Final Report: Analysis of Security In a Modern Processor

Justin Cox and Tyler Travis
Department of Electrical and Computer Engineering
Utah State University
Logan, Utah 84322
email: justin.n.cox@gmail.com, tyler.travis@aggiemail.usu.edu

Abstract— **Hardware security is an ever increasing area of study since exploits have been found on computer systems. It has been found that attackers are able to look at the memory of a system to see what is being processed. This paper proposes adding an extra level of security between the memory and the pipeline. This layer of security would encrypt data headed to memory and decrypt the data headed to the pipeline making the data held in memory incomprehensible to an attacker. To make the added cryptographic system more secure, research will be done on the benefits of utilizing a Physically Unclonable Function (PUF). Further analysis will be done on how this will affect the processor's performance.**

Index Terms—**encryption, decryption, security, pipeline, PUF.**

I. INTRODUCTION

Over the years, there have been many advancements made to computer systems and architecture. These advancements have been primarily focused on improving performance and power consumption. As a result, the security of these architectures has been neglected. Malicious attacks on computer systems are becoming more common and as a result, there is greater need for more secure systems.

The optimal solution to this problem is to completely redevelop the architectures with security as a top priority. However this option is not very practical as it would be far too expensive to replace the current processors and there is a need to support legacy machines. Therefore, we propose to create security modules that can be added to current processor architecture.

This paper shows the authors' process of developing a DES cryptography module designed to be inserted into the gem5 simulation software. The paper also shows how a PUF can be used to increase the security of the DES module. The results of the architecture's performance with the added security module will be simulated and the results will be given.

II. BACKGROUND

Computer security is becoming more important as more systems and devices are being compromised. In particular, there has been an increased number of attacks focused on stealing information stored in memory on computers and servers. This information can be encrypted when not in use, but currently there are very few solutions that allow the computer architecture to work with the encrypted data. As a result, when the architecture is running, the data stored in memory is vulnerable to attackers.

Previous research has been done on the development and implementation of secure processors [1]. Similar to the previous research done, this paper will focus on data encryption and decryption and the toll it takes on processor performance. The objective is to provide as much security as possible without reducing the performance heavily.

III. TECHNIQUE

The following subsections will describe the techniques used to supply current computer architecture with more data security. The first subsection will describe how the data will be protected by cryptography. The last section will describe how the cyptograhic modules will be improved using a PUF.

A. Encryption and Decryption of Data

The memory can be modified by an attacker and as a result the attacker is able to change the execution flow of a program. Since the data is not encrypted, the attacker is also able to steal important information that may be contained in memory. Therefore to prevent the attacker from knowing where to modify or read the data, it is important for the CPU to encrpyt data being stored into memory. Initially, we will encrypt memory that is stored in registers, cache, RAM, and virtual memory [1]. If the performance decreases significantly, encryption will be excluded from the registers and/or cache. Assuming the pipeline is trusted, data being pushed into the pipeline will be decrypted.

These encryption and decryption modules will be built and inserted into the gem5 source code.

B. Data Encryption Standard (DES)

The type of encryption used for the module will be DES. Although DES is not the current encryption standard and is not as secure as the Advanced Encryption Standard AES, DES will allow better processor performance while still making the data more secure.

A brief overview of DES will be given so that the reader has a better understanding of how encryption could take a toll on processor performance. If the reader would like an in-depth understanding of DES, it is recommended that the reader look to other sources [2].

The DES algorithm takes a 64-bit plain text input. It is then run through an initial permutation that outputs 56-bits when are then split into two halves. The data goes through sixteen rounds that each have a sub-key that is generated for each round based on the original 64-bit DES key. After the sixteenth round, the output is run through a finial permutation and the algorithm outputs a 64-bit encrypted cipher text. The rounds are illustrated in Figure 1.

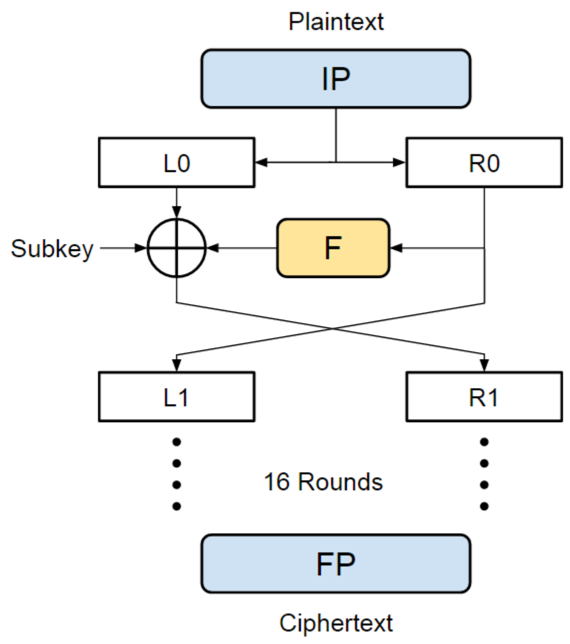

Fig. 1. An illustration of the sixteen round DES algorithm.

During each round, the left 32-bit halve is XORed with the sub key corresponding to the current round as well as the output of the F-function. The inside functionality of the F-function is illustrated in Figure 2. The output of the XOR is used as the next round's right halve and the next round's left halve is the previous round's right halve.

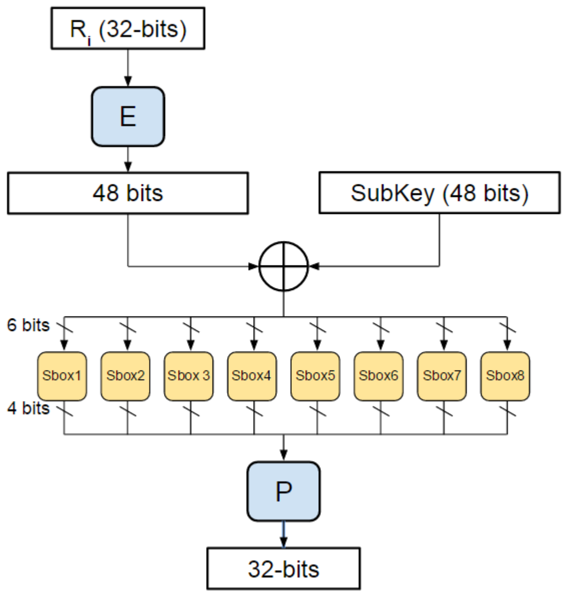

Fig. 2. An illustration of the F-function

C. Physically Unclonable Function (PUF)

A Physical Unclonable Function (PUF) is a physical entity used to produce information that can only be reproduced using the same physical device. Since physical imperfections and deviations are normal in the manufacturing and design process of circuits and devices, each different device of the same family should have different physical characteristics that only pertain to said device. These imperfections, whether they be voltage levels, temperature, or delay times, can be used to produce information.

PUFs are especially effective when used in security. Since PUFs are generally difficult to model, the only way an attacker could recover the information generated by the PUF is to steal the actual physical device. Even then, if the attacker where to inspect the PUF they would risk changing the physical characteristics of the device and the PUF's output may change.

There are many different types of PUFs such as Optical PUFs, Delay PUFs, and SRAM PUFs [3].

D. Secret Key Generation Using a PUF

A PUF is used to generate secrets extracted from a physical device due to manufacturing differences. This uniqueness can be used for secure key generation for encryption/decryption [4].

One of the difficulties of using a PUF for secret key generation is the value of $\mu intra$. The measurement $\mu intra$ is used to determine how much the output response changes of a certain PUF when given the same input challenge. Ideally, a PUF should generate the same response when given the same challenge. Two different PUFs should also generate a different response when given the same challenge. This measurement is referred to as $\mu inter$. A good PUF should have a $\mu inter$ close to 50% and a $\mu intra$ close to 0%.

The reason $\mu intra$ is important is because in a encryption/decryption algorithm, even the slightest change to the secret key will change the cypher-text dramatically. This means that in order to use a PUF for secret key generation, the PUF's challenge-response pairs need to be consistent. Since this is almost impossible to do using a PUF, Error Correcting Code (ECC) may be used to fix the response output. A flow chart of this design is shown in Figure 3.

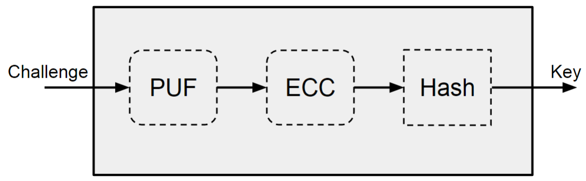

Fig. 3. Flow chart of the secret key generation process.

To fix this, majority voting may be used to determine what bits of the response output typically change given the same challenge input. This process is explained in detail in [4], but a brief overview of the process will be given. A chosen challenge message is sent through the PUF multiple times.

It is then determined which bits have a tendency to change. These bits are thrown out and the new challenge message is sent in the ECC. The ECC encoding process will return parity bits associated with the certain challenge message. During normal operation, these parity bits can be used to fix the response output for a given challenge. This process makes sure that the secret key remains the same for a given challenge.

As shown in Figure 3, it is also a good idea to hash the output from the ECC to inhibit the attacker from knowing the response message. The design can be made even more secure by adding a hash function to the challenge input as well.

E. Problems to Consider with PUF Implementation

Although secret key generation using a PUF is a good way to increase the security of the DES algorithm, there are many challenges that are introduced to the proposed security module. One of the problems to consider is how much the PUF changes after a reboot of the system. Since the data on the system has been encrypted using the PUF on a previous reboot, the same challenge/response pair needs to be used to decrypt the information. If the PUF changes due to reboot variations, the CPU will no longer be able to use the stored data.

Another problem that may be introduced is the ability to keep track of the different challenge/response pairs used in the security module. There are two options available. One option is to only use one challenge/response pair. Although this facilitates keeping track of the challenge/response pair, it makes the security module less secure. If an attacker was able to guess or intercept the response, the system would be compromised. The other option is to change the challenge/response pair every now and then. While increasing security, it creates the necessity to keep track of which data corresponds to which challenge/response pair. This record keeping could become very complex as some of the encrypted data on the system could not be used for a very long time.

This paper will not go over the solutions to these problems, but research has been done to overcome these problems [5].

IV. METHODOLOGY

A. Benchmarking and Performance

The simulator that this project is using is gem5. The security module will be added into the source code for the o3 CPU. Extensive research had to be done in order to determine the right sections of code where the encrypt and decrypt modules need to be added. The o3 CPU in gem5 can use the SPEC2k6 benchmarks. A group of these benchmarks will be used to measure the performance of the o3 CPU. The benchmarks will first be ran without any modifications to the gem5 source code. In order to have more robust results, the benchmarks will be ran using two different pipeline widths of two and eight. The same procedure will be repeated but this time the benchmarks will be ran with the security modules inserted into the gem5 source code. The performance will be compared between the two groups of benchmarks. The benchmarks chosen to be evaluated and compared on performance are:

- bzip2
- gcc
- bwaves
- mcf
- gobmk
- hmmer
- GemsFDTD
- libquantum
- tonto
- omnetpp

Since the security module only affects the memory operations in the CPU, the benchmarks that have many store and load instructions will see the greatest performance hit.

B. DES Security Module Code

The DES encryption and decryption code was written in C++. In order to work with the gem5 source code, the syntax rules and file naming nomenclature used in gem5 were used. The code is organized in a header file named *des.hh* and a source code file named *des.cc*. The code is comprised of many sub-functions, but the two functions that will be called in the gem5 source code are the *encrypt()* function and the *decrypt()* function. The function prototypes look like the following:

```
void encrypt(uint8_t *plain_text, uint16_t
    plain_text_size, uint8_t *cipher_text,
    uint8_t key[8]);

void decrypt(uint8_t *plain_text, uint8_t
    *cipher_text, uint8_t key[8]);
```

The functions take a 64-bit plaintext array and a 64-bit key array. Since DES works with plaintext of size 64-bits, it needs to be ensured that the data that is encrypted and decrypted in gem5 is a multiple of 64-bits.

C. Modifications Made to gem5 Source Code

The authors began looking into the source code that handles load and store instructions. The encryption function needs to be called every time a store instruction is executed. The decryption function needs to be called every time a load instruction is executed.

After going up the hierarchy in the code, the authors were able to determine that gem5 uses a packet class to allow the CPU to communicate with the memory. These packets will be used to encrypt and decrypt the data. Figure 4 illustrates how the packets are used to encrypt the memory system. It should be noted that the registers are not encrypted. This presents a potential security flaw in the system because if an attacker gains control of program flow, he may be able to obtain important information stored in registers. In a future revision of this project, the authors would like to encrypt the registers as well.

The gem5 source files that received modifications are *lsq_unit.hh* and *lsq_unit_impl.hh*.

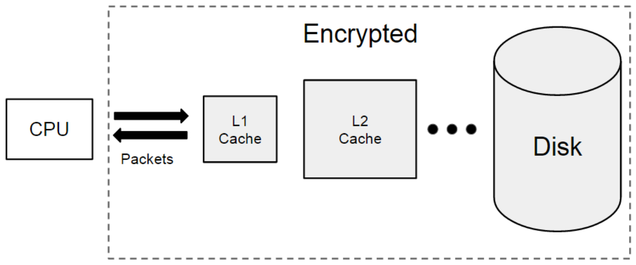

Fig. 4. Illustration to show what is being encrypted with the added security module.

D. Environment Used to Run Simulation

The simulations will be carried out using gem5 on a Linux machine. The benchmarks will be ran starting at a certain checkpoint to ensure that the performance measurements only include the benchmark and not additional overhead from CPU startup. The benchmarks will each be ran until the 100 millionth instruction is reached.

Since each benchmark simulation has the potential to require more than an hour to complete, the simulation process is very time consuming. To overcome this, the benchmarks will be ran on a cluster of Linux machines using condor. The results of each benchmark simulation are stored in a text file and the text files will be compared to generate the results.

V. Results

After the benchmark simulation result files were generated, they were processed and compared. Figures 5,6,7,8,9, and 10 show the simulation parameters that changed based on pipeline width and non-secured vs secured versions of the gem5 o3 CPU.

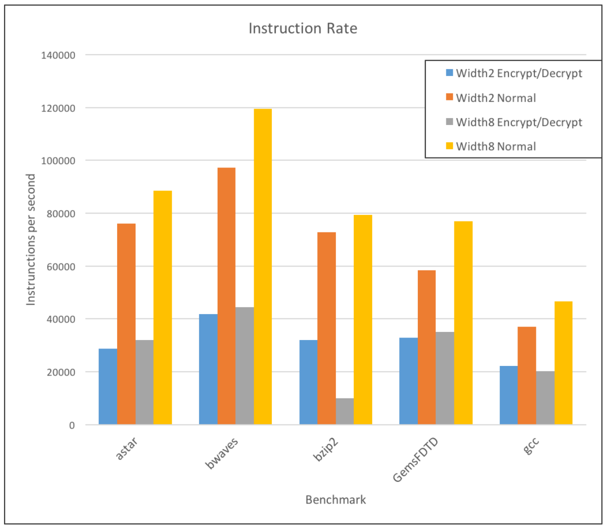

Fig. 5. Graph that compares simulator instruction rate for the first set of benchmarks.

The simulator instruction rate in seconds is shown in Figures 5 and 6. You can see that we observe the expected result that the pipeline with width 8 performs better than a pipeline with width 2. However, it should be observed

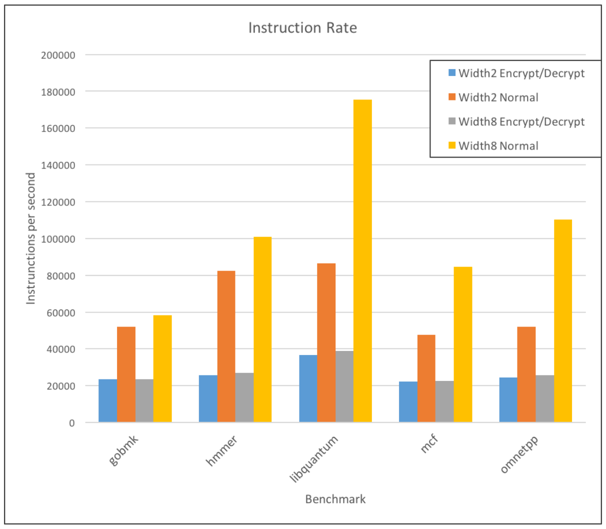

Fig. 6. Graph that compares simulator instruction rate for the second set of benchmarks.

that the benefits of the pipeline width are less prevalent in the encrypted/decrypted CPU. The main thing to observe from the graphs is that the secured CPU benchmarks have a performance decrease of more than half.

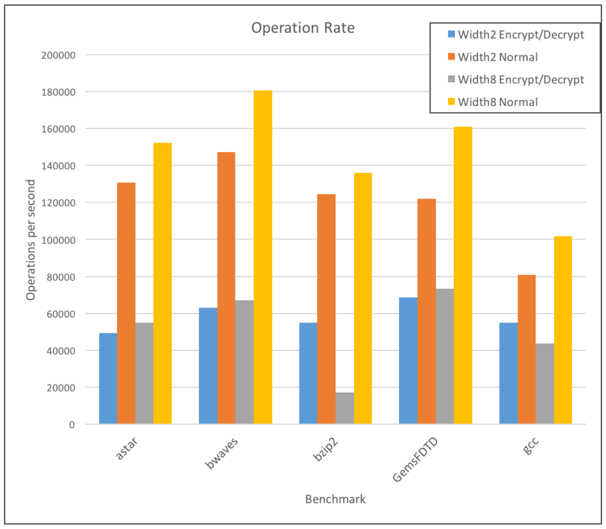

Fig. 7. Graph that compares simulator operation rate for the first set of benchmarks.

The simulator operation rate in seconds is shown in Figures 7 and 8. The operation rate follows the same patterns as Figures 5 and 6 and there is no need to explain them further.

The simulator run time in seconds is shown in Figures 9 and 10. It can be seen that the secured CPU benchmarks take longer to run the simulation. The simulation time is important because it allows the latency of the encryption/decryption functions to be calculated. The benchmark simulation results include the number of memory operations executed. The gem5 o3 CPU is running at 1 GHz so one clock cycle is 1ns. The equation to solve for the encryption/decryption latency

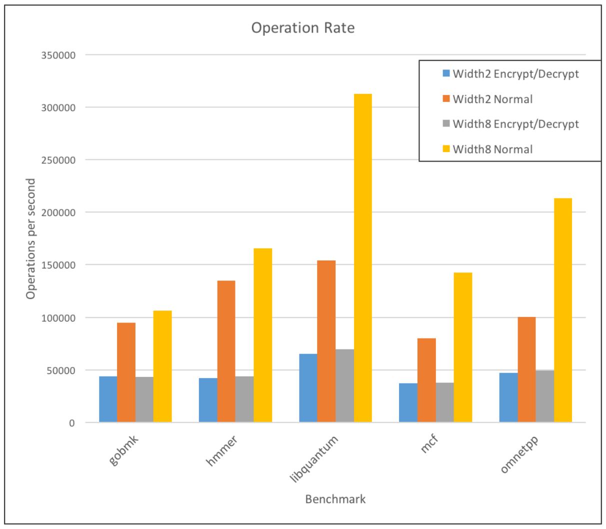

Fig. 8. Graph that compares simulator operation rate for the second set of benchmarks.

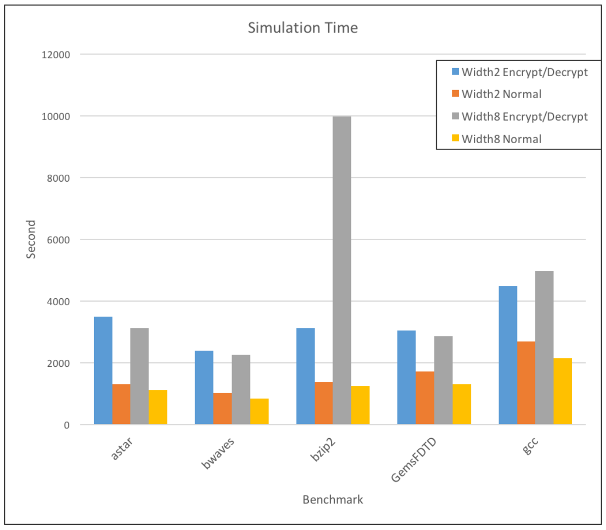

Fig. 9. Graph that compares simulator run time for the first set of benchmarks.

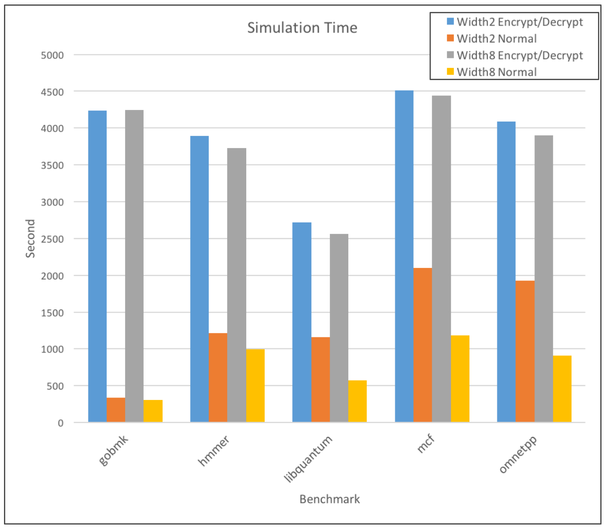

Fig. 10. Graph that compares simulator run time for the second set of benchmarks.

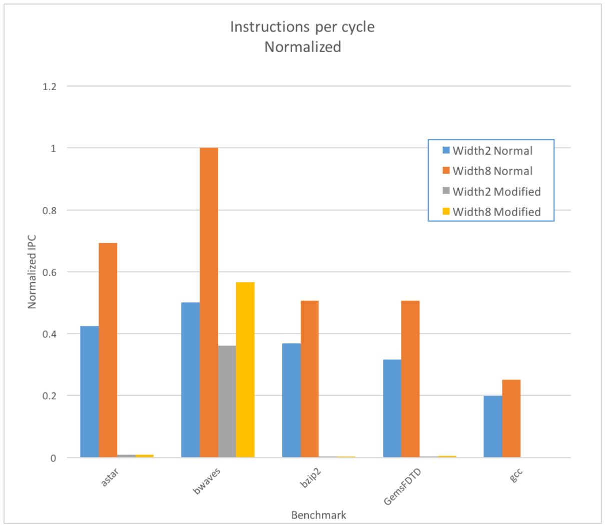

Fig. 11. Graph that compares the instructions per cycle or ipc (values are normalized to max value).

is

$$ED_L = \frac{(t_N - t_S)s\ (\frac{10^9 ns}{1s})}{\#of MemInst} \tag{1}$$

where ED_L is the latency in clock cycles of the encryption/decryption functions and where t_N and t_S are the simulation time in seconds of the normal CPU and secured CPU, respectively. The ED_L is important because it allows gem5 to include this latency in the Instructions Per Cycle (IPC) calculations.

After ED_L was calculated, it was added to the python configuration script and the secure CPU benchmarks were ran again. The results of the benchmarks are shown in Figures 11 and 12.

This set of graphs show results that are more interesting. It can be seen that for the majority of benchmarks, the instruction rate per cycle goes down dramatically. In fact, on some benchmarks it is almost impossible to see the value on the graph because it is so small.

There are two benchmarks that ended up not having a big decrease in instructions per cycle. These benchmarks are *bwaves* and *libquantum*. In order to understand why this is, the result text files where analyzed again. After analyzing the result files, the authors were able to conclude that the *bwaves* and *libquantum* have a lot less memory instructions executed compared to the other benchmarks. For example *mcf* and *gcc* have 64,884,724 and 52,280,113 memory instructions respectively, while *bwaves* and *libquantum* have 28,339,998 and 33,181,223 memory instructions respectively.

These results would also greatly depend on how the memory is referenced. If a certain benchmark references memory that is required to be accessed from disk instead of the cache, it will also effect the amount of instructions

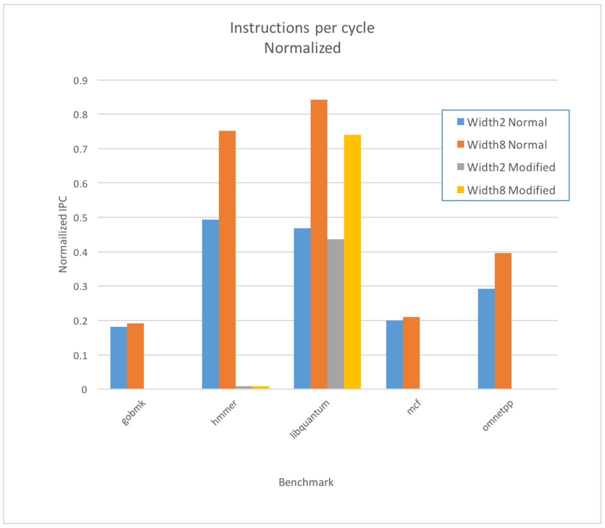

Fig. 12. Graph that compares the instructions per cycle or ipc (values are normalized to max value).

per cycle that is observed in the pipeline.

The results observed match the prediction stated at the beginning of the paper. The added security module does indeed cause the CPU to have a performance decrease. It is a little surprising that some of the benchmarks take a very large performance decrease. This shows that the type of application used greatly determines whether or not the security module is worth it.

It should be noted that the benchmark *bzip2* shows odd behavior in Figures 5, 7, and 9. The results of this benchmark should be taken lightly until it the odd results have been explained.

VI. Future Work

As mentioned in the subsection C of section IV, the CPU registers are currently not being protected with encryption. Given more time, the authors would like to upgrade the security module to include the register files. This would be an improvement with regards to security, but it will also affect the performance. If the encryption and decryption regarding the registers requires too much of a performance hit, it may end up not having an overall improvement. The effects of this added security would need to be tested and simulated in a future work.

Another improvement that can be done to the current security module is to optimize the DES source code. The code was written as a proof of concept, thus it is not optimized to run efficiently. Based on the results of this paper, the CPU has a performance decrease when the security module is introduced. In order to mitigate the performance decrease, the code needs to be optimized as much as possible.

VII. Conclusion

Looking at the results, the performance is nearly half of what the non-modified gem5 simulator produced, sometimes even below 10% of normal gem5 performance. This definitley brings into question, is a security module like this worth the performance hit? An important consideration for this is the application of the computer in use. Is it going to be handling secure data that can't get into the hands of adversaries while running? Is the person running the computer paranoid about people stealing their data? There are many questions to ask here and the added security isn't for every machine, especially one where number crunching is more important than securing the internal data.

During this project, the team learned many new things. Probably the most significant lesson learned was how to use gem5 and its structure for handling data between the pipeline and caches. In particular, the team learned the process of how a load and store instruction are executed in the O3 cpu model. Another thing the team learned during the project is how the gem5 simulator calculated its IPCs. It isn't dependent on the running time of the simulator but in line with the internal tick counter for the clock. The latency for each hit in the cache is more important to the final IPC count for the benchmark.

Whether or not the added security module shown and tested in this paper is the best method to secure modern computer architecture, the security of these systems should definitely be taken seriously.

References

[1] G. Edward Suh, C. W. O'Donnel, I. Sachdev, and S Devadas. Design and Implementation of the AEGIS Single-Chip Secure Processor Using Physical Random Functions. *Proceedings of the 32nd annual international symposium on Computer Architecture*, 2005.
[2] J. Orlin Grabbe. The DES Algorithm Illustrated. *Laissez Faire City Times*, 2006.
[3] A. Sadeghi and C. Wachsmann, "Physically Unclonable Functions: A New Generation of Security Hardware", Technische Universitat Darmstadt, 2012.
[4] M. Deutschman, "Cryptographic Applications with Physically Unclonable Functions," M.S. Thesis, Inst. Mathematics, Alpen-Adria-Universität Klagenfurt, Klagenfurt, Austria, 2010.
[5] C. Bhm, M. Hofer and W. Pribyl, "A Microcontroller SRAM-PUF", Institute for Electronics Graz University of Technology, 2011.

APPENDIX

```c
//----------------------------------------------------
//
//  Code: DES Encryption Header File
//  Authors: Tyler Travis & Justin Cox
//  Date: 3/23/16
//
//----------------------------------------------------

#ifndef DES_HH
#define DES_HH

#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>

extern const uint8_t IP[64];

extern const uint8_t FP[64];

extern const uint8_t E[48];

extern const uint8_t P[32];

extern const uint8_t PC_1[56];

extern const uint8_t PC_2[48];

extern const uint8_t ISV[16];

extern const uint8_t S_box1[4][16];

extern const uint8_t S_box2[4][16];

extern const uint8_t S_box3[4][16];

extern const uint8_t S_box4[4][16];

extern const uint8_t S_box5[4][16];

extern const uint8_t S_box6[4][16];

extern const uint8_t S_box7[4][16];

extern const uint8_t S_box8[4][16];

void encrypt(uint8_t *plain_text, uint16_t plain_text_size, uint8_t
*cipher_text, uint8_t key[8]);
void decrypt(uint8_t *plain_text, uint8_t *cipher_text, uint8_t key[8]);
void generate_subkeys(uint8_t key[8], uint8_t subkey[][6]);
void desRound(uint8_t leftHalve[], uint8_t rightHalve[], uint8_t subkey[6]);
void fFunction(uint8_t rightHalve[], uint8_t subKey[6]);
void copy_bit(uint8_t source[], uint8_t dest[], uint16_t source_bit, uint16_t
dest_bit);
void circular_shift_array(uint8_t array[4], uint8_t shift);
void combine_CD(uint8_t C[4], uint8_t D[4], uint8_t dest[7]);
void getPlainText(uint8_t plainText[], uint32_t genNum);

#endif




//-----------------------------------------------------
//
//  Code: DES Encryption C File
//  Authors: Tyler Travis & Justin Cox
//  Date: 3/23/16
//
//-----------------------------------------------------

#include "cpu/o3/des.hh"

const uint8_t IP[64] = {57, 49, 41, 33, 25, 17, 9,  1,
                        59, 51, 43, 35, 27, 19, 11, 3,
                        61, 53, 45, 37, 29, 21, 13, 5,
                        63, 55, 47, 39, 31, 23, 15, 7,
                        56, 48, 40, 32, 24, 16, 8,  0,
                        58, 50, 42, 34, 26, 18, 10, 2,
                        60, 52, 44, 36, 28, 20, 12, 4,
                        62, 54, 46, 38, 30, 22, 14, 6};

const uint8_t FP[64] = {39, 7, 47, 15, 55, 23, 63, 31,
                        38, 6, 46, 14, 54, 22, 62, 30,
                        37, 5, 45, 13, 53, 21 ,61, 29,
                        36, 4, 44, 12, 52, 20, 60, 28,
                        35, 3, 43, 11, 51, 19, 59, 27,
                        34, 2, 42, 10, 50, 18, 58, 26,
                        33, 1, 41, 9,  49, 17, 57, 25,
                        32, 0, 40, 8,  48, 16, 56, 24};

const uint8_t E[48] = {31, 0, 1, 2, 3, 4,
                       3, 4, 5, 6, 7, 8,
                       7, 8, 9, 10, 11, 12,
                       11, 12, 13, 14, 15, 16,
                       15, 16, 17, 18, 19, 20,
                       19, 20, 21, 22, 23, 24,
                       23, 24, 25, 26, 27, 28,
                       27, 28, 29, 30, 31, 0};

const uint8_t P[32] = {15, 6, 19, 20, 28, 11, 27, 16,
                       0, 14, 22, 25, 4, 17, 30, 9,
                       1, 7, 23, 13, 31, 26, 2, 8,
                       18, 12, 29, 5, 21, 10, 3, 24};

                         // Left
const uint8_t PC_1[56] = {56, 48, 40, 32, 24, 16, 8,
                          0, 57, 49, 41, 33, 25, 17,
                          9, 1, 58, 50, 42, 34, 26,
                          18, 10, 2, 59, 51, 43, 35,
                          // Right
                          62, 54, 46, 38, 30, 22, 14,
                          6, 61, 53, 45, 37, 29, 21,
                          13, 5, 60, 52, 44, 36, 28,
                          20, 12, 4, 27, 19, 11, 3};

const uint8_t PC_2[48] = {13, 16, 10, 23, 0, 4, 2, 27,
                          14, 5, 20, 9, 22, 18, 11, 3,
                          25, 7, 15, 6, 26, 19, 12, 1,
                          40, 51, 30, 36, 46, 54, 29, 39,
                          50, 44, 32, 47, 43, 48, 38, 55,
                          33, 52, 45, 41, 49, 35, 28, 31};

const uint8_t ISV[16] = {1,1,2,2,2,2,2,2,1,2,2,2,2,2,2,1};

const uint8_t S_box1[4][16] = {
    {14,4,13,1,2,15,11,8,3,10,6,12,5,9,0,7},
    {0,15,7,4,14,2,13,1,10,6,12,11,9,5,3,8},
    {4,1,14,8,13,6,2,11,15,12,9,7,3,10,5,0},
    {15,12,8,2,4,9,1,7,5,11,3,14,10,0,6,13}
};

const uint8_t S_box2[4][16] = {
    {15,1,8,14,6,11,3,4,9,7,2,13,12,0,5,10},
    {3,13,4,7,15,2,8,14,12,0,1,10,6,9,11,5},
    {0,14,7,11,10,4,13,1,5,8,12,6,9,3,2,15},
    {13,8,10,1,3,15,4,2,11,6,7,12,0,5,14,9}
};

const uint8_t S_box3[4][16] = {
    {10,0,9,14,6,3,15,5,1,13,12,7,11,4,2,8},
    {13,7,0,9,3,4,6,10,2,8,5,14,12,11,15,1},
    {13,6,4,9,8,15,3,0,11,1,2,12,5,10,14,7},
    {1,10,13,0,6,9,8,7,4,15,14,3,11,5,2,12}
};

const uint8_t S_box4[4][16] = {
    {7,13,14,3,0,6,9,10,1,2,8,5,11,12,4,15},
    {13,8,11,5,6,15,0,3,4,7,2,12,1,10,14,9},
    {10,6,9,0,12,11,7,13,15,1,3,14,5,2,8,4},
    {3,15,0,6,10,1,13,8,9,4,5,11,12,7,2,14}
};

const uint8_t S_box5[4][16] = {
    {2,12,4,1,7,10,11,6,8,5,3,15,13,0,14,9},
    {14,11,2,12,4,7,13,1,5,0,15,10,3,9,8,6},
    {4,2,1,11,10,13,7,8,15,9,12,5,6,3,0,14},
    {11,8,12,7,1,14,2,13,6,15,0,9,10,4,5,3}
};

const uint8_t S_box6[4][16] = {
    {12,1,10,15,9,2,6,8,0,13,3,4,14,7,5,11},
      {10,15,4,2,7,12,9,5,6,1,13,14,0,11,3,8},
      {9,14,15,5,2,8,12,3,7,0,4,10,1,13,11,6},
      {4,3,2,12,9,5,15,10,11,14,1,7,6,0,8,13}
};
```

```
const uint8_t S_box7[4][16] = {
    {4,11,2,14,15,0,8,13,3,12,9,7,5,10,6,1},
      {13,0,11,7,4,9,1,10,14,3,5,12,2,15,8,6},
      {1,4,11,13,12,3,7,14,10,15,6,8,0,5,9,2},
      {6,11,13,8,1,4,10,7,9,5,0,15,14,2,3,12}
};

const uint8_t S_box8[4][16] = {
    {13,2,8,4,6,15,11,1,10,9,3,14,5,0,12,7},
      {1,15,13,8,10,3,7,4,12,5,6,11,0,14,9,2},
      {7,11,4,1,9,12,14,2,0,6,10,13,15,3,5,8},
      {2,1,14,7,4,10,8,13,15,12,9,0,3,5,6,11}
};

void getPlainText(uint8_t plainText[], uint32_t genNum){
    plainText[4] = (genNum & 0xFF000000) >> 24;
    plainText[5] = (genNum & 0x00FF0000) >> 16;
    plainText[6] = (genNum & 0x0000FF00) >> 8;
    plainText[7] = (genNum & 0x000000FF);
    plainText[0] = 0;
    plainText[1] = 0;
    plainText[2] = 0;
    plainText[3] = 0;
}

void encrypt(uint8_t *plain_text, uint16_t plain_text_size, uint8_t
*cipher_text, uint8_t key[8])
{
    // 2d array to hold all the generated subkeys for DES
    uint8_t subkey[16][6];
    uint8_t plain_textIP[8] = {0};
    uint8_t cipher_textPreFP[8] = {0};

    uint8_t leftHalve[4] = {0};
    uint8_t rightHalve[4] = {0};
    uint8_t i;

    // Create the subkeys from the key
    generate_subkeys(key, subkey);

    // Send Input through IP (Initial Permutation)
    for(i = 0; i < 64; ++i)
    {
        copy_bit(plain_text, plain_textIP, IP[i], i);
    }

    // Split 64 bits into two 32 bit chunks (L & R)
    leftHalve[0] = plain_textIP[0];
    leftHalve[1] = plain_textIP[1];
    leftHalve[2] = plain_textIP[2];
    leftHalve[3] = plain_textIP[3];

    rightHalve[0] = plain_textIP[4];
    rightHalve[1] = plain_textIP[5];
    rightHalve[2] = plain_textIP[6];
    rightHalve[3] = plain_textIP[7];
    // Round 1 through 16
    for(i = 0; i < 16; i++)
    {
        desRound(leftHalve, rightHalve, subkey[i]);
    }

    // Recombine final Left and Right Halves
    cipher_textPreFP[0] = rightHalve[0];
    cipher_textPreFP[1] = rightHalve[1];
    cipher_textPreFP[2] = rightHalve[2];
    cipher_textPreFP[3] = rightHalve[3];

    cipher_textPreFP[4] = leftHalve[0];
    cipher_textPreFP[5] = leftHalve[1];
    cipher_textPreFP[6] = leftHalve[2];
    cipher_textPreFP[7] = leftHalve[3];

    // Send 64 bit recombination into FP (Final Permutation)
    for(i = 0; i < 64; i++){
        copy_bit(cipher_textPreFP, cipher_text, FP[i], i);
    }

  // printf("Done with encryption ");

    //End of Function
}

void decrypt(uint8_t *plain_text, uint8_t *cipher_text, uint8_t key[8])
{
    // 2D array to hold subkeys
    uint8_t subkey[16][6];
    uint8_t cipher_textIP[8] = {0};
    uint8_t plain_textPreFP[8] = {0};

    uint8_t leftHalve[8] = {0};
    uint8_t rightHalve[8] = {0};

    uint16_t i;

    // Generate the subkeys
    generate_subkeys(key, subkey);

    for(i = 0; i < 64; i++){
        copy_bit(cipher_text, cipher_textIP, IP[i], i);
    }

    // Split 64 bits into two 32 bit chunks (L & R)
    leftHalve[0] = cipher_textIP[0];
    leftHalve[1] = cipher_textIP[1];
    leftHalve[2] = cipher_textIP[2];
    leftHalve[3] = cipher_textIP[3];

    rightHalve[0] = cipher_textIP[4];
    rightHalve[1] = cipher_textIP[5];
    rightHalve[2] = cipher_textIP[6];
    rightHalve[3] = cipher_textIP[7];
    // Round 1 through 16
    for(i = 0; i < 16; i++)
    {
        desRound(leftHalve, rightHalve, subkey[15 - i]);
    }

    // Recombine final Left and Right Halves
    plain_textPreFP[0] = rightHalve[0];
    plain_textPreFP[1] = rightHalve[1];
    plain_textPreFP[2] = rightHalve[2];
    plain_textPreFP[3] = rightHalve[3];

    plain_textPreFP[4] = leftHalve[0];
    plain_textPreFP[5] = leftHalve[1];
    plain_textPreFP[6] = leftHalve[2];
    plain_textPreFP[7] = leftHalve[3];

    // Send 64 bit recombination into FP (Final Permutation)
    for(i = 0; i < 64; i++){
        copy_bit(plain_textPreFP, plain_text, FP[i], i);
    }

   // printf("Decryption Done ");
    //End of Function
}

void generate_subkeys(uint8_t key[8], uint8_t subkey[][6])
{
    // K+ permuted key array, Use the PS_1 permutation array to move the bits
around
    uint8_t permuted_key[7] = {0};
    uint32_t i, j;
    uint8_t C[17][4];
    uint8_t D[17][4];
    uint8_t temp_array[7];
    //uint8_t temp;

    //printf("%x%x %x%x %x%x %x%x\n", key[0], key[1], key[2], key[3], key[4],
key[5], key[6], key[7]);

    for(i = 0; i < 56; ++i)
    {
        copy_bit(key, permuted_key, PC_1[i], i);
    }


    //printf("%x%x %x%x %x%x %x\n", permuted_key[0], permuted_key[1],
permuted_key[2],
    //         permuted_key[3], permuted_key[4], permuted_key[5],
permuted_key[6]);

    // Initial setup for splitting the key
    // For the C array:
    //      The mask: 0x0FFFFFFF should perserve all data
    //      That is the reason for the shifting
    // For the D array:
    //      Everything already aligns well
    C[0][0] = permuted_key[0] >> 4 & 0x0F;
    C[0][1] = (permuted_key[1] >> 4 & 0x0F) | permuted_key[0] << 4;
    C[0][2] = (permuted_key[2] >> 4 & 0x0F) | permuted_key[1] << 4;
    C[0][3] = (permuted_key[3] >> 4 & 0x0F) | permuted_key[2] << 4;

    D[0][0] = permuted_key[3] & 0x0F;
    D[0][1] = permuted_key[4];
    D[0][2] = permuted_key[5];
    D[0][3] = permuted_key[6];

    // Generate the next 15 C-D pairs by circular shifting
    // using the ISV array
    for(i = 1; i < 17; ++i)
    {
        // Copy the previous C and D to the current C and D
        // Then perform the shifting on these
        memcpy(C[i], C[i-1], sizeof(C[i]));
        memcpy(D[i], D[i-1], sizeof(D[i]));
        circular_shift_array(C[i], ISV[i-1]);
        circular_shift_array(D[i], ISV[i-1]);

        // Combine the C and D arrays to the temp_array
        combine_CD(C[i], D[i], temp_array);

        // Use PC_2 to get the correct subkey
        // Need to start the subkey at index 0
        // Hence the i-1 subscript
        for(j = 0; j < 48; ++j)
        {
            copy_bit(temp_array, subkey[i-1], PC_2[j], j);
        }
    }
}

void desRound(uint8_t leftHalve[], uint8_t rightHalve[], uint8_t subkey[6]){

    uint8_t rightTemp[4];

    //Keep copy of R_i
    rightTemp[0] = rightHalve[0];
    rightTemp[1] = rightHalve[1];
    rightTemp[2] = rightHalve[2];
    rightTemp[3] = rightHalve[3];

    //Send R_i and subKey into fFuntion()
    fFunction(rightHalve, subkey);

    //XOR Output of fFunction() with L_i
    rightHalve[0] = rightHalve[0] ^ leftHalve[0];
    rightHalve[1] = rightHalve[1] ^ leftHalve[1];
    rightHalve[2] = rightHalve[2] ^ leftHalve[2];
    rightHalve[3] = rightHalve[3] ^ leftHalve[3];

    //Make L_i+1 = R_i for next round
    leftHalve[0] = rightTemp[0];
    leftHalve[1] = rightTemp[1];
    leftHalve[2] = rightTemp[2];
    leftHalve[3] = rightTemp[3];

    //End of Function
}

void fFunction(uint8_t rightHalve[], uint8_t subKey[6]){

    uint8_t rightHalveE[6] = {0};
    uint8_t rowBits = 0;
    uint8_t colBits = 0;
    uint8_t post_Sbox_R[4] = {0};

    uint32_t i;

    //Send rightHalve to E permutation
    for(i = 0; i < 48; i++){
        copy_bit(rightHalve, rightHalveE, E[i], i);
    }

    //XOR subKey with output of E
    for(i = 0; i < 6; i++){
        rightHalveE[i] = rightHalveE[i] ^ subKey[i];
    }

    //printf("E: %02x%02x %02x%02x %02x%02x\n", rightHalveE[0],
rightHalveE[1], rightHalveE[2],
    //         rightHalveE[3], rightHalveE[4], rightHalveE[5]);

    //Send output of XOR into Switch Boxes
    //-----------------------------------------------------------------------
-
    // 1111 1111 | 1111 1111 | 1111 1111 | 1111 1111 | 1111 1111 | 1111 1111
    // ^      ^         ^        ^         ^      ^         ^        ^
    // 0      6         4        2         0      6         4        2
    //-----------------------------------------------------------------------
-
    //*************
    //  Sbox1
    //*************
    rowBits = (rightHalveE[0] & 0x80) >> 6;
    rowBits |= (rightHalveE[0] & 0x04) >> 2;

    colBits = (rightHalveE[0] & 0x78) >> 3;

    post_Sbox_R[0] = (S_box1[rowBits][colBits] << 4);
    //*************
    //  Sbox2
    //*************
    rowBits = (rightHalveE[0] & 0x02);
    rowBits |= (rightHalveE[1] & 0x10) >> 4;

    colBits = (rightHalveE[0] & 0x01) << 3;
    colBits |= (rightHalveE[1] & 0xE0) >> 5;

    post_Sbox_R[0] |= S_box2[rowBits][colBits];
    //printf("Sbox1&2: %02x \n", post_Sbox_R[0]);
    //*************
```

```
//  Sbox3
//*************
rowBits = (rightHalveE[1] & 0x08) >> 2;
rowBits |= (rightHalveE[2] & 0x40) >> 6;

colBits = (rightHalveE[1] & 0x07) << 1;
colBits |= (rightHalveE[2] & 0x80) >> 7;

post_Sbox_R[1] = (S_box3[rowBits][colBits] << 4);
//*************
//  Sbox4
//*************
rowBits = (rightHalveE[2] & 0x20) >> 4;
rowBits |= (rightHalveE[2] & 0x01);

colBits = (rightHalveE[2] & 0x1E) >> 1;

post_Sbox_R[1] |= S_box4[rowBits][colBits];
//printf("Sbox3&4: %02x \n", post_Sbox_R[1]);
//*************
//  Sbox5
//*************
rowBits = (rightHalveE[3] & 0x80) >> 6;
rowBits |= (rightHalveE[3] & 0x04) >> 2;

colBits = (rightHalveE[3] & 0x78) >> 3;

post_Sbox_R[2] = (S_box5[rowBits][colBits] << 4);
//*************
//  Sbox6
//*************
rowBits = (rightHalveE[3] & 0x02);
rowBits |= (rightHalveE[4] & 0x10) >> 4;

colBits = (rightHalveE[3] & 0x01) << 3;
colBits |= (rightHalveE[4] & 0xE0) >> 5;

post_Sbox_R[2] |= S_box6[rowBits][colBits];
//printf("Sbox5&6: %02x \n", post_Sbox_R[2]);
//*************
//  Sbox7
//*************
rowBits = (rightHalveE[4] & 0x08) >> 2;
rowBits |= (rightHalveE[5] & 0x40) >> 6;

colBits = (rightHalveE[4] & 0x07) << 1;
colBits |= (rightHalveE[5] & 0x80) >> 7;

post_Sbox_R[3] = (S_box7[rowBits][colBits] << 4);
//*************
//  Sbox8
//*************
rowBits = (rightHalveE[5] & 0x20) >> 4;
rowBits |= (rightHalveE[5] & 0x01);

colBits = (rightHalveE[5] & 0x1E) >> 1;
```

```
    post_Sbox_R[3] |= S_box8[rowBits][colBits];
    //printf("Sbox7&8: %02x \n", post_Sbox_R[3]);

    //Send output of Switch Boxes into P permutation
    for(i = 0; i < 32; i++){
      copy_bit(post_Sbox_R, rightHalve, P[i], i);
    }

    //printf("P: %02x%02x %02x%02x\n", rightHalve[0], rightHalve[1],
rightHalve[2],
    //        rightHalve[3]);

    //End of Function

}

void copy_bit(uint8_t source[], uint8_t dest[], uint16_t source_bit, uint16_t
dest_bit)
{
    // Find which byte to look in
    uint8_t source_byte = source_bit/8;

    // Get the bit offset relative to the byte
    uint8_t source_bit_offset = 7-source_bit%8;

    // Get the data from at the byte
    uint8_t source_byte_data = source[source_byte];

    // Get what the bit is
    uint8_t source_bit_data = ((0x1 << source_bit_offset) & source_byte_data)
>> source_bit_offset;

    // Find which byte to store the value in
    uint8_t dest_byte = dest_bit/8;

    // Get the bit taht we need to store the value in
    uint8_t dest_bit_offset = 7-dest_bit%8;

    // Get the byte data
    uint8_t dest_byte_data = dest[dest_byte];

    // mask for the bit destination
    uint8_t dest_mask;

    // the bit data for the destination
    uint8_t dest_bit_data = source_bit_data << dest_bit_offset;

    // If the the bit needs to be a 1,
    // then the final byte data just needs to be ORed
    // for it to work correctly
    if(source_bit_data == 0x1)
    {
        dest_mask = 0x0;
        dest_byte_data = dest_byte_data | (dest_mask | dest_bit_data);
    }
    // If the bit needs to be a 0, then we need to and it with a mask
    // of 0x11..0..11 where the 0 is the position where the zero needs
    // to be.
    else
    {
        dest_mask = ~(0x1 << dest_bit_offset);
        dest_byte_data = dest_byte_data & dest_mask;
    }

    // Store the data back into the array
    dest[dest_byte] = dest_byte_data;
}

void circular_shift_array(uint8_t array[4], uint8_t shift)
{
    // The temp array holds the values that need to be
    // moved between indices
    uint8_t temp[4];
    // For DES, there are only two options for this
    // shifting step: shift left by 1 or shift left
    // by 2.
    if(shift == 1)
    {
        // 0bxxxx xxxx.xxxx xxxx.xxxx xxxx.xxxx xxxx
        //        ^    |         |         |         <- temp[0]
        //             ^         |         |         <- temp[1]
        //                       ^         |         <- temp[2]
        //                                 ^         <- temp[3]
        temp[0] = (array[0] & 0x08) >> 3;
        temp[1] = (array[1] & 0x80) >> 7;
        temp[2] = (array[2] & 0x80) >> 7;
        temp[3] = (array[3] & 0x80) >> 7;

        // This performs the shifting, and carries over the
        // bits that would have been outside of the operation
        // for array[0], ANDing with 0x0F perserves the structure
        // of the array.
        array[0] = (array[0] << shift & 0x0F) | temp[1];
        array[1] = (array[1] << shift) | temp[2];
        array[2] = (array[2] << shift) | temp[3];
        array[3] = (array[3] << shift) | temp[0];
    }
    else if(shift == 2)
    {
        // 0bxxxx xxxx.xxxx xxxx.xxxx xxxx.xxxx xxxx
        //        ^^   ||        ||        ||        <- temp[0]
        //             ^^        ||        ||        <- temp[1]
        //                       ^^        ||        <- temp[2]
        //                                 ^^        <- temp[3]
        temp[0] = (array[0] & 0x0C) >> 2;
        temp[1] = (array[1] & 0xC0) >> 6;
        temp[2] = (array[2] & 0xC0) >> 6;
        temp[3] = (array[3] & 0xC0) >> 6;

        array[0] = (array[0] << shift & 0x0F) | temp[1];
        array[1] = (array[1] << shift) | temp[2];
        array[2] = (array[2] << shift) | temp[3];
        array[3] = (array[3] << shift) | temp[0];
    }
```

```
}

void combine_CD(uint8_t C[4], uint8_t D[4], uint8_t dest[7])
{
    // Combine the C and D array into one array.
    // The tricky part here is that the first element
    // of the C and D array are padded with 4 bits of 0's.
    // The shifting, ORing, and ANDing take care of this
    // for the C array. The D array is already aligned correctly
    // once we get to the second element
    dest[0] = (C[0] << 4) | ((C[1] >> 4) & 0x0F);
    dest[1] = (C[1] << 4) | ((C[2] >> 4) & 0x0F);
    dest[2] = (C[2] << 4) | ((C[3] >> 4) & 0x0F);
    dest[3] = (C[3] << 4) | (D[0] & 0x0F);
    dest[4] = D[1];
    dest[5] = D[2];
    dest[6] = D[3];
}
```