MEMORY ENCRYPTION TO PROTECT PROCESSES AND DATA

A Project Report

submitted by

PRANJAL JAIN

in partial fulfilment of the requirements
for the award of the degree of

MASTER OF TECHNOLOGY AND BACHELOR OF TECHNOLOGY

DEPARTMENT OF ELECTRICAL ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY MADRAS.

JUNE 2022

THESIS CERTIFICATE

This is to certify that the thesis titled **MEMORY ENCRYPTION TO PROTECT PROCESSES AND DATA**, submitted by **PRANJAL JAIN**, to the Indian Institute of Technology, Madras, for the award of the degree of **MASTER OF TECHNOLOGY AND BACHELOR OF TECHNOLOGY**, is a bona fide record of the research work done by him under our supervision. The contents of this thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Prof. Chester Rebeiro
Project Guide
Associate Professor
Dept. of Computer Science & Engineering
IIT Madras, 600 036

Prof. Nitin Chandrachoodan
Project Co-Guide
Associate Professor
Dept. of Electrical Engineering
IIT Madras, 600 036

Place: Chennai

Date: 17 June 2022

ACKNOWLEDGEMENTS

I would firstly like to thank my project guide Prof. Chester Rebeiro for giving me the opportunity to work in this field and explore it. His insights have been instrumental in the completion of this project. I would also like to thank my co-guide Prof. Nitin Chandrachoodan for his constant support. I would further like to extend my gratitude to Arjun who has constantly guided me throughout the project. This project would not have been possible without his patient explanations and help.

Lastly, I would like to thank my friends, for their help and support, and family, who kept pushing me to work hard throughout.

ABSTRACT

KEYWORDS: Cloud computing, Prince, Memory encryption, Side channel, Cold boot attack, ELF, SHAKTI, Key management.

The increasing popularity of cloud computing has led to a large number of users, businesses and organizations shift to cloud and web services in order to satisfy their requirements for higher storage and computation power. This could, however, prove to be a double edged sword which makes your data vulnerable to attacks from other users as well as the service provider. In this thesis, I present Prince, a memory encryption engine, to mitigate and prevent these risks so as to make the hardware more secure and immune to various cyber attacks. Prince ensures that all the data leaving the L1 cache is encrypted, thereby making it impossible for other users to make sense of the data, even in case they gain access to it.

TABLE OF CONTENTS

LIST OF FIGURES

ABBREVIATIONS

PT	Plain Text
GCM	Galois/Counter Mode
OS	Operating System
ELF	Executable Linker Format
RAM	Random Access Memory
GDB	GNU Debugger
L1	Level 1
FPGA	Field Programmable Gate Array
PC	Program Counter

CHAPTER 1

INTRODUCTION

The last few years have seen a rapid movement towards cloud computing in various spheres. Businesses like banking, IT and trading have found it more convenient to shift to cloud services instead of housing their own data centers and servers due to the reduction in cost, as a result of a pay-per-use model, scalability, wherein the amount of computing power or space they have can be varied as per their need at a short notice, and the overall benefit of avoiding the hassle of maintaining the infrastructure. Other users on the other hand find cloud services an attractive alternative as it allows them to access their data on the go from anywhere in the world.

However, while this solution does look cheap, easy and hassle free, it comes with the possibility of an increased risk of others gaining access to your data. Since cloud services effectively involve multiple users, isolated in different virtual environments, using the same hardware, it opens them up to data leaks and breach of privacy. Further, all the data could be vulnerable if the service provider itself is untrustworthy. To eliminate the possibility of side channel attacks from other users or cold boot attacks from the service provider itself, in this work, I present a memory encryption engine and its implementation.

CHAPTER 2

BACKGROUND

In this chapter we aim to explain a couple of attacks that users are more vulnerable to in a cloud environment. We also explain where Prince comes in and how it is a part of the solution.

2.1 Side channel attacks

Side channel attacks aim to gain information about the data or computations being performed by another user through various observations and readings of the physical state of the system or the signals produced by it. For example, while an adversary sharing hardware resources with a victim can gain information about the cache by filling it up repeatedly and noting the blocks which are emptied when the victim's program runs, an adversary in physical proximity to the victim's system on the other hand can monitor the power and radiation signals emitted by it and gain useful data.

These attacks usually aim at exfiltrating the information that the victim program has rather than change the course of its execution. With gadgets of higher accuracy in their readings being more readily available in recent times, these attacks have become easier to carry out and hence, an important part of our threat model.

2.1.1 Cold boot attacks

Cold boot attack is a type of a side channel attack that involves an adversary that has physical access to the victim's RAM. While contents in the RAM are supposed to decay within a few seconds, the charge can be made to stay on the capacitor for much longer periods of time using freeze spray. Further, even once decayed, bits can be reconstructed as they fade away in a predictable manner. The attacker is then free to dump the contents of the memory in another system and analyze it to extract information from it.

2.2 Prince

Keeping in mind that adding software also adds to the area that is exposed and vulnerable to attacks, software solutions alone would not suffice to solve the problem. Furthermore, while choosing hardware solutions, we need to be mindful that it does not affect the overall performance of the processor, thereby, beating the purpose.

Hence, we propose Prince, a lightweight cipher that will be used to encrypt and decrypt all the data leaving and entering the processor respectively. The basic concept is such that all data leaving the L1 cache (which is private to a processor) will be encrypted and any block of data being loaded into the L1 cache will be decrypted just before it. Hence, only when the data is present in the processor, it is in decrypted/plain text form and at all other times, it is encrypted. This ensures that even if an adversary was to get their hands on the data, it would be encrypted and impossible to decipher without the cipher itself.

CHAPTER 3

RELATED WORK

This chapter contains some brief details about similar concepts having been implemented and documented in other papers. While some of them elaborate on the encryption itself, most of them provide details about the key management.

3.1 Encryption schemes

Yin *et al.* (2020) mentions the use of AES-GCM architecture. It uses 4KB blocks to encrypt data to reduce the frequency at which the data needs to be encrypted. We, however, will be encrypting data at a much smaller boundary to ensure more security. Further, the paper also mentions usage of a random number generator and counters in their encryption scheme.

3.2 Key management

Yin *et al.* (2020), Gueron (2016*a*) and Gueron (2016*b*) mention an integrity tree scheme (a specific implementation of Merkel Trees) which is used to read and verify; and write and update data as and when required with the help of the key stored on chip securely. SHAKTI itself does have an integrity tree and the same concepts and resources can be used for this purpose as well. This is one of the future scopes that can be implemented along with Prince once it is completely ready and integrated. More on this is elaborated in 6

CHAPTER 4

OVERVIEW

Here I will be elaborating on how Prince works, it's placement within the processor, its advantages and encryption of ELFs in detail.

4.1 Prince encryption

The Prince cipher targets low latency. It has a 128 bit key k divided into 2 parts k_0 and k_1, and encrypts in blocks of 64 bits. The encryption is done as shown below:

$$k := k_0||k_1$$

$$\alpha := k_{0,63}k_{0,62}...k_{0,0}k_{1,63}k_{1,62}...(k_{1,1} \bigoplus k_{1,0})$$

$$k_0^{'} := k_{0,0}k_{0,63}k_{0,62}...(k_{0,1} \bigoplus k_{0,63})$$

$$k^{'} := k_0||k_0^{'}||k_1$$

Steps for encryption

1. XOR the input with k_0
2. Apply the core function with k_1
3. XOR its output with $k_0^{'}$

Steps for decryption

1. XOR the input with $k_0^{'}$
2. Apply the core function with $(k_1 \bigoplus \alpha)$
3. XOR its output with k_0

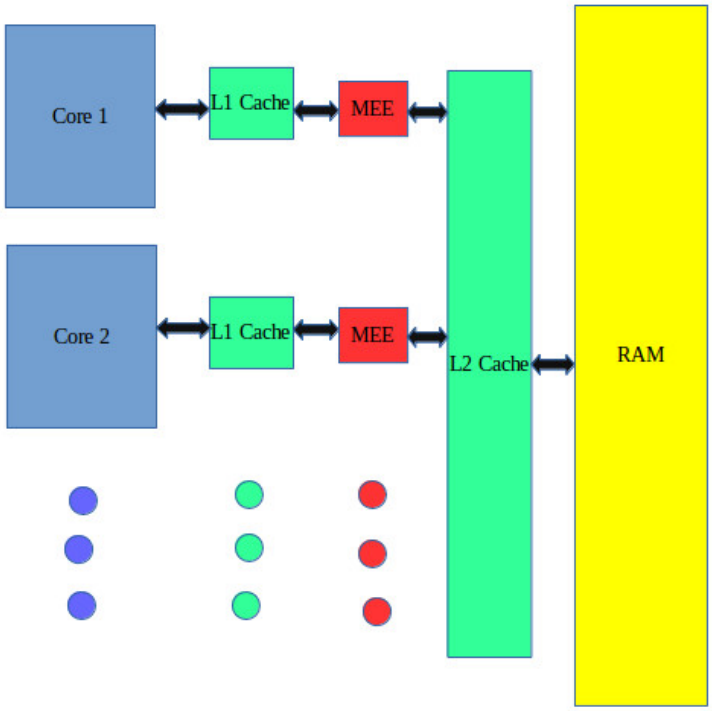

Figure 4.1: This figure shows the general concept of Prince. In SHAKTI, however, we do not have a L2 cache and hence, Prince is placed directly between the L1 cache and the memory

Core function

It has 3 parts

1. It consists of 5 forward rounds that take a cycle each.
2. It has one middle round which takes 2 cycles. This round uses a 4-bit sandbox
3. It again has 5 backward rounds which take a cycle each. This consists of multiplication by a 64x64 matrix M 0 and a shift row similar to the one in AES but operating on 4-bit nibbles rather than bytes.
4. Overall it takes 12 cycles to encrypt or decrypt a block. This also means that if we have 12 such modules in hardware, no block has to wait to get encrypted or decrypted as there will always be a Cipher ready to service it.

4.2 Changes to SHAKTI

The Prince module is placed between the L1 cache and the memory in SHAKTI since there is no L2 cache in SHAKTI. Appropriately, all the relevant cache modules were modified so as to instantiate Prince modules within them. Then the rules were modified

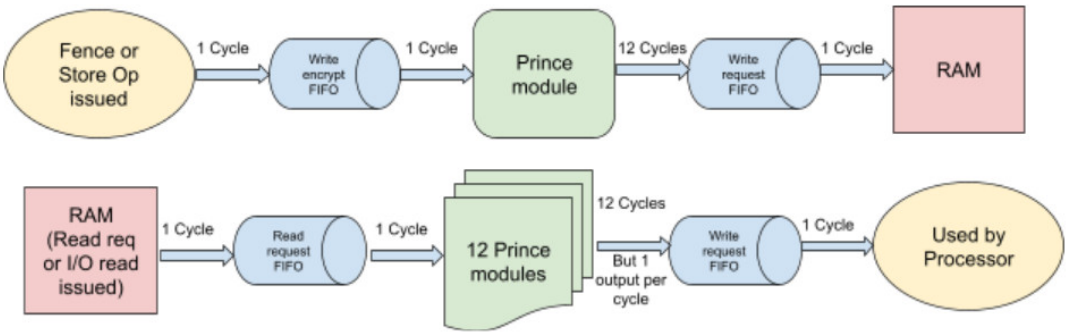

Figure 4.2: This figure shows the placement of Prince in SHAKTI.

to encrypt any data leaving the cache module and decrypt any data just being loaded into it. More details on this are given in A.1

4.3 Advantages

The Prince module being used for encryption and decryption has a few advantages as compared to other MEEs.

- It is a lightweight and fast module.
  - It requires no warm-up phase and hence, can start the encryption instantaneously.
  - If implemented in modern chip technology, low delays resulting in moderately high clock rates can be achieved.
- It is relatively compact and low cost.
  - PRINCE circuit's large parts of the implementation can be used both for encryption and decryption, reducing the area of the circuit on chip and making it favourable for small area chips.
  - For the same time constraints and technologies, PRINCE uses 6-7 times less area than PRESENT-80 and 14-15 times less area than AES-128.
- Prince cipher is quite robust against various attacks despite the fact of the same circuit for encryption and decryption.

4.4 ELF encryption

Any program passing through to the L1 cache gets decrypted. Also, it is the job of the ELF loader, which is also a program, to read the binary and make it ready for execution.

Since the ELF is the data for the loader program, which is decrypted at a layer before it, if we supply a regular ELF file to the loader, it will read its decrypted version which will be gibberish.

Hence, the loader as well as the nature of the ELF needed to be changed. The ELF has a main encryption as well as a secondary encryption and one of these 2 layers of encryption are decrypted by the Prince Cipher and the loader.

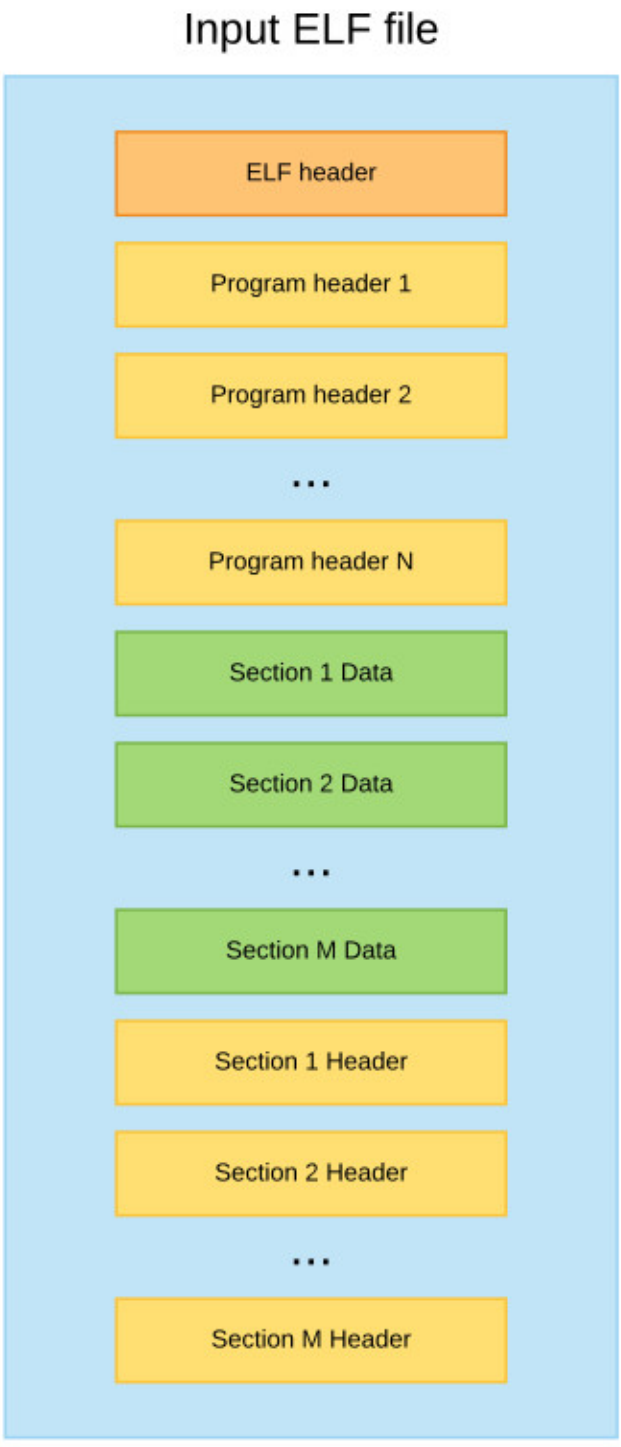

Figure 4.3: This figure shows an original ELF file

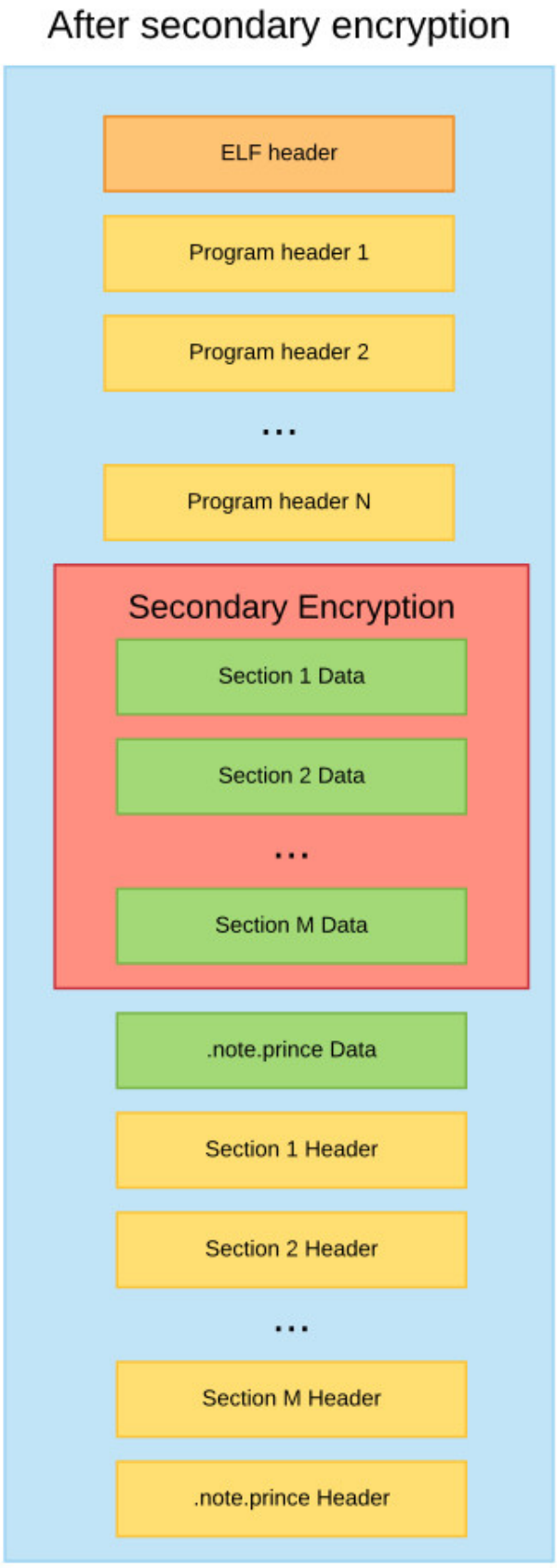

Figure 4.4: This figure shows an ELF file with first layer of encryption. The last section is added in this step.

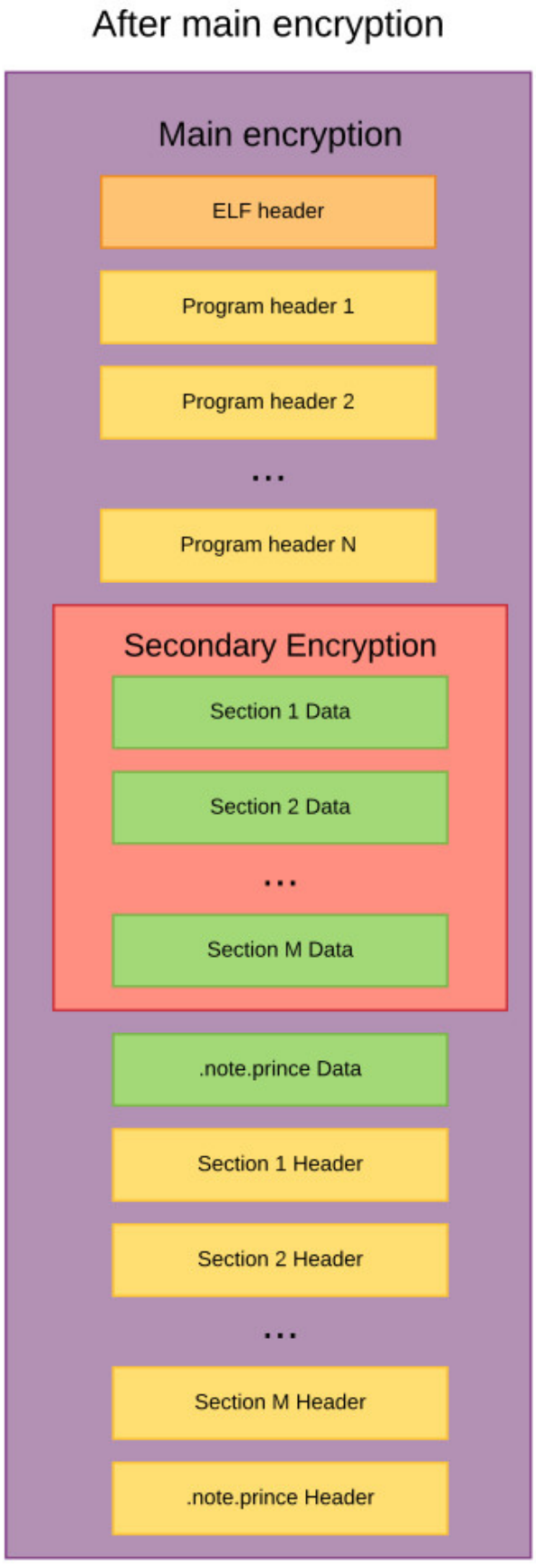

Figure 4.5: This figure shows an ELF file with both layers of encryption.

CHAPTER 5

TESTING AND RESULTS

In this chapter I will cover various methods through which we have tried to test and verify the working of Prince with SHAKTI. I will also include the various issues and challenges faced in the process.

5.1 Running encrypted ELF on SHAKTI

5.1.1 Running normal programs on SHAKTI

Usually the method to run programs on SHAKTI is as follows:

1. Compile the code, generate the boot files and link verilator.
2. Run make hello which compiles the hello.c, generates its elf and uses elf2hex to generate a code.mem.
3. Copy over the code.mem to the folder that contains the SHAKTI executable and run ./out.

Since SHAKTI does not have an OS, it is not possible to print anything on the terminal using printf. It runs the code on bare metal, generates the output and stores it in a file called app_log, which, in this case, will have the output “hello world”. More information on the setup is given in B.1.

5.1.2 Encrypting the ELF

Our approach here was to encrypt a simple hello world ELF using a cpp script and then run it on the modified SHAKTI to verify if the entire setup works. More details on this script are included in A.2 and B.2.The approach would have been the following:

1. The script only performs one level of encryption on the ELF, i.e., it only encrypts the sections and adds another note section. This is because we are directly going to be loading this on SHAKTI instead of fetching it from the RAM where it would be in double encrypted form.

```
pranjal@pranjal-Inspiron-15-3567:~/DDP/TESTING/c-class/benchmarks$ make encrypt
encrypting Hello-Shakti
elf2hex $((64/8)) 4194304*2 output/hello.riscv.enc 2147483648 > output/code.mem
elf2hex: ../fesvr/elfloader.cc:89: std::map<std::__cxx11::basic_string<char>, lo
ng unsigned int> load_elf(const char*, memif_t*, reg_t*): Assertion `strnlen(shs
trtab + from_le(sh[i].sh_name), max_len) < max_len' failed.
Aborted (core dumped)
Makefile:56: recipe for target 'encrypt' failed
make: *** [encrypt] Error 134
```

Figure 5.1: The error faced in this procedure.

2. Using this encrypted ELF, we would generate a code.mem through elf2hex.
3. This would then be run on SHAKTI the same way as mentioned above.

5.1.3 Issues

Ultimately we were not able to use this method to completion since we faced errors in step 2 which involves generating the code.mem. Changing the parameters and debugging the ELF also did not reveal the reason for the error. Hence, this procedure had to be abandoned since it was not possible to continue the testing without the code.mem.

5.2 Running encrypted ELF on SHAKTI with GDB

Since the generation of the code.mem from the encrypted ELF was causing the problem, our second approach involved running the encrypted ELF directly on the modified SHAKTI using gdb.

5.2.1 Running procedure

Various parameters need to be changed before compiling the SHAKTI code to include support for debugging with GDB. In the end, while linking verilator, we need to link it with GDB as well. More information on this is present in the B.3.

5.2.2 Issues with GDB

One of our first observations was that GDB was not able to run the encrypted ELF as it is. This was because GDB uses the section names which had all been encrypted while

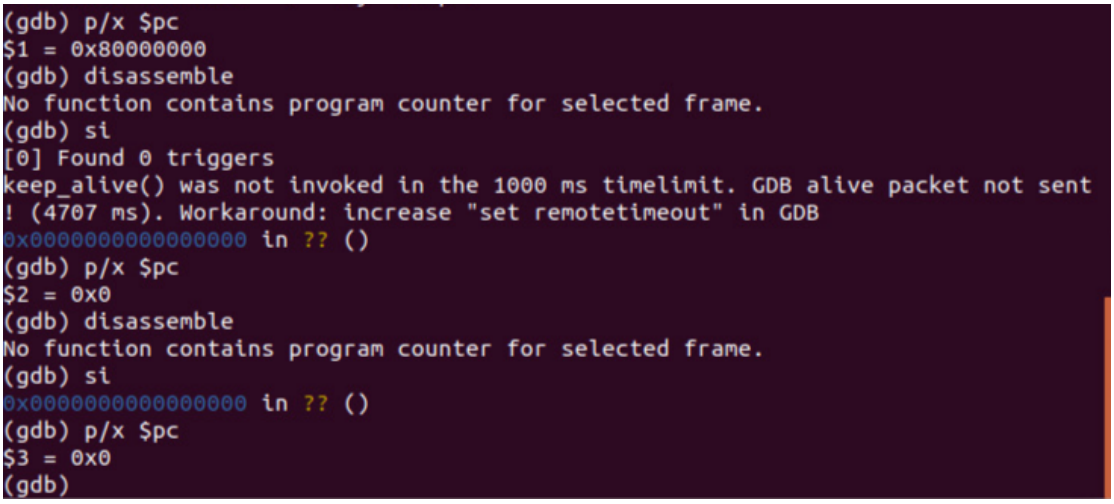

Figure 5.2: The error faced in this procedure.

encrypting the contents of the ELF. Leaving the last section of the ELF (apart from the note section) unencrypted solved this issue since it contained all the section names.

However, with all the function names and contents encrypted, it was still not possible to debug the ELF effectively since GDB was not able to identify the functions or display the instructions that would be executed. Further, the PC jumps to 0x0 after the first instruction executes and then gets stuck there. Again, this method had to be abandoned since it did not yield any information.

5.3 Running encrypted code.mem on SHAKTI

5.3.1 Encrypting the code.mem

The third approach used to try and test Prince with SHAKTI is as follows:

1. Remove the encryption part from the script used above. This ensures that the script just generates a copy of the ELF with all sections size aligned to 8 byte boundary.
2. Use this ELF to generate a code.mem. Then use another script to encrypt the code.mem by taking 8 bytes at a time. Snippets from this code can be found in A.3
3. Run this code.mem on SHAKTI as mentioned above.

Running this code.mem directly on SHAKTI does not yield the expected result and it goes into an infinite loop. Running it on GDB did not yield the desired results either.

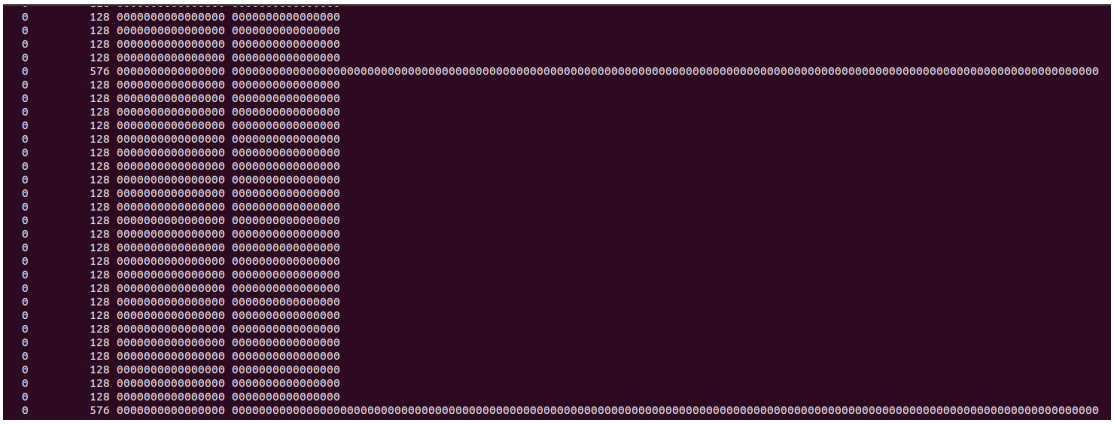

Figure 5.3: The error faced in this procedure.

The procedure to do the same are given in B.3.

5.4 Running on Spike

While some work on this had been done by previous students, we discovered quite a few issues with setting up Spike to run the ELFs for our purpose.

Firstly Spike does not have a cache memory. The architecture is quite direct and adding cache to it is not straightforward. Further, the ELF alignment would also cause issues in Spike. To top it off, as has been reported by previous students, running the entire setup on Spike would be quite slow and inefficient. It would become significantly different from the original environment that we would want to run it in. Hence, this method was abandoned after elaborate discussions.

CHAPTER 6

FUTURE SCOPE

This chapter will go over the work that can be carried forward with respect to Prince and how the design and implementation can be improved further.

6.1 Testing and verification

Currently the testing of Prince is yet to be completed in verilator. The next step would be to test it and run the same on FPGA. Once this is done, the testing can be deemed completed and work can be done on improving the design and verifying it against attacks.

6.2 Key management

Currently one single key is being used to encrypt and decrypt all the data. Once this can be verified to be working, work would need to be done on different programs having different keys. The storage and security of keys also needs to be looked at and improved.

CHAPTER 7

CONCLUSION

In conclusion, the concept of a memory encryption engine can be absolutely vital as far as mitigation techniques are concerned with respect to side channel attacks. While new, there has been a significant amount of work done in this field in the recent years.

While a memory encryption engine does improve the overall safety of the system, the challenge arises when trade-offs between security and performance are considered. A fine grained encryption might lead to a higher latency or more number of clock cycles which might beat the purpose altogether. A design that can strike a balance between the two is the need of the hour.

While Prince, our choice for the memory encryption engine, is designed, tested standalone and integrated with SHAKTI, the testing yet remains. Testing Prince in this environment and vetting it against side channel attacks created by us is the way to move forward.

APPENDIX A

CODE

Here I will be including code and other important information. All the code for this project can be found here.

A.1 Changes to cache modules in SHAKTI

This section explains the changes made to the SHAKTI cache code and how to understand them.

```
  `ifdef MEM_ENC_1
    //module instantiations and temporary value declaration/
   instantiation  //CLEAN LATER
    //for Prince cipher
    Reg#(int) encr_write_pending <- mkReg(0);
    Reg#(int) dec_read_pending <- mkReg(0);
    Reg#(int) dec_read_ctr <- mkReg(0);
    Reg#(int) dec_read_get_ctr <- mkReg(0);
    Integer my_blocksize = valueOf(`dblocks);
    //Integer my_blocksize = blocksize;
    Modular_prince_ifc #(`paddr, TMul#(`dblocks, TMul#(`dwords, 8)))
   prince <- mkmodular_prince;
    Modular_prince_ifc #(`paddr,  `respwidth) prince_read[
   my_blocksize];

    for(Integer i=0;i<my_blocksize;i=i+1) begin
      prince_read[i]<-mkmodular_prince;
    end
`endif

`ifdef MEM_ENC_1
    //rule to send data to be written for encryption to Prince module
    rule rl_encr_put ;
      let x = ff_write_encr_request.first;
```

```
22        ff_write_encr_request.deq;
23        prince.encrypt_block_d(x,'h084c2a6e195d3b7f0000000000000000');
24      endrule
25
26      //rule to receive data to be written to next level memory after
      encryption
27      rule rl_encr_get ;
28        let x <- prince.get_encrypt_block_d();
29        ff_write_mem_request.enq(x);
30      endrule
31  `endif
```

This is a snippet of the dcache1rw module in SHAKTI.

Here the parameter MEM_ENC_1 tells us which changes are specific to Prince and memory encryption. As can be seen, among other instantiations, we make one instantiation of Prince for encrypting purpose and 12 for decrypting. This is because writes can be issued to the memory with delay but read requests must be serviced as quickly as possible.

Upon searching for the same parameter, other such changes made can also be found which were summarized in the diagram in the section.

Lastly, similar changes were also made to the icache module in SHAKTI.

A.2 Encryption script: ELF

This section explains the script used to encrypt the hello world ELF.

```
1  void encrypt(std::string filename) {
2    create_elf();
3
4    ELFIO::note_section_accessor *note_reader = create_note_section();
5
6    for(auto *section : writer.sections) {
7      size_t new_size = ((section->get_size() + req_align - 1) /
       req_align) * req_align;
8      uint8_t *new_data = copy_data((const uint8_t *)section->get_data
       (), new_size);
```

```
9
10      if(section->get_index() != writer.sections.size() - 1) {
11        ELFIO::Elf_Word type, descSize;
12        std::string name;
13        uint8_t *desc;
14
15        note_reader->get_note(section->get_index(), type, name, (void
      *&)desc, descSize);
16
17        uint64_t key_high = *(uint64_t *)&desc[0];
18        uint64_t key_low = *(uint64_t *)&desc[8];
19
20        encrypt_data(new_data, new_size, key_high, key_low);
21      }
22
23      section->set_data((const char *)new_data, new_size);
24    }
25
26    writer.save(filename);
27  }
28
29  void encrypt_data(uint8_t *data, size_t size, uint64_t key_high,
      uint64_t key_low) {
30    Prince prince_side = Prince_new(key_high, key_low);
31
32    uint64_t message, ciphertext;
33
34    for(size_t i = 0; i < size; i += 8) {
35      memcpy(&message, &data[i], 8);
36      ciphertext = prince_encrypt(&prince_side, message);
37      ciphertext = prince_decrypt(&prince_main, ciphertext);
38      memcpy(&data[i], &ciphertext, 8);
39    }
40  }
```

Some important points to note here are:

- The encryption is being performed section wise in the encrypt_data which is called by encrypt.
- The function encrypt first creates a new ELF almost identical to the old one in terms of number and size of sections and adds a note section to it.

- It then proceeds to read sections from the old ELF, encrypt them, and store them in the new ELF with a different size that is byte aligned.
- In the function encrypt_data, the section itself is encrypted in blocks of 8 bytes each.

A.3 Encryption script: code.mem

This section explains the script used to encrypt the code.mem which is generated from the aligned ELF.

```
void encrypt(std::string filename) {
  uint64_t key_high = 0x1234567890abcdef;
  uint64_t key_low = 0xabcdef1234567890;
  encrypt_data(key_high, key_low, filename);
}

void encrypt_data(uint64_t key_high, uint64_t key_low, std::string
    filename) {

  std::string line;
  int line_count = 0;

  Prince prince_side = Prince_new(key_high, key_low);

  uint64_t message, ciphertext;
  std::ofstream fout;
  fout.open(filename);

  while(fin){
    if(line_count > 4194303)
      break;
    getline(fin, line);
    message = std::stoull(line, nullptr, 16);
    ciphertext = prince_encrypt(&prince_side, message);
    ciphertext = prince_decrypt(&prince_main, ciphertext);
    std::string writeback = std::to_string(ciphertext);
    fout << writeback << std::endl;
    line_count = line_count + 1;
  }
```

```
29 }
```

Some important points to note here are:

- The function encrypt just initializes a couple of keys and passes them on to the function encrypt_data. As has been mentioned before, the keys are hard coded for now.
- The function encrypt_data first does some file handling and initialization. Then it starts reading from the code.mem.
- Each line in the code.mem has 8 bytes. Hence, we read, encrypt and write back one line at a time.
- We stop once a particular number of lines are done since that is the total number of lines in the code.mem.

APPENDIX B

PROCEDURES

Here I will elaborate on the procedures to run various files and simulations.

B.1 Running files on SHAKTI

The setup of SHAKTI processor requires the user to follow all the steps given here up until but not including the "Run Smoke Tests" part.

Another important point to note is that upon generating the boot files, the procedure often generates boot.MSB and boot.LSB of 8200 lines each. This will give an error on running ./out. In this case, the user needs to delete the last few lines of both files to make sure they have only 8193 lines each.

B.2 princeutils.cpp

Under the folder Script_New/casl-encryptedmemory-d61d61be308d/20b_shivammittal are the files used to encrypt the ELF. The readme.md presents the procedure to compile and run princeutils.cpp which is the main script used to encrypt the ELF.

Most other files names princeutils_<some name>.cpp are modifications of the same and can be compiled the same way as this one. For example princeutils_codemem.cpp is used to encrypt a code.mem file which is required in one of the testing procedures.

B.3 Running SHAKTI on GDB

Here I will elaborate how to run an ELF on SHAKTI directly using GDB without having to convert it to a code.mem file.

B.3.1 Setup

The setup for debugging support is done as given below:

1. Open c-class/sample_config/default.yaml.
2. set openocd and debugger_support to true.
3. Run "make link_verilator_gdb" after "make link_verilator" while compiling SHAKTI.

B.3.2 Running GDB

Once this setup is done, the procedure to run it is as follows:

1. Open a terminal in the c-class/bin folder and run "./out". It will start listening for openocd to start.
2. Open another terminal in the c-class/test_soc/gdb_setup and run "openocd -f shakti_ocd.cfg". The first terminal detects this and starts running.
3. Open the folder with the ELF that needs to be run and run "riscv64-unknown-elf-gdb".
4. When the prompt comes, run "set remotetimeout unlimited" and then "target remote :3333".
5. At the next prompt, run "file <ELF name>". Then run "load".
6. Once this is done, GDB will have loaded the ELF and can be used.

To run a code.mem file directly on SHAKTI using GDB, just follow up until the step 2 given above. Then open a new terminal in c-class/test_soc/gdb_setup and run "riscv64-unknown-elf-gdb -x gdb.script". Then regular GDB commands can be used.

REFERENCES

1. **Gueron, S.** (2016*a*). A Memory Encryption Engine for General Purpose Processors. URL https://ieeexplore.ieee.org/document/7782706.

2. **Gueron, S.** (2016*b*). A Memory Encryption Engine Suitable for General Purpose Processors. Cryptology ePrint Archive, Paper 2016/204. URL https://eprint.iacr.org/2016/204. https://eprint.iacr.org/2016/204.

3. **Lee, D.**, **D. Kohlbrenner**, **S. Shinde**, **K. Asanović**, and **D. Song** (2017). Keystone: An Open Framework for Architecting Trusted Execution Environments. URL https://n.ethz.ch/~sshivaji/publications/keystone_eurosys20.pdf.

4. **Yin, T.**, **G. Xin**, and **J. Han** (2020). HPME: A High-Performance Hardware Memory Encryption Engine Based on RISC-V TEE. URL https://ieeexplore.ieee.org/document/9278286.