Project for ECE 525

Assigned: Wed., Mar. 23, 2016
Due: Progress Reports Every Wednesday, Presentation/demonstration last week of class

Description: Implement the HELP Authentication Protocol

You will implement an 'in-field' mutual, privacy preserving authentication protocol using the Zedboard as the token. Your laptop will serve the role of 'secure server' initially. Later, I will provide the secure server function on my laptop and everyone will connect their boards to an ethernet switch so we can model a network of tokens communicating to a secure server.

The bitstream implements a VHDL version of SHA-3, Keccak-F200 version with a rate of 72 bits and a capacity of 128 bits. The implementation uses WDDL-style logic to ensure that no circuit hazards exist. The SHA-3 algorithm can be operated in two different modes: Functional and PUF modes.

In Functional mode, a 70-bit bitstring to be hashed is first transferred from the C program to VHDL registers using the two GPIO ports and then a start bit is asserted. After 18 clock cycles (MMCM should be set to 50 MHz), a 64-bit hash value can be retrieved into the C program from a set of VHDL registers, again using the GPIO ports. An example demonstrating this mode of operation is given toward the bottom of the supplied *token_enrollment.c* program.

In PUF mode, two 416-bit vectors are transferred to VHDL registers using the GPIO ports and a start bit is asserted to carry out the timing operation. 10-bit timing values are then retrieved from VHDL registers, one at a time, into the C program. This functionality is implemented as the main component of the supplied *token_enrollment.c* program.

The vectors used during PUF mode are selected from a set of 5,000 rising and 5,000 falling random vectors that have been characterized in hardware experiments as 'low-noise' and 'minimally-overlapping', i.e., the variations in the timing values are small across TV corners and the more than 4,000 paths tested a nearly distinct. The set of 20 vectors provided will provide more than 4,000 timing values (each vector sensitizes a minimum of 200 paths simultaneously), which should be plenty to carry out the 'in-field' authentication operation that you will write C code for later.

Details of the protocol will be provided below and in the course slides discussed in class.

Part I:
1) Download the bitstream, *design_1_wrapper.bit*, along with the C code for *token_enrollment.c* and *verifier_enrollment.c* and the test vector file *enrollment_20Vs.txt* from the course website under *project*.

The SHA-3 algorithm accepts the following as inputs to the hash:
"11xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"
and produces a 64-bit hash. Note that you MUST include the leftmost two '11' as hash input. The 'x' are to be filled in with a 'strong bit string' of length 70 (discussed below). An example

of how you invoke the SHA-3 hardware engine is located at the bottom of the *token_enrollment.c* file using the following all-zero bitstring:
"1100000000000000000000000000000000000000000000000000000000000000000000000000000000"

The hash mode of operation of the SHA-3 will be required at several points in the execution of our authentication protocol, as described below and discussed in class.

2) The first part of the authentication operation is referred to as enrollment, and is carried out immediately after manufacturing the tokens in a secure environment. Its purpose is to collect timing values from the tokens and store them in a secure database (possibly encrypted). These timing values will later be used during 'in-field' authentication to construct response bitstrings. The complete enrollment operation is provided to you in the above files. You should

1) Program the board with the bitstream provided.
2) Compile *token_enrollment.c* using SDK and transfer the *token_enrollment.elf* file to the Zedboard.
3) Configure the network using the *ifconfig eth0 192.168.1.10 netmask 255.255.255.0* (or whatever you have been using for the lab assignments)
4) Run the command *token_enrollment.elf Cx 0 0 0 192.168.1.10* (in the background if you prefer). The token will run as a 'server' during the enrollment process, accepting a connection from the server (laptop), and then obeying the commands transmitted to it over the network.
5) Compile *verifier_enrollment.c* on your laptop.
6) Run the command *verifier_enrollment 192.168.1.10 enrollment_20Vs.txt* on your laptop. Your laptop acts as a client in the client-server model during enrollment, first sending the 20 vector pairs to the token and then receiving the timing values. The timing values are written to an output file in the same format used in the labs. We will not concern ourselves with storing them in a 'secure database', at least in the initial part of the project.

3) Prepare to demonstrate this operation during class on Monday.

PART II: (due: April 4th) and PART III: (due April 11th)
The authentication protocol is a sequence of steps that occur, in parallel, on the token and verifier. We will discuss the entire protocol in the coming weeks. These 'parts' of the project involve programming the components of the protocol. The execution of your protocol must printf the messages at the beginning of each step as shown below.

The roles of the client and server are reversed during authentication, with the Zedboard (token) representing the client and your laptop (verifier) representing the server. The token code should be put into a file called *token_authenication.c* and the server code in a file called *verifier_authentication.c*. You can copy the *token_enrollment.c* and *verifier_enrollment.c* files to the *_authentication* versions, but need to interchange the *OpenSocketServer()* and *OpenSocketToken()* code and calls.

The server will run continuously during authentication (unless you kill it with <ctrl>C). To achieve this, you need to place the steps given below in an infinite 'while(1)' loop. NOTE: the first call to *OpenSocketServer()* should set the parameter 'accept_only' to false and then subsequent calls should set it to 1 (you only need to open the socket once and subsequently accept connection requests from tokens withOUT re-opening the socket).

Token steps:

"1) Send 'GO' request to the verifier"

Send the string "GO\n" to the verifier

"2) Receiving ID vectors from verifier"

Receive 20 challenges (10 rising and 10 falling) from the vector file supplied. Do this inside of a subroutine called *ReceiveIDPhaseVectors()*

"3) Generating nonce 'n1'"

a) Do this inside of a subroutine called *GenerateNonce()*

b) Generate nonce 'n1' by applying only the first vector to keccak in PUF mode and retrieving the timing values (more than 200 of them). Use ONLY the first sample, i.e., you do not need to produce all 16 samples and compute an average as you've done in the labs. YOU ONLY NEED TO PRODUCE 10 bytes of nonce for now.

b) Compute the nonce (random bitstring) by taking each timing value 'mod 2'

c) Pack the bitstring into an array of 'unsigned char' with the low order bit of the first byte storing the result from the first timing value, the second bit storing the result of the second timing value, etc. When you have stored 8 values in the first 'unsigned char' byte, increment to the second byte in the array and repeat. NOTE: THESE ARE NOT ASCII characters, they are binary values stored 8-bits at a time in a unsigned character array. You'll need to use printf("%d", xxx) to print them and not "%c", and will make use of the '<<', '>>', '&' and 'I' C operators to construct the binary unsigned character array.

NOTE: As discussed in class, in order to ensure randomness in your nonce, you should consider collecting 16 samples per timing value (as the enrollment code does) and then screen timing values that do not exhibit meta-stability. This can be accomplished in the simplest way by simply counting how many times the first sample is repeated in the 16 samples. If the first sample is not repeated exactly 8 times, skip it (do not allow it to be used as a bit in the nonce). You will very likely need to process more than 1 vector to achieve the 10 byte goal for the nonce.

Print the generated nonce using:

```
printf(\t"Generate nonces n1 and n3:\t");
for ( i = 0; i < num_token_nonce_bytes; i++ )
  printf("%02X\t", token_nonces[i]);
printf("\n");
```

Parameter 'num_token_nonce_bytes' is set to 10.

"4) Sending nonce 'n1' and 'n3'"

Nonces are defined as 35 bits each (although we may increase the size to 70 bits each in the future). This is 4-bytes + 3 additional bits of the 5th byte. For this 'ID' phase of the authentication protocol, you need nonces 'token_n1' and 'verifier_n2', each 35 bits, to be taken from the first 5 bytes of the generated nonces. The second 5 bytes will be for the 'Authen' phase that you will develop later and labeled 'token_n3' and 'verifier_n4'.

Just do a 'SockSendB' to transmit the 'token_n1' nonce to the verifier.

"5) Receiving nonce 'n2' and 'n4'"

Just do a ‘SockGetB’ to retrieve the ‘verifier_n2’ nonce.

At this point, both the token and verifier have both nonces ‘token_n1’ and ‘verifier_n2’, each 5 bytes in length.

Print the verifier nonce using the following:
printf(\t"Received nonces n2 and n4:\t");
for (i = 0; i < HASH_IN_LEN_BYTES + 1; i++)
printf("%02X\t", verifier_n2[i]);
printf("\n");

“6) Compute hash of nonces ’n1’ and ’n2’”
Create a nine byte combined nonce as follows. NOTE: The high order 2 bits MUST be “11” for the token and MUST be “00’ on the server (as described below).
“11” & 35-bits of ‘token_n1’ & 35-bits of ‘verifier_n2’
The ‘&’ operator indicates concatenation. You may want to write a routine that ‘joins’ binary unsigned char arrays, e.g.:
JoinPairBinStr(MAX_STRING_LEN, MAX_BINARY_BYTES, (HASH_IN_LEN_BITS-2)/2, token_n1, (HASH_IN_LEN_BITS-2)/2, verifier_n2, hash_in_bin_str, 0)
The last parameter in my routine indicates whether to add “00” or “11” as the high order two bits.

Compute the hash using the hardware version of keccak using calls similar to:
LoadHashInputBinStr(MAX_STRING_LEN, CtrlRegA, DataRegA, hash_in_bin_str, ctrl_mask, HASH_IN_LEN_BYTES);
ComputeHashBinStr(MAX_STRING_LEN, CtrlRegA, DataRegA, ctrl_mask, HASH_OUT_LEN_BYTES, hash_out_bin_str);

NOTE: The example code that I provided assumes the data to be hashed is ASCII. ‘LoadHash-InputBinStr’ is a version of the ‘load’ code that treats the input as a binary unsigned char array. You will need to modify the routine I’ve given you in the enrollment code since your input is binary now. The binary version is simpler than the ASCII version since you do not need to convert the bytes to binary any longer.

Print out the hash using the following:
printf(\t”Hash of nonces:\t");
for (i = 0; i < HASH_OUT_LEN_BYTES; i++)
printf("%02X\t", hash_out_bin_str[i]);
printf("\n\n");

“7) Selecting parameters using hash output’”
Call *SelectParams* as follows. *SelectParams* is given the *utilities.c* and posted on-line:
SelectParams(HASH_OUT_LEN_BYTES, hash_out_bin_str, &LFSR_seed, &mean_scaler, &range_scaler, &modulus, &margin);

“8) Generating PNs”
Switch to PUF mode:
ctrl_mask = (1 << OUT_CP_MODE_FUNC_PUF);

And call a routine called *GenTimingVals*, which applies the 20 vectors to the PUF and generates 4730 timing values. As indicated in class, you want want to limit the number of PNs (timing values) that you store in your array to 2048 for the 10 rising vectors (first 10) and the 10 falling vectors (second 10). You actually reach 2048 rising PNs somewhere in the middle of the 9th vector so technically, you don't need to even apply the 10th rising vector but up-to-you how you want to handle it. Compute an average PN after collecting 16 samples and store the average PNs as an array of floating point numbers.

"\t8.1) Computing PNDiffs\n"

Write a routine called *ComputePNDiffs* that computes differences between the rising and falling edge PNs. Sequentially step through the rising edge PNs from 0 to 2047. For each rising edge PN, call the 11-bit LFSR to determine the falling edge PN to pair with this rising edge PN and create the difference as 'rising edge PN - falling edge PN'. Use the 'LFSR_seed' returned from *SelectParams* in the first call to the LFSR. I've provided the LFSR as *LFSR_11_A_bits* in the *utilities.c* file. The floating point *PNDiffs* array that stores the differences should have 2048 values.

"\t8.2) TVCOMP\n"

Write a routine called *TVCOMP*. The routine first needs to compute the 'cur_mean' and 'cur_range' of the 2048 PNDiffs. The 'cur_range' should be computed by multiplying the standard deviation computed from the distribution by 3. Compute a floating point constant called 'range_conversion' using the 'cur_range' and the 'range_scaler' returned from *SelectParams* as follows:

range_conversion = range_scaler/cur_range;

Apply the following transformation to the PNDiffs:

*PNDiffs[PND_num] = (PNDiffs[PND_num] - cur_mean)*range_conversion + mean_scaler;*

Note that 'mean_scaler' is produced by *SelectParams*

"\t8.3) Computing modPNDiffs\n

Write a routine called *ComputeModulus*. Round the PNDiffs using the following:

rounded_PNDiff = PNDiffs[PND_num] < 0 ? (int)(PNDiffs[PND_num] - 0.5) : (int)(PNDiffs[PND_num] + 0.5);

Apply the *SelectParams* 'modulus' to the 'rounded_PNDiff'. Be sure to add the modulus if the result is negative, i.e., all values MUST be positive.

"\t8.4) Computing strong bitstring\n"

Write a routine call *BitGenSingleM* that constructs a strong bitstring 'token_ID_SBS' and helper data 'token_ID_SHD' using the algorithm discussed in class. Use the 'margin' and 'modulus' returned by *SelectParams* to define the strong bit regions. You should pack the bits into unsigned char bytes as you did for the nonces since we are going to hash the first 62 bits of the strong bitstring (see below). Same is true for the helper data string, which will be transmitted to the verifier. Note that this routine *BitGenSingleM* is different than the verifier's *BitGenSingleH* described below.

Print out the strong bitstring:

```
printf("\t\tID strong bitstring\n\t\t\t");
for ( i = 0; i < token_ID_SBS_size_bits/8; i++ )
  {
  printf("%02X\t", token_ID_SBS[i]);
```

```
  if ( (i + 1) % 10 == 0 )
    printf("\n\t\t\t");
  }
printf("\n\n");
```

“\t8.5) Hashing strong bitstring\n"

Switch to hash mode:

```
ctrl_mask = (0 << OUT_CP_MODE_FUNC_PUF);
```

Join the first 8 bits of the nonce ‘token_n1’ with 62 bits of the strong bitstring ‘token_ID_SBS’, being sure to add “11” to the upper to bits of the 9 bytes hash input. Load and compute the hash.

Print it out:

```
printf(“\tFinal hash of Strong bitstring ID Phase + token n1\n\t\t");
for ( i = 0; i < HASH_OUT_LEN_BYTES; i++ )
  printf("%02X\t", hash_out_bin_str[i]);
printf("\n\n");
```

"9) Send hash of strong bitstring and helper data\n"

Transmit the 8 bytes of the hash and helper data bitstring to the verifier.

"10) MUTUAL: Compute hash of ID strong bitstring + verifier n2\n"

Join the bitstring “11” with 8 bits of the ‘verifier_n2’ nonce and 62 bits of the ‘token_ID_SBS’ (token strong bitstring) -- in this order from left to right and then hash it.

"11) MUTUAL: Receive hash of ID strong bitstring + verifier n2\n"

Get the equivalent from the verifier from the socket.

"12) Comparing computed hash with verifier’s hash\n"

Compare byte-for-byte the two hashes.

Print if they don’t match:

"********** FAIL -- MUTUAL AUTHENTICATION **********\n"

And, if they do:

"********** SUCCEED -- MUTUAL AUTHENTICATION ********** \n"

"13) Sending ’DONE’\n"

Send the string “DONE” to the verifier and exit.

VERIFIER:

Before entering the infinite loop below, carry out the following operations:

1) Print the line:

printf("Initialization Operations:\n"); fflush(stdout);

2) Read vector pairs and store them in the ‘first_vecs’ and ‘second_vecs’

3) Read timing data from the file that was generated during Enrollment, e.g., *C1_25C_1.00V_E_PUFNums.txt*

Note that your data file will contain data from only 1 chip. In the actual application, this file would be a database and would contain data from multiple tokens, all collected during enrollment.

Verifier steps: NOTE: This code is in an infinite loop

"1) Waiting 'GO' request from token"

Call *OpenSocketServer()* as described above. This routine will block until a token requests a connection.

Call *SockGetS()* to get the 'GO' string

"2) Sending ID vectors to token"

Read the vector file and send the vectors to the token inside of a subroutine that you call *Send-Vectors()*

"3) Generating nonce 'n2'"

ABOVE this infinite while loop, open up the /dev/urandom device using the following call:
if ((RANDOM = open("/dev/urandom", O_RDONLY)) == -1)
You can then do a 'read' on this device, passing as arguments a 'verifier_nonce' 10 byte buffer and a size 'num_verifier_nonce_bytes' set to 10.

Print the generated nonces on one line using:

```
printf(\t"Generate nonces n2 and n4:\t");
for ( i = 0; i < num_verifier_nonce_bytes; i++ )
  printf("%02X\t", verifier_nonces[i]);
printf("\n");
```

"4) Receiving nonce 'n1' and 'n3'"

Nonces are defined as 35 bits each (although we may increase the size to 70 bits each in the future). This is 4-bytes + 3 additional bits of the 5th byte. For this 'ID' phase of the authentication protocol, you need nonces 'token_n1' and 'verifier_n2', each 35 bits, to be taken from the first 5 bytes of the generated nonces. The second 5 bytes will be for the 'Authen' phase that you will develop later and labeled 'token_n3' and 'verifier_n4'.

Just do a 'SockGetB' to fetch the 'token_n1' nonces sent by the token. Print them out on one line using:

```
printf(\t"Received nonces n1 and n3:\t");
for ( i = 0; i < HASH_IN_LEN_BYTES + 1; i++ )
  printf("%02X\t", token_n1[i]);
printf("\n");
```

"5) Sending nonce 'n2' and 'n4'"

Just do a 'SockSendB' with the 'verifier_n2' bytes retrieved from 'urandom' above.

At this point, both the token and verifier have both nonces 'token_n1' and 'verifier_n2', each 5 bytes in length.

"6) Compute hash of nonces 'n1' and 'n2'"

Create a nine byte combined nonce as follows. NOTE: The high order 2 bits MUST be "00" for the server and MUST be "11' on the token (as described above).

“00” & 35-bits of ‘token_n1’ & 35-bits of ‘verifier_n2’
The ‘&’ operator indicates concatenation. You may want to write a routine that ‘joins’ binary unsigned char arrays, e.g.:
JoinPairBinStr(MAX_STRING_LEN, MAX_BINARY_BYTES, (HASH_IN_LEN_BITS-2)/2, token_n1, (HASH_IN_LEN_BITS-2)/2, verifier_n2, hash_in_bin_str, 0)
The last parameter in my routine indicates whether to add “00” or “11” as the high order two bits.

Compute the hash of the 72 bit nonces (with high-order 2-bits set to “00”) using the supplied ‘keccak.c’ code:

```
if ( ComputeHashKeccak(HASH_IN_LEN_BYTES, hash_in_bin_str, HASH_OUT_LEN_BYTES, hash_out_bin_str) < 0 )
    { printf("ERROR: Keccak failed!\n"); fflush(stdout); exit(EXIT_FAILURE); }
```

Print the hash using:

```
printf(\t"Hash of nonces:");
for ( i = 0; i < HASH_OUT_LEN_BYTES; i++ )
  printf("%02X\t", hash_out_bin_str[i]);
printf("\n\n");
```

NOTE: THIS HASH STRING MUST MATCH the hash string printed by the token above.

"7) Selecting parameters using hash output\n"
Same as token -- see above

“8) Getting token’s hash of strong bitstring and helper data\n"
Wait for token to transmit the strong bitstring and helper data.

Print it out:

```
printf(“\tToken strong bitstring hash size bytes %d\n\t", token_ID_hash_size_bytes); fflush(stdout);
for ( i = 0; i < token_ID_hash_size_bytes; i++ )
  printf("%02X\t”, token_ID_hash[i]);
printf("\n");
printf(“\tToken helper data size bytes %d\n\n", token_ID_SHD_size_bytes); fflush(stdout);
```

"9) Searching database\tChip %d\n", chip_num); fflush(stdout);
Note that I’ve added a section above which requires that you read the file *Cx_25C_1.00V_E_PUFNums.txt* before entering the infinite loop. As noted, you file will contain data for only 1 chip so the following *while(1)* loop will only execute once during the search process for a match. Authentication should ALWAYS succeed in this search since the authentication hash received from the token will match the hash you compute using the enrollment data. ‘chip_num’ referenced above should be 0 for you since you have data from only 1 chip in your data file.

Add the following code to a second infinite while loop, *while(1)*

“\t9.1) Computing PNDiffs\n"

Same as token -- see above.

“\t9.2) TVCOMP\n"
Same as token -- see above.

"\t9.3) Computing modPNDiffs\n"
Same as token -- see above.

“\t9.4) Computing strong bitstring\n"
Write a routine called *BitGenSingleH*. This version of the strong bitstring generator takes the ‘modPNDiffs’ just computed and the helper data received above this loop from the token and uses the helper data to select the bits to be used in the construction of the strong bitstring ‘verifier_ID_SBS’. You need only the ‘modulus’ returned from *SelectParams* to decide the value of the bit.

As an aside: This version does NOT produce helper data since it uses the helper data from the token. However, in another version of this algorithm (not required in this project), both the token and verifier run the a routine called *GenHelperData*, which computes only the helper data. The helper data bitstrings are exchanged and AND’ed by each the token and verifier to produce a new helper data bitstring which is used in *BitGenDualH* to produce the strong bitstrings. A careful analysis shows that doing this reduces the possibility of a bit flip error by approx. 1/2 over the *SingleHD* scheme and allows the ‘margin’ parameter to be set to a value of 5 (instead of 10 as *SelectParams* returns). This in turn, allows smaller moduli and better access to within die variations (the best source of entropy).

Print it out:

```
printf(“\t\tID strong bitstring\n\t\t\t”);
for ( i = 0; i < verifier_ID_SBS_size_bits/8; i++ )
  {
  printf("%02X\t", verifier_ID_SBS[i]);
  if ( (i + 1) % 10 == 0 )
    printf("\n\t\t\t");
  }
printf("\n\n");
```

“\t9.5) Hashing strong bitstring\n"
Same as token -- see above. NOTE: be sure the two high order bits of the hash input string are “00” for keccak.c

“9.6) Comparing computed hash with token’s hash\n"
Compare the token and verifier hashes, byte-by-byte. If they match, print:

```
printf("********** SUCCEED -- AUTHENTICATION **********\n");
```

If they do NOT match, then in an actual application, you would start again with the data from the next chip.

In your case, if they don’t match, print:

*printf("********** FAIL -- MUTUAL AUTHENTICATION **********\n");*

'break' from the inner while loop either way.

"10) MUTUAL: Compute hash of ID strong bitstring + verifier n2\n"
Do this ONLY if you succeeded above.

Joing the bitstring "00" with 8 bits of the 'verifier_n2' nonce and 62 bits of the 'verifier_ID_SBS' (verifier strong bitstring) -- in this order from left to right and then hash it.

"11) MUTUAL: Send hash of ID strong bitstring + verifier n2\n"
Send the hash to the token.

"12) Waiting token's 'DONE' signal\n"
Wait for the token to send the 'DONE' string. When/if received, close token socket connection and go back to top of outter loop. Wait for next connection.

You will demonstrate your protocol at the project due date. Details to be discussed in class.