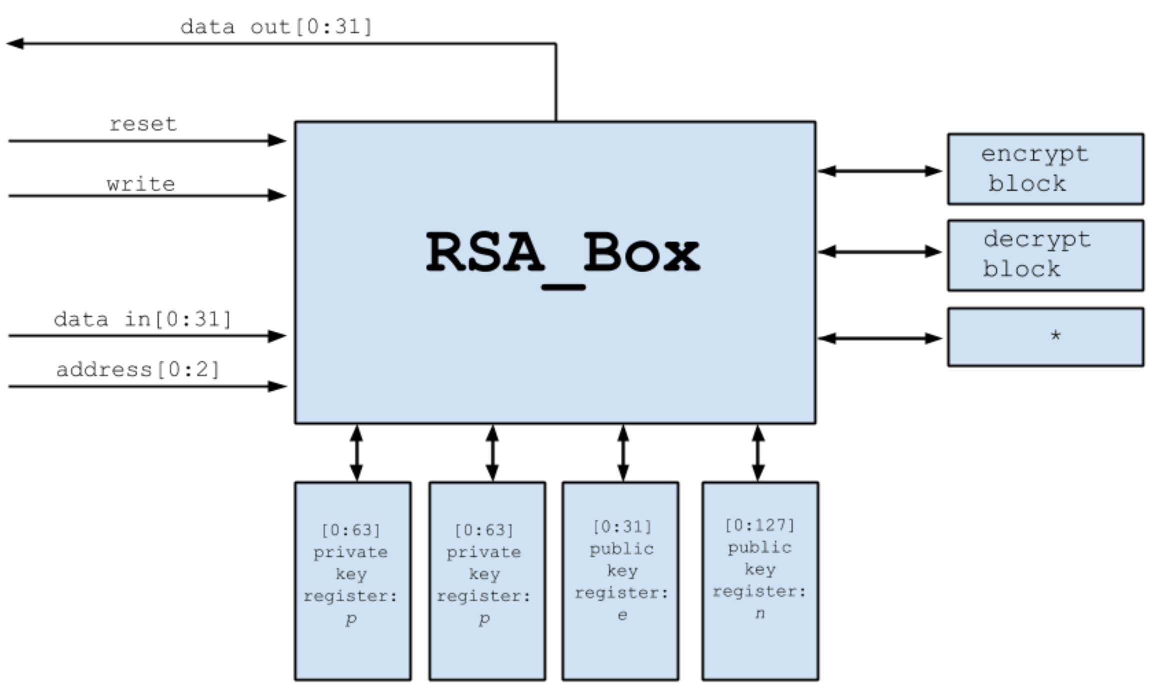
In a nutshell, we repeatedly apply division between two numbers to determine the GCD of two integers. We do this until the remainder is 0, and look at the last remainder computed. Below is the pseudocode, detailing the Extended Euclidean algorithm.

```
function inverse(a, p)
    t := 0;     newt := 1;
    r := p;     newr := a;
    while newr ≠ 0
        quotient := r div newr
        (r, newr) := (newr, r - quotient * newr)
        (t, newt) := (newt, t - quotient * newt)
    if degree(r) > 0 then
        return "Either p is not irreducible or a is a multiple of p"
    return (1/r) * t
```

source: *http://en.wikipedia.org/wiki/Extended_Euclidean_algorithm#Pseudocode*

Hardware

We implemented the costliest part of RSA in hardware, namely the encryption/decryption. Our revised design of the RSA box is included below.



Modulo Multiplication

We implemented modular multiplication, and modular exponentiation. We used Blakley's algorithm to compute modular multiplication, and a form of binary exponentiation to compute the modular exponentiation.

Blakley's algorithm is as follows:

```
Given input (a, b, n), output a*b mod (n)
r = 0;
for each bit in a:
      if(a = =1):
            r = r<<1 + b
      if(a== 0)
            r = r<<1
      if(r>= n)
            r -= n
      if(r>= n)
            r-=n
```

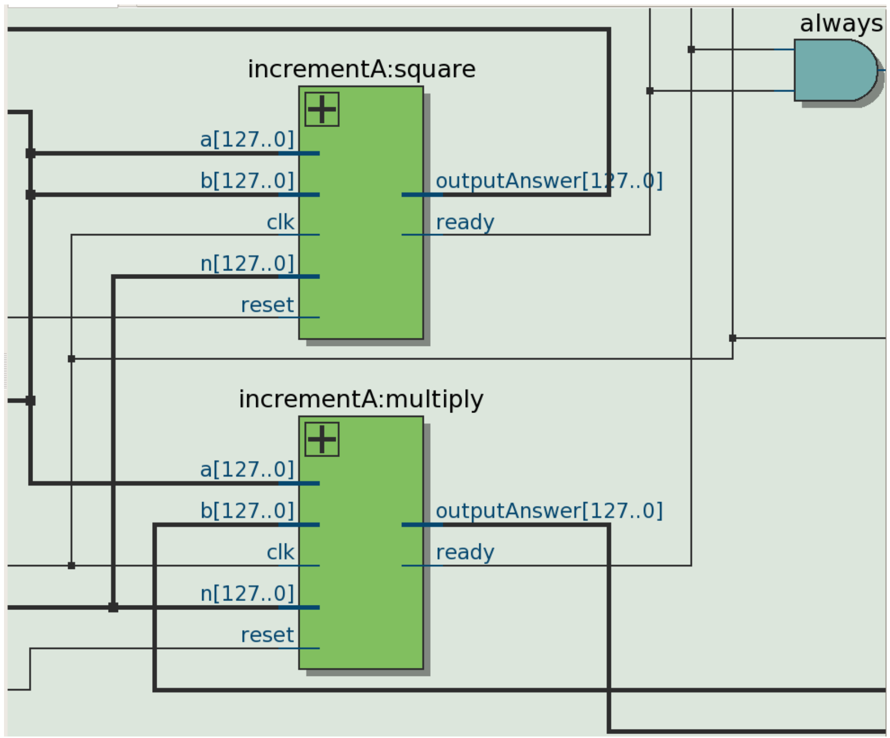
Our first iteration of the modular multiplication block was very simple. There was a mutex choosing the bit of a, and *step* variables that enforced the current step of the iteration. I.e step 1 corresponded choosing the bit of a, and step 2 was updating r, and step 3 was one of the if statements. Overall we were doing 6 clock cycles per bit of a. This was reduced to 2 cycles per bit as describe as bellow.

We realized that we could pipeline this algorithm. In every step, r could take on 2 possible values, either `r << 1`, or `r << 1 + b`. Also, after discussing with Professor Edwards, we realized that the sequential subtraction steps could be expanded into choosing between r, r-n, r-2*n. This could be done in parallel. Combining these two ideas, we made a circuit that computes the 6 possible cases of r, and in the next clock cycle chooses the correct r and updates the bit of a.

Using this modulo multiplication we constructed modulo exponentiation.

Modulo Exponentiation

For modulo exponentiation, we created a separate module in the SystemVerilog code which called upon two modulo multiplication blocks. They are labelled as incrementA in the figure below.



For modulo exponentiation, at each iteration, it checks if the zeroth bit of the exponent is high than the result is set to the result times the base. And if regardless of that result, the base is squared at each iteration with the modulo multiplication. The reduction happens through the modulo multiplication block. Modulo exponentiation is further elaborated in our “Timing Design” section.

```
function modular_pow(base, exponent, modulus)
    Assert :: (modulus - 1) * (modulus - 1) does not overflow base
    result := 1
    base := base mod modulus
    while exponent > 0
        if (exponent mod 2 == 1):
           result := (result * base) mod modulus
        exponent := exponent >> 1
        base := (base * base) mod modulus
    return result
```

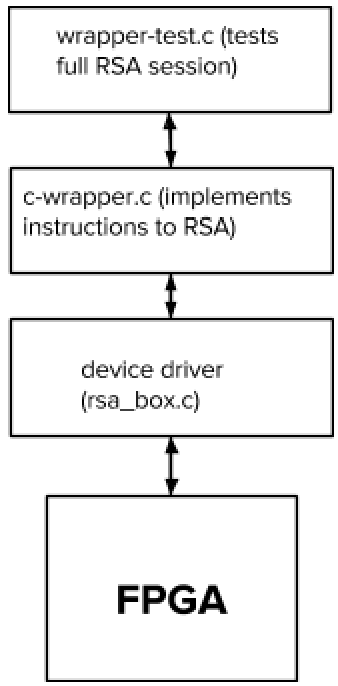
source: http://en.wikipedia.org/wiki/Modular_exponentiation

Software/Hardware Interface

Device Driver

We used the device driver created in lab III as a template for creating one for the RSA box. This is used by the wrapper (c-wrapper.c) for communicating to the FPGA. We handle I/O calls with vga_led_ioctl. When communicating with the FPGA, we set address (segment #) and data_in (corresponding data values).

The overall architecture, from wrapper test to the RSA box, is shown below. We also created a higher level, called c-interface.c, which abstracts away the implementation details from the wrapper. Thus, you can call encrypt with the ciphertext, the length, and a message buffer to write into and it will take care of the implementation for you.



Communication Protocol

We defined an ISA to streamline communication between the host machine and the FPGA. This is defined in `user-level/instructions.h`, shown below:

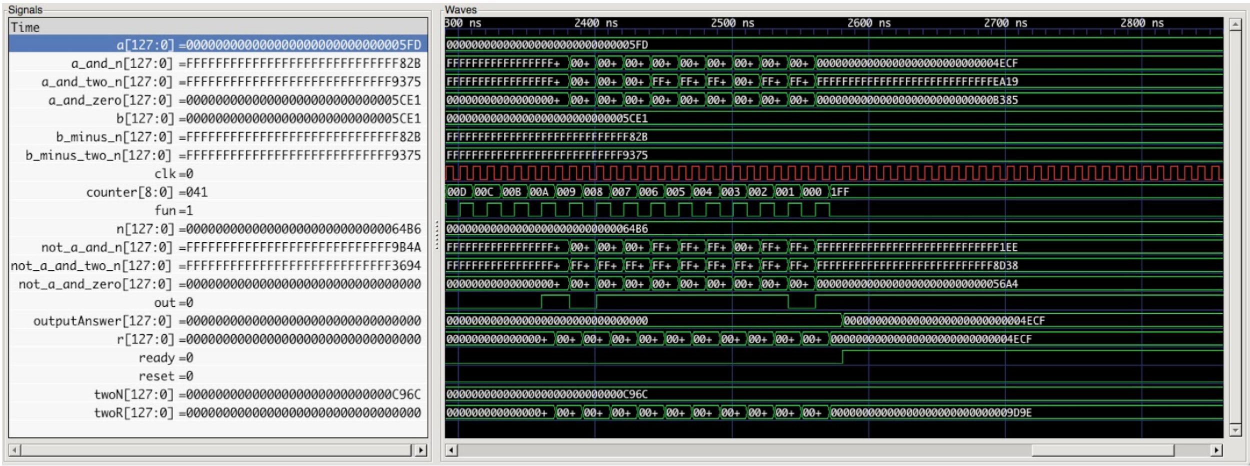
```
#define INSTRUCTION             0
#define RESET                   1
#define STORE_PUBLIC_KEY_1      2 // n
#define STORE_PUBLIC_KEY_2      3 // e
#define STORE_PRIVATE_KEY_1     4 // p
#define STORE_PRIVATE_KEY_2     5 // q
#define DECRYPT_BITS            6 // DECRYPT_3
#define ENCRYPT_BITS            7
#define READ_PUBLIC_KEY_1       8 // n
#define READ_PUBLIC_KEY_2       9 // e
#define STORE_MESSAGE           10 // m
#define STORE_MESSAGE2          11 // m
#define STORE_D                 12 // m
#define MAKE_OUR_N              13 // carry out p * q op
```

```
#define READ_OUR_N               14 // read back (p * q)
```

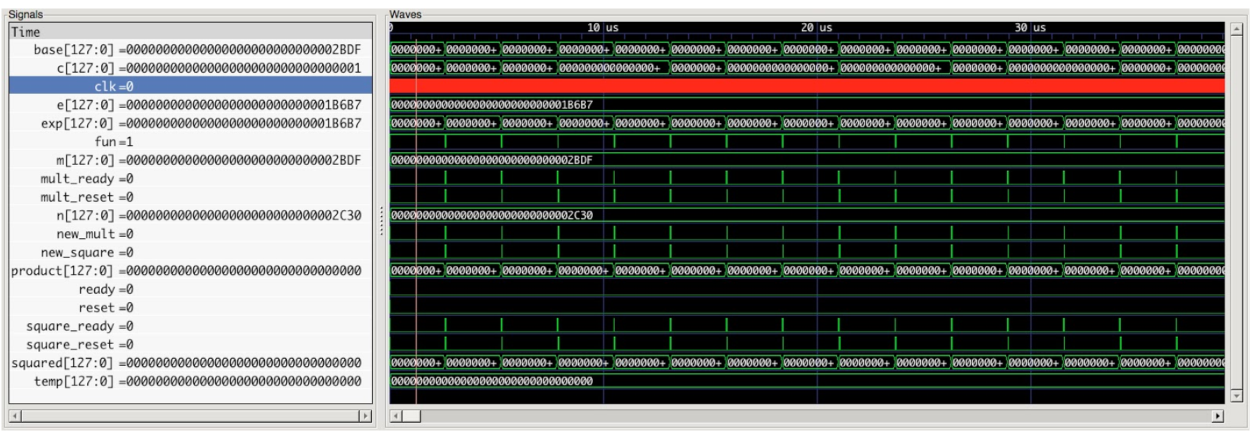
Sending INSTRUCTION tells the hardware that the next set(s) of signals sent will be an operation. Instructions 1-14 define these operations, ranging from storing different values to reading them to the actual encryption.

This is defined at the ISA level because they are highly low level operations. Many are not exposed in the user-facing C wrapper because some operations are only defined if they are called in a strict order. The C wrapper avoids many of the worst mis-orderings of operations.

Timing Design



This screenshot demonstrates the timing of the multiplication module. The



This screenshot from gtkwave demonstrates the timing of the modular exponentiation module. The mult_ready and square_ready signals spike each time the multiplication modules finish computing. This then triggers a spike in both the square_reset and mult_reset signals, indicating to the multiplication modules that they can safely read the inputs and begin the next iteration.

While the next iteration is being computed, the base is squared, and if the 0th bit is 1, the result is set to the product of itself and the base. When the exponent reaches 0, the ready bit is set to 1, indicating that the computation is complete.

General Issues

Hardware Level

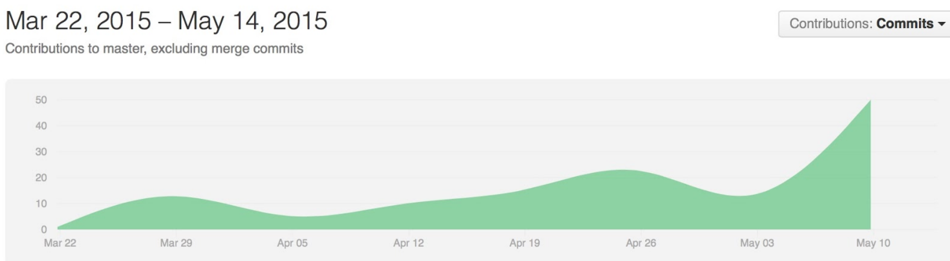
- The most challenging aspect was waiting for the code to compile, we didn't realize that we could use simulators until around milestone 2.
- We also forgot a polling bit, in the first iteration of the hardware/software interface so we had no guarantee on the correctness of the output. However, for our final project we are polling on a bit to see if the output is ready.
- It also took several iterations of rewriting blakley, and modulo exponentiation in order to be as efficient as possible. We realized that our circuit could be made much faster but adding some extra addition and subtraction blocks. There is of course the penalty of using more space, but because it the FPGA is a dedicated "RSA BOX", we wanted to make the 2 main operations as quick as possible.
- Our interface was also inconsistent with the hardware, and vice versa, so that lead to a lot of very hard-to-debug errors. However after completely rehauling the design in a different version, we standardized the functions and interface that the hardware and software would use.

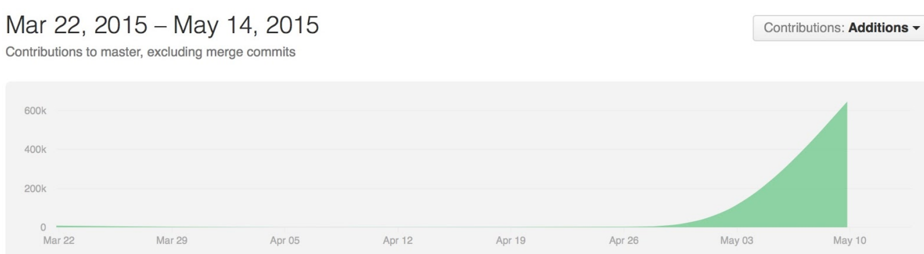
Software Level

- Programming software features on computers other than the lab ones posed compile-time issues when porting over to the CentOS machines. This was due to different versions of Python, different hardware support, etc.
- It was difficult to work with big (128-bit) numbers in C. We tried different big number library approaches, most of which were architecture independent.
- Fit multiple chars into each 128 bit
- We need to add randomized padding after speaking with Professor Lewko about how to best approach this
- We couldn't use the GMP or other popular big number packages because of the ARM processor. So we ended up using Python as our big number library, as integer operations are bounded by memory.

Lessons Learned & Advice for Future

It took us a while to get started on the hardware and device driver due to lack of familiarity. We should have taken the pre-reqs. Advanced Logic Design would have been helpful. This led to a rush of commits at the end, rather than evenly distributed work over the course of the semester. Below you can see the visualization of commits.





Next Steps

We tested all the functionality within one client on the terminal, that is we executed a wrapper-test. If we had additional time, we would have constructed a TCP/IP client on top of our current functionality so that two end users could communicate with encrypted messages. We would have different kinds of messages, including <share> messages, which you would use to give someone your public key so that they could encrypt a message to send to you.

Source Code

```c
/*
 * Device driver for the VGA LED Emulator
 *
 * A Platform device implemented using the misc subsystem
 *
 * Stephen A. Edwards
 * Columbia University
 *
 * Modified by: Emily Pakulski (enp2111)
 *
 * References:
 * Linux source: Documentation/driver-model/platform.txt
 *               drivers/misc/arm-charlcd.c
 * http://www.linuxforu.com/tag/linux-device-drivers/
 * http://free-electrons.com/docs/
 *
 * "make" to build
 * insmod rsa_box.ko
 *
 * Check code style with
 * checkpatch.pl --file --no-tree rsa_box.c
 */

#include <linux/module.h>
#include <linux/init.h>
#include <linux/errno.h>
#include <linux/version.h>
#include <linux/kernel.h>
#include <linux/platform_device.h>
#include <linux/miscdevice.h>
#include <linux/slab.h>
#include <linux/io.h>
#include <linux/of.h>
#include <linux/of_address.h>
#include <linux/fs.h>   /* struct file_operations */
#include <linux/uaccess.h>
#include "rsa_box.h"
#include <linux/types.h>

#define DRIVER_NAME "rsa_box"

/*
 * Information about our device
 */
struct vga_led_dev {
        struct resource res; /* Resource: our registers */
        void __iomem *virtbase; /* Where registers can be accessed in memory */
} dev;

/*
 * Write segments of a single digit
 * Assumes digit is in range and the device information has been set up
 */
static void write_digit(int address, u32 segments)
{
        iowrite32(segments, dev.virtbase + address * 4);
}

static u32 read_digit(int address)
{
        u32 answer;
        answer = ioread32(dev.virtbase + address * 4);
        return answer;
```

```
}

/*
 * Handle ioctl() calls from userspace:
 * Read or write the segments on single digits.
 * Note extensive error checking of arguments
 */
static long vga_led_ioctl(struct file *f, uint32_t cmd, unsigned long arg)
{
        rsa_box_arg_t vla;

        switch (cmd) {
        case RSA_BOX_WRITE_DIGIT:
                if (copy_from_user(&vla, (rsa_box_arg_t *) arg,
                                   sizeof(rsa_box_arg_t)))
                        return -EACCES;
                write_digit(vla.address, vla.data_in);
                break;

        case RSA_BOX_READ_DIGIT:
                if (copy_from_user(&vla, (rsa_box_arg_t *) arg,
                                   sizeof(rsa_box_arg_t)))
                        return -EACCES;
                vla.data_in = read_digit(vla.address);
                if (copy_to_user((rsa_box_arg_t *) arg, &vla,
                                 sizeof(rsa_box_arg_t)))
                        return -EACCES;
                break;

        default:
                return -EINVAL;
        }

        return 0;
}

/* The operations our device knows how to do */
// www.tdlp.org/LDP/lkmpg/2.4/html/c577.htm
static const struct file_operations vga_led_fops = {
        .owner          = THIS_MODULE,
        .unlocked_ioctl = vga_led_ioctl,
};

// file_operations holds pointers to functions defined by the driver that performs
// various operations on the device.
// Each field of the structure corresponds to the address of some function
// defined by the driver to handle a requested operation.

/* Information about our device for the "misc" framework -- like a char dev */
static struct miscdevice vga_led_misc_device = {
        .minor          = MISC_DYNAMIC_MINOR,
        .name           = DRIVER_NAME,
        .fops           = &vga_led_fops,
};

/*
 * Initialization code: get resources (registers) and display
 * a welcome message
 */
static int __init vga_led_probe(struct platform_device *pdev)
{
        int ret;

```

```
        /* Register ourselves as a misc device: creates /dev/rsa_box */
        ret = misc_register(&vga_led_misc_device);

        /* Get the address of our registers from the device tree */
        ret = of_address_to_resource(pdev->dev.of_node, 0, &dev.res);
        if (ret) {
                ret = -ENOENT;
                goto out_deregister;
        }

        /* Make sure we can use these registers */
        if (request_mem_region(dev.res.start, resource_size(&dev.res),
                               DRIVER_NAME) == NULL) {
                ret = -EBUSY;
                goto out_deregister;
        }

        /* Arrange access to our registers */
        dev.virtbase = of_iomap(pdev->dev.of_node, 0);
        if (dev.virtbase == NULL) {
                ret = -ENOMEM;
                goto out_release_mem_region;
        }

        return 0;

out_release_mem_region:
        release_mem_region(dev.res.start, resource_size(&dev.res));
out_deregister:
        misc_deregister(&vga_led_misc_device);
        return ret;
}

/* Clean-up code: release resources */
static int vga_led_remove(struct platform_device *pdev)
{
        iounmap(dev.virtbase);
        release_mem_region(dev.res.start, resource_size(&dev.res));
        misc_deregister(&vga_led_misc_device);
        return 0;
}

/* Which "compatible" string(s) to search for in the Device Tree */
#ifdef CONFIG_OF
static const struct of_device_id vga_led_of_match[] = {
        { .compatible = "altr,rsa_box" },
        {},
};
MODULE_DEVICE_TABLE(of, vga_led_of_match);
#endif

/* Information for registering ourselves as a "platform" driver */
static struct platform_driver vga_led_driver = {
        .driver = {
                .name   = DRIVER_NAME,
                .owner  = THIS_MODULE,
                .of_match_table = of_match_ptr(vga_led_of_match),
        },
        .remove = __exit_p(vga_led_remove),
};

/* Called when the module is loaded: set things up */
static int __init vga_led_init(void)
```

```
{
        pr_info(DRIVER_NAME ": init\n");
        return platform_driver_probe(&vga_led_driver, vga_led_probe);
}

/* Called when the module is unloaded: release resources */
static void __exit vga_led_exit(void)
{
        platform_driver_unregister(&vga_led_driver);
        pr_info(DRIVER_NAME ": exit\n");
}

module_init(vga_led_init);
module_exit(vga_led_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("RSA Box Team - Columbia University (based on code by Professor Stephen Edwards at Columbia)");
MODULE_DESCRIPTION("RSA Box - hardware RSA implementation device driver");
```

```
#ifndef _RSA_BOX_H
#define _RSA_BOX_H

#include <linux/ioctl.h>
#include <linux/types.h>     /* for int32_t */

typedef struct {
  int address;
  uint32_t data_in;
} rsa_box_arg_t;

#define RSA_BOX_MAGIC 'q'

/* ioctls and their arguments */
#define RSA_BOX_WRITE_DIGIT _IOW(RSA_BOX_MAGIC, 1, rsa_box_arg_t *)
#define RSA_BOX_READ_DIGIT  _IOWR(RSA_BOX_MAGIC, 2, rsa_box_arg_t *)

#endif
```

```
/* verilator lint_off WIDTH */
/* verilator lint_off UNUSED */
/*
* Code for RSA Box, a hardware implementation of the RSA algorithm.
 */

 module RSA_BOX(input logic      clk,
     input logic             reset,
     input logic             write,
     input                   chipselect,
     input logic[31:0]       data_in, // the current 32 bit input
     input logic[2:0]        address, // which 32 bit segment of each structure to write into
     output logic[31:0]      data_out
//                output        logic                             is_ready
    );


    /* instruction bits (can pick from instructions defined in user-level/instructions.h) */
    logic[31:0] instrBits;
    /* structures/registers */
    logic[127:0] outputBits;
    // private keys
    logic[63:0] p;
    logic[63:0] q;
    logic[127:0] m;
    // public keys
    logic [127:0] c;
    logic[127:0]        n; // p * q
    logic[31:0] e;
    /* enabler for ALU */
    logic[1:0] functionCall;
    logic ready;
    logic ready_for_encrypt;
    logic reset_exponent;

    logic reset_decrypt;
    logic[127:0] m1;
    logic [127:0] d;
    logic[127:0] decrypt_message;
    logic ready_for_decrypt;
         logic ready_for_read;
    logic[127:0] our_n;
         //assign reset_exponent = (reset | reset_exponent_signal);

         exponentiate encryptModule(.reset(reset_exponent), .clk(clk), .m(m), .e(e), .n(n), .c(c), .ready(ready_for_encrypt)) ;
         exponentiate decryptModule(.reset(reset_decrypt), .clk(clk), .m(m1), .e(d), .n(our_n), .c(decrypt_message), .ready(ready_for_decrypt)) ;

         always_ff @(posedge clk) begin
             if (reset || (address == 3'b000 && instrBits == 1'b1)) begin
                 /* reset triggered when clock starts */
                 data_out[31:0] <=              32'd0;
                 instrBits[31:0] <=             32'd0;     // reset typeof(instr)
                 p[63:0] <=                                      64'd0;
                 q[63:0] <=                                      64'd0;
                 n[127:0] <=                                     128'd0;
                 e[31:0] <=                                      32'd0;
                 m[127:0] <= 128'd0;
                 ready <= 1'b0;
                 d[127:0] <= 128'd0;
```

```
                m1[127:0] <= 128'd0;
                functionCall <= 2'b00;
                reset_exponent <= 1'b1;
                reset_decrypt <= 1'b1;
                our_n[127:0] <= 128'd0;
            end

            /* reading */
            if (chipselect && !write) begin
                case(functionCall)
                    2'b01: begin //encrypt
                        case (address)
                            3'b000: data_out[31:0] <= c[31:0];
                            3'b001: data_out[31:0] <= c[63:32];
                            3'b010: data_out[31:0] <= c[95:64];
                            3'b011: data_out[31:0] <= c[127:96];
                            3'b100: data_out[0] <= ready_for_encrypt;
                            default: begin end
                        endcase
                    end
                    2'b10: begin
                        case (address) //decrypt
                            3'b000: data_out[31:0] <= decrypt_message[31:0];
                            3'b001: data_out[31:0] <= decrypt_message[63:32];
                            3'b010: data_out[31:0] <= decrypt_message[95:64];
                            3'b011: data_out[31:0] <= decrypt_message[127:96];
                            3'b100: data_out[1] <= ready_for_decrypt;
                            default: begin end
                        endcase
                    end
                    2'b11: begin //multiply to read from n or n1 (depending on what's on output bits)
                                                                                                case (address)
                            3'b000: data_out[31:0] <= outputBits[31:0];
                            3'b001: data_out[31:0] <= outputBits[63:32];
                            3'b010: data_out[31:0] <= outputBits[95:64];
                            3'b011: data_out[31:0] <= outputBits[127:96];
                            3'b100: data_out[1] <= ready_for_read;
                            default: begin end
                        endcase
                    end
                    default: begin end
                endcase
            end
            /* writing */
            else if (chipselect && write) begin
                /* determine what kind of instruction this is */
                if (address == 3'b000) begin
                    instrBits[31:0] <= data_in[31:0];
                end

                /****** INSTRUCTIONS: check which each instruction *******/
                /* STORE_PUBLIC_KEY_1: n */
                case(instrBits)
                    32'd2: begin
                        case(address)
                            3'b001: n[31:0] <=        data_in[31:0];
                            3'b010: n[63:32] <=       data_in[31:0];
                            3'b011: n[95:64] <=       data_in[31:0];
                            3'b100: n[127:96] <= data_in[31:0];
                            default: begin end
                        endcase
```

```
                    end
                    32'd3: begin
                        /* STORE_PUBLIC_KEY_2: e */
                        case(address)
                            3'b001: begin
                                e[31:0] <=     data_in[31:0];
                            end
                            default: begin end
                        endcase
                    end
                    32'd4: begin
                        /* STORE_PRIVATE_KEY_1: p */
                        case(address)
                            3'b001: p[31:0] <=         data_in[31:0];
                            3'b010: p[63:32] <=        data_in[31:0];
                            default: begin end
                        endcase
                    end
                    32'd5: begin
                        /* STORE_PRIVATE_KEY_2: q */
                        case(address)
                            3'b001: q[31:0] <= data_in[31:0];
                            3'b010: q[63:32] <= data_in[31:0];
                            default: begin end
                        endcase
                    end

                    32'd6: begin
                        /* DECRYPT_BITS */
                        case(address)
                            3'b001: begin
                                functionCall[1:0] <= 2'b10;
                                reset_decrypt <= 1'b1;
                            end
                            3'b010: begin
                                functionCall[1:0] <= 2'b10;
                                reset_decrypt <= 1'b0;
                            end
                            default: begin end
                        endcase
                    end

                    32'd7: begin
                        /* ENCRYPT_BITS */
                        case(address)
                            3'b001: begin
                                functionCall[1:0] <= 2'b01;
                                reset_exponent <= 1'b1;
                            end
                            3'b010: begin
                                functionCall[1:0] <= 2'b01;
                                reset_exponent <= 1'b0;
                            end
                            default: begin end
                        endcase
                    end
                    32'd8: begin
                        /* READ_PUBLIC_KEY_1: n */
                        case (address)
                            3'b001: begin
    outputBits[127:0] <=    n[127:0];
```

```
ready_for_read <= 1'b1;

functionCall <= 2'b11;
                                                                                    end
                            default: begin end
                        endcase
                    end
                    32'd9: begin
                        /* READ_PUBLIC_KEY_2: e */
                        case(address)
                            3'b001: begin
                                outputBits[31:0] <= e[31:0];
                            end
                            default: begin end
                        endcase
                    end
                    32'd10: begin
                        /* STORE_MESSAGE: m*/
                        case (address)
                            3'b001: m[31:0] <=        data_in[31:0];
                            3'b010: m[63:32] <=       data_in[31:0];
                            3'b011: m[95:64] <=       data_in[31:0];
                            3'b100: m[127:96] <=      data_in[31:0];
                            default: begin end
                        endcase
                    end

                    32'd11: begin
                        /* STORE_MESSAGE: m1*/
                        case (address)
                            3'b001: m1[31:0] <=       data_in[31:0];
                            3'b010: m1[63:32] <=      data_in[31:0];
                            3'b011: m1[95:64] <=      data_in[31:0];
                            3'b100: m1[127:96] <=     data_in[31:0];
                            default: begin end
                        endcase
                    end


                    32'd12: begin
                        /* STORE D*/
                        case (address)
                            3'b001: d[31:0] <=        data_in[31:0];
                            3'b010: d[63:32] <=       data_in[31:0];
                            3'b011: d[95:64] <=       data_in[31:0];
                            3'b100: d[127:96] <=      data_in[31:0];
                            default: begin end
                        endcase
                    end


                    32'd13: begin
                        /* STORE D*/
                        case (address)
                            3'b001: our_n[127:0] <= p[63:0] * q[63:0];
                            default: begin end
                        endcase
                    end

                                                        32'd14: begin
                        /* READ_PUBLIC_KEY_1: n */
                        case (address)
                            3'b001: begin
```

```
                                                                        outputBits[127:0] <=         our_n[127:0];
                                                                        ready_for_read <= 1'b1;
                                                                        functionCall <= 2'b11;
                                                                        end
                              default: begin end
                          endcase
                      end

                      default: begin
                      end
                  endcase
              end // end for _writing_
          end // end always_ff
          endmodule




module exponentiate( input logic reset, clk,
    input logic[127:0] m,
    input logic[127:0] e,
    input logic[127:0] n,
    output logic[127:0] c,
    output logic ready
);

logic[127:0] base;
logic mult_ready;
logic square_ready;
logic fun;
logic[127:0] squared;
logic[127:0] product;
logic mult_reset;
logic square_reset;
logic new_mult;
logic new_square;
logic[127:0] temp;
logic[127:0] exp;

incrementA multiply(
    .reset      (mult_reset),
    .clk,
    .a          (base),
    .b          (c),
    .outputAnswer (product),
    .ready      (mult_ready),
    .n
);

incrementA square(
    .reset      (square_reset),
    .clk,
    .a          (base),
    .b          (base),
    .outputAnswer (squared),
    .ready      (square_ready),
    .n
);

assign mult_reset = (reset | new_mult);
```

```
assign square_reset = (reset | new_square);

always_ff @(posedge clk)
begin
    if(reset)
    begin
        ready <= 0;
        c <= 128'd1;
        base <= m;
        fun <= 1'b0;
        new_mult <= 0;
        new_square <= 0;
        exp <= e;
    end
    else if(exp > 32'b0) begin
        case(fun)
            1'b0: begin
                new_mult <= 1;
                new_square <= 1;
                if(!mult_ready & !square_ready)
                    fun <= 1'b1;
            end
            1'b1: begin
                new_mult <= 0;
                new_square <= 0;
                if(mult_ready & square_ready) begin
                    if(exp[0])
                        c <= product;
                    base <= squared;
                    fun <= 1'b0;
                    exp <= exp >> 1;
                end
            end
        endcase
    end
    else
        ready <= 1;
end
endmodule




/* verilator lint_off UNUSED */
/* verilator lint_off WIDTH */
/* verilator lint_off UNSIGNED */
module incrementA(input logic reset,
    input logic clk,
    input logic[127:0] a,
    input logic[127:0] b,
    input logic[127:0] n,
    output logic ready,
    output logic[127:0] outputAnswer
);

logic[8:0] counter;

logic[127:0] a_and_zero;
logic[127:0] not_a_and_zero;
logic[127:0] a_and_n;
logic[127:0] not_a_and_n;

logic[127:0] a_and_two_n;
```

```
logic[127:0] not_a_and_two_n;
logic[127:0] twoN;
logic fun;

logic[127:0] r;
logic[127:0] twoR;
logic[127:0] b_minus_n;
logic[127:0] b_minus_two_n;

logic out;

always_ff @(posedge clk)
begin
    if(reset)
    begin
        ready <= 0;
        counter[8:0] <= 9'd128;

        r[127:0] <= 6'd0;
        a_and_zero[127:0] <= 9'd0;
        not_a_and_zero[127:0] <= 9'd0;

        a_and_n[127:0] <= 9'd0;
        not_a_and_n[127:0] <= 9'd0;

        a_and_two_n[127:0] <= 9'd0;
        not_a_and_two_n[127:0] <= 9'd0;

        twoN[127:0] <= n[127:0]<<1;
        fun <= 1'b1;

        b_minus_n[127:0] <= b[127:0] - n[127:0];

    end
    else
    begin
        b_minus_two_n[127:0] <= b[127:0] - twoN[127:0];
        case(fun)
            1'b0: begin
                if($signed(counter) == -9'd1) begin
                    outputAnswer[127:0] <= r[127:0];
                    ready <= 1'b1;
                end
                else begin
                    out <= a[counter];
                    a_and_zero[127:0] <= (twoR[127:0] + b[127:0] );
                    not_a_and_zero[127:0] <= (twoR[127:0]);
                    a_and_n[127:0] <= (twoR[127:0] + b_minus_n[127:0]);
                    not_a_and_n[127:0] <= (twoR[127:0] - n[127:0]);
                    a_and_two_n[127:0] <= (twoR[127:0] + b_minus_two_n[127:0]);
                    not_a_and_two_n[127:0] <= (twoR[127:0] - twoN[127:0]);
                    fun <= 1'b1;
                end
            end
            1'b1: begin
                if($signed(counter) >= 9'd0) begin
                    counter <= $signed(counter) - 1'b1;
                    fun <= 1'b0;
                    case(out)
                        1'b0: begin
                            if($signed(not_a_and_zero[127:0]) >= 0 && not_a_and_zero[127:0]<n[127:0]) begin
                                r[127:0] <= not_a_and_zero[127:0];
```

```
                                twoR[127:0] <= not_a_and_zero[127:0] <<1;
                            end
                            else if($signed(not_a_and_n[127:0]) >= 0 && not_a_and_n[127:0]<n[127:0]) begin
                                r[127:0] <= not_a_and_n[127:0];
                                twoR[127:0] <= not_a_and_n[127:0] <<1;
                            end
                            else begin
                                r[127:0] <= not_a_and_two_n[127:0];
                                twoR[127:0] <= not_a_and_two_n[127:0] <<1;
                            end
                        end
                        1'b1: begin
                            if($signed(a_and_zero[127:0]) >= 0 && a_and_zero[127:0]<n[127:0]) begin
                                r[127:0] <= a_and_zero[127:0];
                                twoR[127:0] <= a_and_zero[127:0] <<1;
                            end
                            else if($signed(a_and_n[127:0]) >= 0 && a_and_n[127:0]<n[127:0]) begin
                                r[127:0] <= a_and_n[127:0];
                                twoR[127:0] <= a_and_n[127:0] <<1;
                            end
                            else begin
                                r[127:0] <= a_and_two_n[127:0];
                                twoR[127:0] <= a_and_two_n[127:0] <<1;
                            end
                        end
                    endcase
                end

            end

        endcase
    end
end
endmodule
```

```c
#ifndef __C_INTERFACE_H__
#define __C_INTERFACE_H__

#include <stdint.h>     /* for unit32_t */

/*
 * Set private keys to allow encrypting. Set public keys
 * to allow decrypting.
 */
void set_private_keys(int32_t *p, int32_t *q);
void set_public_keys(int32_t *e, int32_t *n);
void __read_public_keys(int32_t *e, int32_t *n);

// Encryption and decryption using values stored in registers.
// Raise exception and set errno if relevant register not set.
void encrypt(char *msg_buf, int32_t *cypher_buf, int len);
void decrypt(int32_t *cypher_buf, char *msg_buf, int len);

#endif
```

```
1: #ifndef __C_WRAPPER_H__
2: #define __C_WRAPPER_H__
3:
4: #define PRIVATE                 0
5: #define PUBLIC                  1
6: #define DECRYPT_SEND            0
7: #define ENCRYPT_SEND            1
8:
9: // comment or uncommon line 10 to add/remove debug print statements
10: #define PRINTVERBOSE            1
11:
12: /* store private keys, getting back public key */
13: void key_swap(int32_t *p, int32_t *q, int32_t *our_n);
14:
15: /* encrypt or decrypt */
16: void send_int_encrypt_decrypt(int action, int32_t *message_n, int32_t *output);
17:
18: /* read back value encrypted/decrypted */
19: void __read_encryption(int32_t *encryption);
20: void __read_decryption(int32_t *decryption);
21: void read_our_N(int32_t *n);
22:
23: /* helper functions */
24: void set_fd();
25: void print_128_bit_integer(int32_t *input_x);
26:
27: #endif
```

```c
#ifndef __EXTEUC_H__
#define __EXTEUC_H__

#include <stdint.h>     /* for unit32_t */

void err_sys(char *err);
void e_euclid(int32_t e, int32_t phi[4], int32_t *d);

#endif
```

```c
#ifndef __INSTRUCTIONS_H__
#define __INSTRUCTIONS_H__

/*
 * before writing any data, specify which instruction will be used:
 * write INSTRUCTION with desired action (e.g. MAKE_KEY, ENCRYPT, etc)
 */
#define INSTRUCTION             0
#define RESET                   1
#define STORE_PUBLIC_KEY_1      2 // n
#define STORE_PUBLIC_KEY_2      3 // e
#define STORE_PRIVATE_KEY_1     4 // p
#define STORE_PRIVATE_KEY_2     5 // q
#define DECRYPT_BITS            6 // DECRYPT_3
#define ENCRYPT_BITS            7
#define READ_PUBLIC_KEY_1       8 // n
#define READ_PUBLIC_KEY_2       9 // e
#define STORE_MESSAGE           10 // m
#define STORE_MESSAGE2          11 // m
#define STORE_D                 12 // m
#define MAKE_OUR_N              13 // carry out p * q op
#define READ_OUR_N              14 // read back (p * q)

void log_instruction(int opcode);

#endif
```

```
#ifndef _PRIMEGENERATOR_H_
#define _PRIMEGENERATOR_H_

#include <stdint.h>
#include <inttypes.h>

/* GNU C seeder */
unsigned long long rdtsc();
/* modular exponentiation */
uint64_t modulo(uint64_t base, uint64_t exponent, uint64_t mod);
/* Miller-Rabin Primality Test */
int miller(uint64_t p, int iteration);
uint64_t get_random(int tries);
uint64_t generate_prime();
void generate_prime_as_int32_t(int32_t *prime_64);

#endif
```

```c
#include <stdint.h>             /* for unit32_t */
#include <stdlib.h>             /* for malloc */
#include <stdio.h>              /* for printf */
#include <string.h>             /* for memcpy */
#include "c-interface.h"
#include "c-wrapper.h"          /* for all functions making syscalls */

#define TRUE    1
#define FALSE   0

void set_private_keys(int32_t *p, int32_t *q)
{
    store_keys(PRIVATE, p, q);
}

void set_public_keys(int32_t *e, int32_t *n)
{
    store_keys(PUBLIC, e, n);
}

void read_public_keys(int32_t *e, int32_t *n)
{
    __read_public_keys(e, n);
}

/*
 * encrypt message and return as 32-bit int array.
 */
void encrypt(char *msg_buf, int32_t *cypher_buf, int len)
{
    int i;
    int32_t curr_val;

    for (i = 0; i < len; i++)
    {
        memcpy(&curr_val, msg_buf + i, sizeof(int32_t));
       // send_int_encrypt_decrypt(ENCRYPT_SEND, &curr_val);
        memcpy(cypher_buf + i, &curr_val, sizeof(char));
    }
}

/*
 * decrypt cypher and return message as char array.
 */
void decrypt(int32_t *cypher_buf, char *msg_buf, int len)
{
    int i;
    int32_t curr_val;

    for (i = 0; i < len; i++)
    {
        memcpy(&curr_val, cypher_buf + i, sizeof(int32_t));
       // send_int_encrypt_decrypt(DECRYPT_SEND, &curr_val);
        memcpy(msg_buf + i, &curr_val, sizeof(char));
    }
}
```

```c
/*
 * Userspace program that communicates with the RSA_Box device driver
 * primarily through ioctls.
 *
 * Original VGA_LED code by Stephen A. Edwards, Columbia University
 */

#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <time.h>       /* for sleep() */
#include <stdint.h>     /* for unit32_t */
#include "../rsa_box.h"
#include "instructions.h"
#include "c-wrapper.h"
#include "exteuc.h"

void read_segment(int32_t *bit_output, int size);
void send_bits(int32_t *value, int count);
void __store_d(int32_t *d);
void store_keys(int type, int32_t *key_1, int32_t *key_2);

/* globals */
static int BIT_SEGMENTS[5] =  {1, 2, 3, 4, 5};
static int BIT_SEGMENTS_READ[5] = {0, 1, 2, 3, 4};
static int rsa_box_fd = -1;
static int empty[4] = {0, 0, 0, 0};

void set_fd()
{
    char *filename = "/dev/rsa_box";
    if ( (rsa_box_fd = open(filename, O_RDWR)) == -1)
    {
        fprintf(stderr, "could not open %s\n", filename);
    }
}

/*
 * Tells hardware what instruction to include the incoming
 * data with.
 */
void send_instruction(int operation)
{
    rsa_box_arg_t rsa_userspace_vals;
    if (rsa_box_fd == -1)
        set_fd();

    rsa_userspace_vals.address =  INSTRUCTION;
    rsa_userspace_vals.data_in = operation;

#ifdef PRINTVERBOSE
    log_instruction(operation);
#endif

    if (ioctl(rsa_box_fd, RSA_BOX_WRITE_DIGIT, &rsa_userspace_vals))
    {
        perror("ioctl(RSA_BOX_WRITE_DIGIT) failed");
    }
```

```c
}

/*
 * Sends count int32_t's to the hardware.
 * Always call send_instruction() first or the hardware won't know
 * what to do with the incoming data.
 */
void send_bits(int32_t *value, int count)
{
    rsa_box_arg_t rsa_userspace_vals;
    int i;

    if (rsa_box_fd == -1)
        set_fd();

    for (i = 0; i < count; i++)
    {
        rsa_userspace_vals.address = BIT_SEGMENTS[i];
        rsa_userspace_vals.data_in = value[i];

#ifdef PRINTVERBOSE
        printf("[sending] %d // %d\n", BIT_SEGMENTS[i], value[i]);
#endif

        if (ioctl(rsa_box_fd, RSA_BOX_WRITE_DIGIT, &rsa_userspace_vals))
        {
            perror("ioctl(RSA_BOX_WRITE_DIGIT) failed");
        }
    }

}

/*
 * Store private keys and get back our public key.
 */
void key_swap(int32_t *p, int32_t *q, int32_t *our_n)
{
    int32_t p_phi[2];
    int32_t q_phi[2];
    int32_t phi_n[4];
    int32_t d[4];

    // calculate p - 1, q - 1
    p_phi[0] = p[0] - 1;
    p_phi[1] = p[1];

    q_phi[0] = q[0] - 1;
    q_phi[1] = q[1];

    // store d, the extended euclid of (p - 1)(q - 1) and e
    store_keys(PRIVATE, p_phi, q_phi);
    read_our_N(phi_n);

    int32_t E = 65537;
    e_euclid(E, phi_n, d);
    __store_d(d);

    // store actual p and q
    store_keys(PRIVATE, p, q);
    read_our_N(our_n);
}

/*
```

```
 * Stores keys into the specified registers, PUBLIC or PRIVATE
 * key registers.
 */
void store_keys(int type, int32_t *key_1, int32_t *key_2)
{
    if (type == PRIVATE)
    {
        send_instruction(STORE_PRIVATE_KEY_1);
        send_bits(key_1, 2); // p
        send_instruction(STORE_PRIVATE_KEY_2);
        send_bits(key_2, 2); // q
    }

    if (type == PUBLIC)
    {
        send_instruction(STORE_PUBLIC_KEY_1);
        send_bits(key_1, 4); // n
        send_instruction(STORE_PUBLIC_KEY_2);
        send_bits(key_2, 1); // e
    }
}


void __store_d(int32_t *d)
{
    send_instruction(STORE_D);
    send_bits(d, 4);
}

/*
 * Writes input to m2, the cyphertext to be decrypted.
 */
void __send_cyphertext(int32_t *m)
{
    send_instruction(STORE_MESSAGE2);
    send_bits(m, 4);
}

/*
 * Send data to encrypt/decrypt to device.
 */
void send_int_encrypt_decrypt(int action, int32_t *input, int32_t *output)
{
    if (action == ENCRYPT_SEND)
    {
        send_instruction(STORE_MESSAGE);
        send_bits(input, 4); // cleartext, m
        __read_encryption(output);
    }

    if (action == DECRYPT_SEND)
    {
        __send_cyphertext(input);
        __read_decryption(output);
    }
}

/*
 * Return the public keys on this device. Encrypt data already stored
 * on board.
 *
 * (Note: the interface to read private keys was intentionally ommitted.
 */
```

```c
void __read_encryption(int32_t *encryption)
{
    int32_t valid[5] = {0,0,0,0,0};
    int i;
    send_instruction(ENCRYPT_BITS);
    send_bits(empty, 2);
    read_segment(valid, 5);

    while (valid[4] == 0)
    {
        read_segment(valid+4, 1);
    }

    read_segment(valid, 5);

    for (i = 0; i < 5; i++)
    {
        encryption[i] = valid[i];
    }
}

void __read_decryption(int32_t *decryption)
{
    int32_t valid[5] = {0, 0, 0, 0, 0};
    int i;

    send_instruction(DECRYPT_BITS);
    send_bits(empty, 2);
    read_segment(valid, 5);

    while (valid[4] == 0 || valid[4] == 1)
    {
        read_segment(valid + 4, 1);
    }

    read_segment(valid, 5);

    for (i = 0; i < 5; i++)
    {
        decryption[i] = valid[i];
    }

}

/*
 * Read "size" 32 bit segments into bit output.
 */
void read_segment(int32_t *bit_output, int size)
{
    rsa_box_arg_t rsa_userspace_vals;
    int i;

    if (rsa_box_fd == -1)
        set_fd();

    for (i = 0; i < size; i++)
    {
        rsa_userspace_vals.address = BIT_SEGMENTS_READ[i];

        if (ioctl(rsa_box_fd, RSA_BOX_READ_DIGIT, &rsa_userspace_vals))
        {
            perror("ioctl(RSA_BOX_READ_DIGIT) failed");
        }
```

```c

        bit_output[i] = rsa_userspace_vals.data_in;
#ifdef PRINTVERBOSE
        printf("[sending] %d // %d\n", BIT_SEGMENTS_READ[i], bit_output[i]);
#endif
    }
}

/*
 * Get the product of p and q.
 */
void read_our_N(int32_t *n)
{
    send_instruction(MAKE_OUR_N);
    send_bits(empty, 1);

    send_instruction(READ_OUR_N);
    send_bits(empty, 1);
    read_segment(n, 4);
}

/** Extended Euclid's implementation below **/

#include <string.h>
#include <sys/wait.h>

#define READ_BUF 4096

struct IntSet {
    int x[4];
};

void err_sys(char *err) {
    perror(err);
    exit(1);
}

void e_euclid(int32_t e, int32_t phi[4], int32_t *d)
{
    int phi1 = phi[3];
    int phi2 = phi[2];
    int phi3 = phi[1];
    int phi4 = phi[0];

    pid_t pid;
    int fd[2];

    if(pipe(fd) < 0) {
        err_sys("pipe error");
    }

    if((pid = fork()) < 0) {
        err_sys("fork error");
    }
    else if(pid > 0) { // parent
        close(fd[1]); // close write end

        if(fd[0] != STDIN_FILENO) { // set STDIN
            if(dup2(fd[0], STDIN_FILENO) != STDIN_FILENO) {
                err_sys("dup2 error");
            }
        }

```

```
        char buf[READ_BUF];
        if(read(STDIN_FILENO, buf, READ_BUF) < 0) {
            err_sys("read error");
        }

        // printf("[received]: %s\n", buf);

        struct IntSet my_s;

        /* parse buf */
        const char s[2] = " ";
        char *token = strtok(buf, s);
        int curr = 0;

        while(token != NULL && curr < 4) {
            my_s.x[curr] = atoi(token);
            printf("curr: %d, token: %s\n", curr, token);
            token = strtok(NULL, s);
            curr++;
        }

        if (waitpid(pid, NULL, 0) < 0)
            err_sys("waitpid error");

        d[0] = my_s.x[3];
        d[1] = my_s.x[2];
        d[2] = my_s.x[1];
        d[3] = my_s.x[0];
    }
    else { // child
        close(fd[0]); // close read end

        if(fd[1] != STDOUT_FILENO) { // set STDOUT
            if(dup2(fd[1], STDOUT_FILENO) != STDOUT_FILENO) {
                err_sys("dup2 error");
            }
        }

        char e_s[READ_BUF];

        char phi_s[READ_BUF];
        char phi2_s[READ_BUF];
        char phi3_s[READ_BUF];
        char phi4_s[READ_BUF];

        snprintf(e_s, READ_BUF, "%d\n", e);

        snprintf(phi_s, READ_BUF, "%d\n", phi1);
        snprintf(phi2_s, READ_BUF, "%d\n", phi2);
        snprintf(phi3_s, READ_BUF, "%d\n", phi3);
        snprintf(phi4_s, READ_BUF, "%d\n", phi4);

        printf("%s\n", e_s);

        printf("%s\n", phi_s);
        printf("%s\n", phi2_s);
        printf("%s\n", phi3_s);
        printf("%s\n", phi4_s);

        // execute Python script
        execlp("python", "python", "exteuc.py", e_s, phi_s, phi2_s, phi3_s, phi4_s, (ch
ar *)NULL);
    }
```

```
}

```

```
#include <stdio.h>
#include "instructions.h"

void log_instruction(int opcode)
{
    printf("[sending instruction] ");
    switch(opcode)
    {
        case RESET:
            printf("RESET");
            break;
        case STORE_PUBLIC_KEY_1:
            printf("STORE_PUBLIC_KEY_1");
            break;
        case STORE_PUBLIC_KEY_2:
            printf("STORE_PUBLIC_KEY_2");
            break;
        case DECRYPT_BITS:
            printf("DECRYPT_BITS");
            break;
        case ENCRYPT_BITS:
            printf("ENCRYPT_BITS");
            break;
        case READ_PUBLIC_KEY_1:
            printf("READ_PUBLIC_KEY_1");
            break;
        case READ_PUBLIC_KEY_2:
            printf("READ_PUBLIC_KEY_2");
            break;
        case STORE_MESSAGE:
            printf("STORE_MESSAGE");
            break;
        case STORE_MESSAGE2:
            printf("STORE_MESSAGE2");
            break;
        case STORE_D:
            printf("STORE_D");
            break;
        default:
            break;
   }
   printf("\n");
}
```

```
#include <stdlib.h>
#include "c-interface.h"

int main(int argc, char **argv)
{
    int32_t p[2];
    int32_t q[2];

    p[0] = 2;
    p[1] = 3;
    q[0] = 39;
    q[1] = 5000;

    set_private_keys(p, q);

    return 0;
}
```

```c
#include <stdio.h>
#include <stdlib.h>

#include <stdint.h>
#include <inttypes.h>

#include <time.h>

static uint64_t primes[50] = {
    (((uint64_t) 1) << 63) - 25,
    (((uint64_t) 1) << 63) - 165,
    (((uint64_t) 1) << 63) - 259,
    (((uint64_t) 1) << 63) - 301,
    (((uint64_t) 1) << 63) - 375,
    (((uint64_t) 1) << 63) - 387,
    (((uint64_t) 1) << 63) - 391,
    (((uint64_t) 1) << 63) - 409,
    (((uint64_t) 1) << 63) - 457,
    (((uint64_t) 1) << 63) - 471,

    (((uint64_t) 1) << 62) - 57,
    (((uint64_t) 1) << 62) - 87,
    (((uint64_t) 1) << 62) - 117,
    (((uint64_t) 1) << 62) - 143,
    (((uint64_t) 1) << 62) - 153,
    (((uint64_t) 1) << 62) - 167,
    (((uint64_t) 1) << 62) - 171,
    (((uint64_t) 1) << 62) - 195,
    (((uint64_t) 1) << 62) - 203,
    (((uint64_t) 1) << 62) - 273,

    (((uint64_t) 1) << 61) - 1,
    (((uint64_t) 1) << 61) - 31,
    (((uint64_t) 1) << 61) - 45,
    (((uint64_t) 1) << 61) - 229,
    (((uint64_t) 1) << 61) - 259,
    (((uint64_t) 1) << 61) - 283,
    (((uint64_t) 1) << 61) - 339,
    (((uint64_t) 1) << 61) - 391,
    (((uint64_t) 1) << 61) - 403,
    (((uint64_t) 1) << 61) - 465,

    (((uint64_t) 1) << 60) - 93,
    (((uint64_t) 1) << 60) - 107,
    (((uint64_t) 1) << 60) - 173,
    (((uint64_t) 1) << 60) - 179,
    (((uint64_t) 1) << 60) - 257,
    (((uint64_t) 1) << 60) - 279,
    (((uint64_t) 1) << 60) - 369,
    (((uint64_t) 1) << 60) - 395,
    (((uint64_t) 1) << 60) - 399,
    (((uint64_t) 1) << 60) - 453,

    (((uint64_t) 1) << 59) - 55,
    (((uint64_t) 1) << 59) - 99,
    (((uint64_t) 1) << 59) - 225,
    (((uint64_t) 1) << 59) - 427,
    (((uint64_t) 1) << 59) - 517,
    (((uint64_t) 1) << 59) - 607,
    (((uint64_t) 1) << 59) - 649,
    (((uint64_t) 1) << 59) - 687,
    (((uint64_t) 1) << 59) - 861,
    (((uint64_t) 1) << 59) - 871
```

```c
};

/* GNU C seeder: measures total pseudo-cycles since device on */
unsigned long long rdtsc(){
    unsigned int lo,hi;
    __asm__ __volatile__ ("rdtsc" : "=a" (lo), "=d" (hi));
    return ((unsigned long long)hi << 32) | lo;
}

/* modular exponentiation (base ^ exponent % mod) */
uint64_t modulo(uint64_t base, uint64_t exponent, uint64_t mod) {
    uint64_t x = 1; uint64_t y = base;

    while (exponent > 0) {
        if (exponent % 2 == 1) // odd exponents
            x = (x * y) % mod;
        y = (y * y) % mod;
        exponent = exponent / 2;
    }
    return x % mod;
}

/*
 * Miller-Rabin Primality Test, iteration = accuracy
 */
int miller(uint64_t p, int iteration) {
    int i;
    printf("%" PRIu64 "\n",p);

    if (p < 2) { return 0; }
    if (p != 2 && p % 2 == 0) { return 0; }

    uint64_t s = p - 1;
    while (s % 2 == 0) { s /= 2; }

    for (i = 0; i < iteration; i++) {

        uint64_t a = rand() % (p - 1) + 1, temp = s;
        uint64_t mod = modulo(a, temp, p);

        while (temp != p - 1 && mod != 1 && mod != p - 1) {
            mod = (mod * mod) % p;
            mod = (mod * mod) % p;
            temp *= 2;
        }

        if (mod != p - 1 && temp % 2 == 0) { return 0; }
    }
    return 1;
}

uint64_t get_random(int tries) {

    uint64_t r30 = (uint64_t)rand();    // top 30
    uint64_t s30 = (uint64_t)rand();    // middle 30
    uint64_t t4  = rand() & 0xf;                // bottom 4

    uint64_t res = (r30 << 34) + (s30 << 4) + t4;

    while(tries > 0) {
        if(res > (((uint64_t) 1) << 50))
            res >>= 1;
        tries--;
```

```c
    }

    return res;
}

uint64_t generate_prime() {

    int iteration = 5;
    int tries = 0; /* LINEAR BACKOFF */

    srand(rdtsc()); // randomize seed

    for(;;) {

        uint64_t num = 0x0LL; int j;
        for(j = 0; j <= tries && j <= 1000; j++) {
            // printf("tries: %d, j: %d\n", tries, j);
            num = get_random(tries);

            if(miller(num, iteration) == 1) { return num; }
        }
        tries++;
    }
    return -1;
}

uint64_t pick_prime() {
    int i;
    srand(rdtsc());
    i = rand() % 50;
    return primes[i];
}



#define BIT_MASK 0xffffffff

void generate_prime_as_int32_t(int32_t *prime_64)
{
    uint64_t prime = pick_prime();
    printf("%llX\n", prime);
    prime_64[0] = (int32_t) (prime % BIT_MASK);
    prime_64[1] = (int32_t) (prime >> 32);
}

int main() {
//    printf("%" PRIu64 "\n", generate_prime());
//    printf("%" PRIu64 "\n", generate_prime());
    int32_t prime_64[2];
    generate_prime_as_int32_t(prime_64);
    printf("%X %X", prime_64[0], prime_64[1]);

    return 0;
}
```

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <time.h>       /* for sleep() */
#include <stdint.h>     /* for unit32_t */
#include "../rsa_box.h"
#include "c-wrapper.h"
#include "instructions.h"

#include "prime-generator.h"

int rsa_box_fd;

// print out 128 bit int, but by [sections]
void print_128_bit_integer(int32_t *input_x)
{
    int i;

    for (i = 0; i < 4; i++)
        printf("quartile(%d): %u\n", i, input_x[i]);
}

/*
 * Return 1 if all size 32 bit numbers in the value are
 * equal; else return 0.
 */
int large_numbers_equal(int32_t *a, int32_t *b, int size)
{
    int i;
    for (i = 0; i < size; i++)
        if (a[i] != b[i]) return 0;

    return 1;
}

int main()
{
    /*
     * main tests
     */
    int32_t p[2];
    int32_t q[2];

    //int32_t n[4];
    int32_t n_our[4]; // our copy of n
    int32_t e_message[4];
    int32_t d_message[4];

    int32_t message[4]  = {13,0,0,0};

    printf("RSA Box device driver started\n");

    /* STORING PRIVATE KEYS, e.g. 23 and 17. */
    p[0] = 23;
    p[1] = 0;
    q[0] = 17;
    q[1] = 0;

```

```
    printf("\n[test case: storing private key...]\n");
    key_swap(p, q, n_our);

    printf("current value of n:");
    print_128_bit_integer(n_our);

    /* ENCRYPT/DECRYPT TEST */
    printf("Original message:");
    print_128_bit_integer(message);

    send_int_encrypt_decrypt(ENCRYPT_SEND, message, e_message);

    printf("Encrypted message:");
    print_128_bit_integer(e_message);

    send_int_encrypt_decrypt(DECRYPT_SEND, e_message, d_message);

    printf("Decrypted message (should match original):");
    print_128_bit_integer(d_message);


    return 0;
}
```