

## **Introduction**

The CSAW 2008 Hardware Challenge calls for a variety of malicious code to be added to the HDL of an encryption device. This HDL is a mix of Verilog and VHDL. The functions of the malicious code can range from DoS attacks, to revealing the encryption key, to disrupting the normal operations of the device, and ultimately breaching the security of the device. All this needs to be done transparent to the end user up until the point that the code takes effect. In other words, the device must still function as normal until the malicious code is activated. By definition, this code can be called a Trojan. We have tried to implement these Trojans in such a way that the size (chip utilization) of our final synthesized design matches the original given code. We also tried to ensure that the power utilization of the device stayed the same. These features allow the compromised code to pass by many known Trojan detection methods.

## **Normal Operation - Uncompromised Code**

Originally, the HDL is a mix of publicly available code, such as AES encryption, tied together by proprietary code, and programmed onto a BASYS FPGA board. When programmed, the board will interface with the user through the on-board VGA port, and PS/2 ports. A diagram of the

BASYS board is included for easy reference.

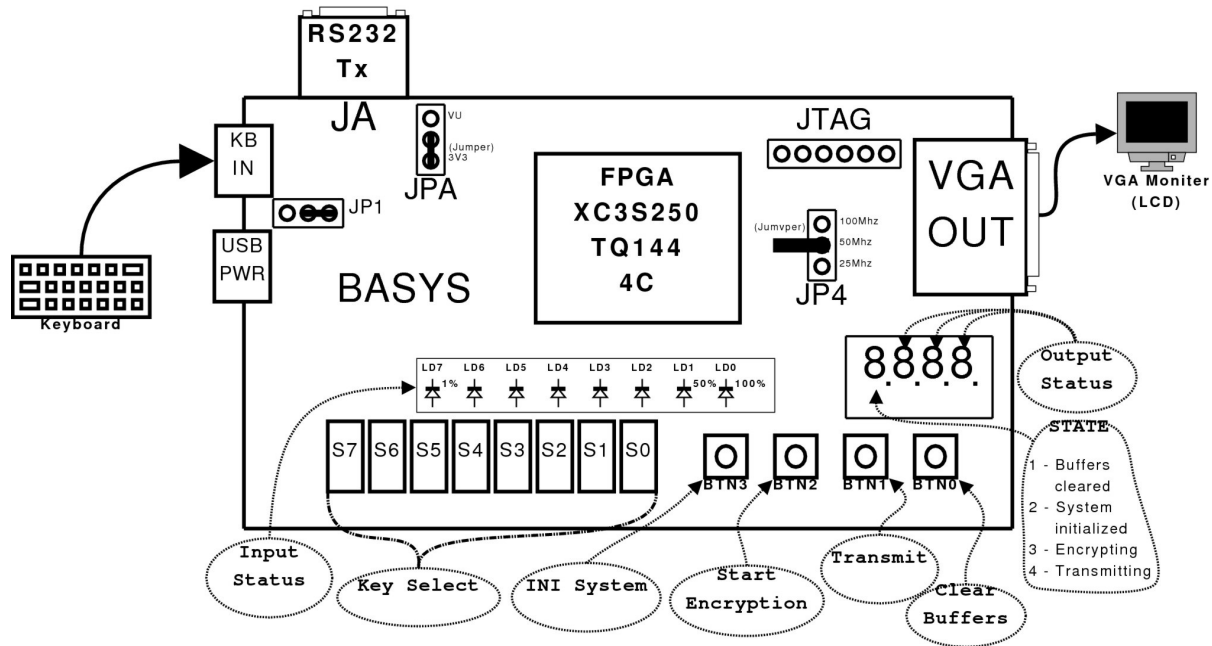


Diagram 1. BASYS board layout

Pressing Button#0 will initiate and reinitiate the board, aka “soft reset” it, and bring up the Alpha1.1 logo screen. Upon depressing Button#3, the screen will clear, leaving a cursor to prompt the user for plaintext. The key to be used for encryption is put in via the DIP switches on the BASYS board. The plaintext is input through a standard PS/2 keyboard connected to the board’s PS/2 port. When input is complete, the user will then press Button#2 to encrypt the plaintext. This function is entirely internal, meaning there will be no changes to the VGA or RS232 outputs. When the user is ready, pressing Button#3 again will output the AES- encrypted ciphertext through the RS232 serial port. This is the extent of the original functionality of the board. To encrypt more text, the user will have to press Button#0 again to reinitiate, or “soft reset” the board.

### Optimization - Freeing Up Resources for Additional “Functions”

After extensively analyzing the provided HDL package, we decided to first optimize a few areas to make available space to work with. One of the main places where we realized we can save

space was in the numerous places where extensive case statements are used, with entirely sequential cases. In HDL, case statements are implemented as a bank of registers, which is then multiplexed to 1 output. Since this is inefficient, we decided to re-implement these case statements as a ROM, whose outputs can be selected directly via the address line. In Verilog, we used "parameter", and in VHDL we used "constant" to achieve this. These changes made to pt\_exp.v and aes\_package.vhd net us enough free space on the chip to program in our Trojans. While the Trojans were implemented by expanding an output MUX of the aes128\_fast.vhd file to allow it to output desired data; and an implementation of a simple state machine that is activated by the sequences mentioned below. The complete and working files can be seen in the source file package included in the official submission.

## **Insertion Points - Security Vulnerabilities and Compromised Sources**

Our Trojans required the use of new signals and ports, therefore we have modified alphasop.v for proper signal routing. In addition to proper signal routing, alphasop.v also contains the "master key" for the encryption, which is needed since AES uses a 128-bit key, and the dip switches only set 8 bits. This is very insecure, because there will only be 256 possible keys. One of our Trojans takes advantage of this.

The bulk of the code for our Trojans is added to the aes128\_fast.vhd file. Amongst the modifications are the addition of a button, an input buffer, and a master key port. There are other pieces of code as well, which allow the function of our Trojans, and these changes can be seen in the source file package included in the official submission.

## **Trojan Behavior - Activation and Additional "Functions"**

Our compromised HDL code works in 3 "stages," with the higher-numbered stages having precedence once activated. In other words, the Trojan starts out in Stage 1, but once it gets activated to Stage 2, it can only go on to Stage 3 or stay in Stage 2. Stage 1 is basically the "dormant" stage, where the device will operate as if it were clean, uninfected, and uncompromised, as described above.

Stage 2 activation of our Trojan puts it into DoS (Denial of Service) mode. This effect is permanent until the board is completely reprogrammed. This takes advantage of the fact that the actual Flash PROM on the BASYS never gets programmed. Usually, when an FPGA board is programmed, the bit file gets written to the on-board Flash PROM, which then gets loaded when the board is turned on. FPGAs do not retain memory when they are powered off. It would depend on loading the program from the non-volatile Flash PROM during every power cycle. In this design, programming the board writes it directly to the FPGA's memory. When the board is

simply power cycled or “hard reset,” it will then have no PROM to read off of. As a result, the effects of our DoS Trojan remain until the board is reprogrammed with the bit file. This stage of our Trojan is activated by having both Button#1 and Button#2 depressed together when the board has received plaintext input. Under normal operation, Button#2 is pressed to encrypt the plaintext after it is entered. When this stage of our Trojan is activated, the plaintext will still be encrypted, but it will not be used. The values stored will be all 0’s on the first iteration of operation, and all subsequent iterations will yield stored values of “DEADBEEF” in hexadecimal. When the user goes to transmit the ciphertext, they will be unsuccessful, and transmit 0’s or “DEADBEEF” instead. The intention of the activation procedure is for the Trojan to activate when the user mistakenly “fat fingers” Button#2, and presses Button#1 along with it. Because of the close proximity of the buttons to one another, this activation is highly likely, leaving the user with a system which Denies Service.

Stage 3 activation of our Trojan puts it into key leak mode. Unlike Stage 2, this effect only lasts for the duration of the current iteration. On the next iteration, it will return to its previous iteration’s stage. For example, if the Trojan was in Stage 1 when Stage 3 was activated, it will return to Stage 1 on the next iteration. If the Trojan was in Stage 2 when Stage 3 was activated, it will return to Stage 2 on the next iteration. If the Trojan was originally in Stage 1, and Stages 2 and 3 are activated in the same situation, it would follow the behavior of Stage 3, since Stage 3 is a higher stage and thus has precedence over Stage 2. The next iteration would revert back to Stage 2, since Stage 3 only lasts for a single iteration, and the Trojan would output “DEADBEEF,” since it would then be the second iteration of Stage 2. This stage of the Trojan is activated by pressing Button#1 when the device is waiting for plaintext on an empty keyboard input buffer. That is, when the device is expecting plaintext, but before anything is input. The user would continue entering text as usual and activating encryption as usual by pressing Button#2. However, when the user presses Button#1 to transmit the ciphertext over the RS232 port, it will transmit the user selected key (DIP Switches) instead. To transmit the master key, one should activate Stage 3, enter any amount of text and press and release Button#1 again followed by more text input or pressing Button#2 to encrypt the plaintext. The output over the RS232 port will be the master key instead of the user selected key.

## **Difficulties in Implementation**

The official rules of the CSAW 2008 Hardware Challenge calls for use of the Xilinx Synthesis Tool 10.1. This version of the tool is known to be buggy, and has caused much headache for the Trojan Development Team. Over the course of 2 days, we have produced numerous different builds of the bit file, all with different FPGA utilization numbers. This makes matching

the original synthesis utilization consistently extremely difficult. As it turns out, the XST benchmarks the computer during synthesis, and adjusts the placement and routing effort accordingly. So to yield consistent results, our design would need to be synthesized in a completely controlled environment every time with the exact same system resources available, and needless to say, this was quite impossible, given today's OSes numerous background processes and physical/virtual memory configurations, amongst other things. To be conclusively consistent with what the judges would get when synthesizing, we would need to be able to replicate exactly the amount of CPU, HDD, RAM, etc. utilization during their time of testing. Specifically, our builds have jumped back and forth between 92% and 98% chip utilization at one computer, while sometimes dropping down to as low as 91% chip utilization when synthesized on a more powerful computer with the exact same software and source files.

With the same exact source files, bit files were produced so vastly differently that some builds would function properly, while others would not. One of our Trojans was put in by putting an if statement at the output of AES encryption, and during a build of this one day, the board would always output the same last 8 digits of ciphertext. When this same code was built another day on the same computer, the board would work properly. A later version of this file also exhibits the same problems, although we did not touch the if statement at all in this later version.

Another bug that we found during testing was the keyboard controller's inconsistencies. Sometimes, pressing a key would not display through the VGA port, and pressing the next key will have both of them show up. At other times, the keyboard would get "stuck" and repeat characters. These, although each possibly miniscule in certain respects, added up and caused quite the delay in our development of better malicious code. Thus, we have enclosed the working build of our bit file for reference.

## **Intended Trojans**

**We started to produce a handful of other Trojans, but were forced to stop development because of lack of time and increasing levels of frustration. Two of the Trojans that were nearing completion relied on inconsistency of the ending sequence. The ending sequence normally varies between 14 and 15  $F_{16}$ 's because the definition of AES does not allow more than 3 consecutive bits to be the same. One Trojan uses this inconsistency to slowly leak the key; it would output 14  $F_{16}$ 's for 1 and 15  $F_{16}$ 's for 0. An advantage to this is that it would be very hard to detect, as it would require 128 transactions and a close analysis of the end sequence that is normally not displayed or used.**

The second version of this would output an ending sequence of 8  $F_{16}$ 's and place the master key afterwards. The alpha decryption algorithm ignores any and all transmission after it reads 5  $F_{16}$ 's, so neither user would see the hidden code, while any Serial Port Terminal software would uncover the hidden end sequence. This method is advantageous as it only requires one transmission, can be activated all the time (just like the previous Trojan idea), and is completely transparent to most users.

Yet another idea was to transmit the key in the second half of every transmission packet, effectively doubling the frequency of the transmission. This would be a side channel that once again, regular users would not notice, while a program that looks at the shifted packets would be able to identify the hidden message. Upon further research, we found out that most UART devices look in the middle of every transmission bit and our original thought would be very hard to implement. The key would need to be in the edges of every bit of data, while the encrypted text in the center. Further more, this is implemented differently on various UART chipsets so it would not always work as expected in all situations.

## Utilization

The utilization of the device ended up being lower than the original version. As seen in the figure below.

Device Utilization Summary			
Logic Utilization	Used	Available	Utilization
Number of Slice Flip Flops	1,246	4,896	25%
Number of 4 input LUTs	4,052	4,896	82%
<b>Logic Distribution</b>			
Number of occupied Slices	2,300	2,448	93%
Number of Slices containing only related logic	2,300	2,300	100%
Number of Slices containing unrelated logic	0	2,300	0%
<b>Total Number of 4 input LUTs</b>	<b>4,205</b>	<b>4,896</b>	<b>85%</b>
Number used as logic	4,052		
Number used as a route-thru	153		
Number of bonded <a href="#">IOBs</a>			
Number of bonded	46	108	42%
IOB Flip Flops	16		
Number of RAMB16s	8	12	66%
Number of BUFGMUXs	4	24	16%
Number of DCMs	3	4	75%
Number of MULT18x18SIOs	2	12	16%

Figure 1. Device Utilization

We were unable to produce utilization numbers closer to the original version because as soon as we added any new code to our files, the output of the encryption would always end in DAC7DBB5<sub>16</sub> no matter what input was given. Interestingly enough, the Trojans would still function properly. We are still not sure if the problem is because of the inconsistencies with the Xilinx ISE software or the original code, which has also been reported to be buggy.

## **Conclusion**

In conclusion, we learned that to be secure, code must be clean and thoroughly bug-tested before release. Sloppy practices such as improper use of case structures, elementary key “randomization”, buggy keyboard drivers, etc. allow for easy security breaches to be put in. Our compromised version of Alpha V1.1 is a good demonstration of this, and can be seen as detailed in the cleanly commented source code.