Security Analysis on the PPMLAC: High Performance Chipset Architecture for Secure Multi-Party Computation

Choka Thenappan
MS Computer Engineering
ct3185

Yaagna K. Modi
MS Electrical Engineering
ykm2110

ABSTRACT

Secure multi-party computation (also known as secure computation, multi-party computation (MPC) or privacy-preserving computation) is a subfield of cryptography with the goal of creating methods for parties to jointly compute a function over their inputs while keeping those inputs private. In this project, we aim to understand and analyse the PPMLAC: High Performance Chipset Architecture for Secure Multi-Party Computation [2] for security vulnerabilities. PPMLAC unlike other MPC schemes use techniques to reduce network traffic, hence resulting in higher throughput. We aim to do the analysis by using the OpenTitan Framework to simulate an System on Chip as proposed in the paper [2]. The project hopes to find and analyse issues related to vulnerabilities caused by the functions executed and the synchronization of the CSPRNG, that will leak information.

1 INTRODUCTION

PPMLAC stands for Privacy-Preserving Machine Learning Acceleration Chipset. It's a cutting-edge hardware architecture designed to accelerate secure multi-party computation (MPC) [2]. MPC allows multiple parties to compute jointly on their private data without revealing it to each other. PPMLAC combines strong security with high performance, making it ideal for privacy-preserving machine learning applications. The architecture leverages trusted hardware components to efficiently handle tasks like generating random numbers, masking data, and performing secure multiplications. PPMLAC achieves remarkable speedup over existing software-based MPC solutions, enabling near real-time execution of large-scale ML models while maintaining robust security guarantees.

The project report is organized as follows: initially, we provide an overview of PPMLAC [2] and LARCH [1]. Then, we discuss the possible cloud architecture where these kinds of systems can be used. Since the paper [2] requires new ISA modifications to the system, the report then talks about the ISA Simulation Model we built. Subsequently, we discuss possible vulnerabilities we think are present in the simulation model. Following this, we delve into the OpenTitan framework, including its setup, features, and simulation environment. Finally, we conclude the report with the challenges faced and a conclusion.

2 PPMLAC: AN OVERVIEW

PPMLAC is a novel chipset architecture that efficiently runs secret-sharing based MPC protocols. The paper addresses privacy issues in data sharing and collaborations, proposing a solution that combines the security of MPC with the performance of hardware acceleration.

- **Architecture**: The paper addresses privacy issues in data sharing and collaborations, proposing a solution that combines the security of MPC with the performance of hardware acceleration. The proposed architecture aims to eliminate communications bottlenecks in MPC to achieve significant speed improvement over software-based techniques.
- **Security**: The architecture is designed with minimum number of hardware components to reduce attack surfaces and this in turn makes the architecture robust against the side-channel attacks and malicious adversaries.

2.1 Protocol 1

The Protocol 1 gives the information to perform secure multiplications on two secret-shared numbers between two parties. As shown in the example from figure:1 Alice and Bob wants to perform multiplication operation on two numbers 6 and 8.

- **Precondition**: The CSPRNGs (Cryptographically Secure Pseudo-Random Number Generators) of both parties are synchronized as shown in figure:1(a), and the secret shares create a valid value.
- **Multiplication Process**: The protocol involves one party (Alice) sending encrypted versions of her secret shares to the other party (Bob). Figure:1(c),(d), who then performs local computations to obtain his part of the product. This results in one-way communication that decreases the channel usage. The paper uses concepts related Beaver Triple and one-time padding to perform secure multiplication. Though initially we felt it is counter-intuitive to use one-time pad which are rarely used cryptographic schemes due to its long key size, we felt the use of synchronized CSPRNG seed to be a great technique.
- **Security**: The protocol ensures that neither party learns anything about the other's secret shares, as random numbers act as one-time pads to mask the information.

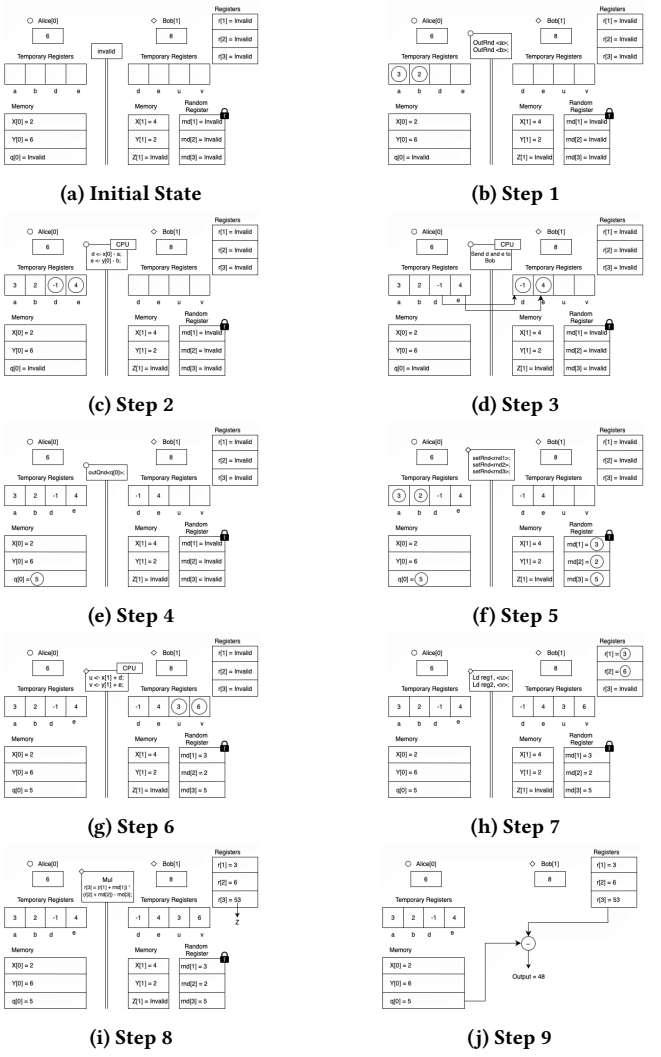

Figure 1: Protocol 1

- **Postcondition**: The resulting secret shares from Alice and Bob form a valid product from the original values, and the CSPRNGs of both parties remain synchronized.

2.2 Protocol 2

The Protocol 2 allows the secure initialization of the CSPRNGs on both parties. The sequence of steps are shown in figure: 2.

- **Precondition**: Alice has her public key inside her chip and a hidden key inside her trusted chip. Bob also has a public key in his chip and a hidden key inside his trusted chip. Both parties have some asymmetric encryption keys stored in their memory as shown in figure: 2(a).
- **Procedure**: Bob sends his public key to Alice and then Alice verifies it figure: 2(c),(d). Alice then generates a random number from her TRNG: m, and concatenates it with her public key. This data is encrypted uses Bob's public key received earlier by Alice. This cipher text is then sent to Bob, who decrypts it with his hidden key to get the value m figure: 2(e). Bob then generates an other random number from his TRNG TR and adds it to m giving $S = m + TR$. This S is encrypted with Alice's public key and sent to Alice over a network figure: 2(f). This network need not be secure for PPMLAC to be secure. Alice now decrypts the data with her hidden key and uses S to initialise her CSPRNG. As the integrity of the PPMLAC algorithm depends on the synchronization of the two CSPRNGs, this step is an important part of the algorithm.
- **Postcondition**: The CSPRGs of both Alice and Bob are initialized with the same seed as shown in figure 2(g).

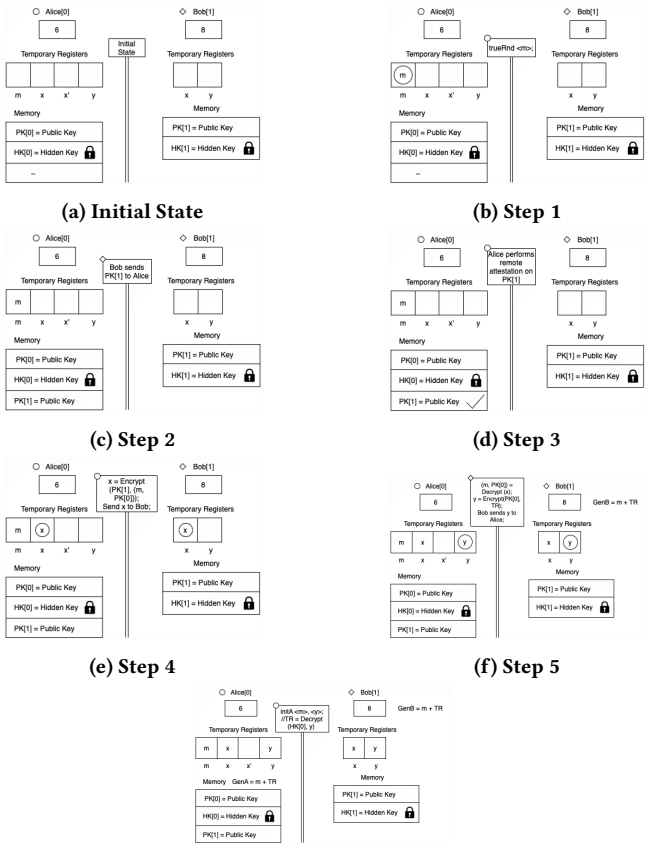

Figure 2: Protocol 2

2.3 Security Guarantees

The paper discusses the following security guarantees for the PPMLAC architecture:

- **Malicious Adversaries**: The system is secure even if a participant deviates from the protocol. For example, if Bob tries to manipulate the computation, the protocol ensures that Alice's data remains secure.

- **Side-Channel Attacks**: The design is resistant to side-channel attacks, which could leak information through indirect means like power consumption. For instance, even if an attacker measures the power usage of the chip during computation, they won't be able to deduce information as it is masked.
- **Forward Security**: Conversations remain secure even if future key leaks occur. Imagine if Bob's chip is compromised in the future, the design ensures that past communications cannot be decrypted, even with the leaked key.
- **Man-in-the-Middle Attack**: The protocol prevents an attacker from intercepting and altering the communication between parties. If an attacker tries to intercept Alice's messages to Bob, the system's design would prevent any tampering or leakage of information as the communication is encrypted.

These guarantees ensure that the architecture provides security for executing computations, protecting against various potential attacks and ensuring the confidentiality and integrity of the computation.

3 LARCH

The authentication systems are crucial for verifying the identity of users accessing online services. Traditional authentication methods requires users to share sensitive information (i.e. passwords) with each service, raising privacy concerns. Also, tracking users across different services can compromise their privacy.

LARCH [1] is a significant paper in the realm of authentication systems. It introduces a novel approach to authentication that emphasizes both privacy protection and accountability.

The paper addresses these challenges by ensuring user privacy is maintained during the authentication process and allowing user to authenticate across multiple services without needing to share sensitive information with each service.

For this LARCH presents an accountable authentication framework designed to enhance security and privacy during user authentication. It introduces a log service that records every authentication attempt, providing users with a comprehensive history of their authentication activities.The framework ensures that even if user's device is compromised, the attacker cannot authenticate without leaving evidence in the log. Furthermore, this framework maintains user privacy by preventing the log server from learning which web service the user is authenticating to.

4 CLOUD ARCHITECTURE

The PPMLAC architecture can be used in cloud setting for various privacy preserving multi-party computations like financial analysis, healthcare applications, etc.

In healthcare, PPMLAC architecture offers a robust solution for maintaining the confidentiality of sensitive medical data while enabling collaboration. For example, in medical research, multiple institutions can securely share data and jointly train machine learning models without disclosing their private patient information. Another crucial application is in drug development where pharmaceutical companies, research organizations and healthcare providers can utilize PPMLAC to analyze vast datasets of clinical trials and results. Through MPC, these institutions can jointly analyze data while preserving their confidentiality. Thus, this enables collaborative research without the risk of data exposure.

5 INSTRUCTION-SET ARCHITECTURE MODELLING IN C++

The PPMLAC defines 5 ISA that are to be included in the trusted chip architecture. These ISAs are used to perform the Secure multiplication. The ISA instructions as follows:

- **ld** $reg, addr$: Load a word from memory to operand register
- **st** $reg, addr$: Store a word from operand register to memory.
- **setRnd** rnd: Set random number to random register rnd.
- **outRnd** $memAddr$: Set random number from the CSPRNG to the memory address. (Stream R)
- **outQnd** $memAddr$: Set random number from the CSPRNG to the memory address. (Stream Q)
- **mul** $reg_1, rnd_1, reg_2, rnd_2, reg_3$: Set reg_3 to $(reg_1 + rnd_1) * (reg_2 + rnd_2) - q_k$

As the system requires us to include new ISA to an already existing architecture, we felt it would be a good idea to make a ISA model of the system. The GitHub Repository contains our working ISA model for PPMLAC.

In this section we will talk about the micro-architecture of both the Bob and Alice's trusted chip and each instruction apart from the above mentioned instructions.

5.1 Trusted Chip architecture of the Master

In the *master.hpp* file the class master defines a chip of an master-Alice. Here, elements like $gcd\ function\ :\ method\ of\ the\ class$ and Hidden Key are private variables. And the Hidden Key can only be accessed by calling the *RSA_decrypt* Instruction. As seen in figure 3,the $Hidden\_Key$ register is architecturally isolated and the user in the real world can't get access to the $Hidden\_Key$.

For simulation purposes the network is simulated through an universal variable called the $network$. When the master wants to communicate to the slave the instruction $send\_data(a, b)$ moves the data from the $temprorary\_register[a]$ to the network. This data is then read from the network variable using the slave $read(a, b)$ instruction, where data is read into $temprorary\_register[a]$ of the slave.

In the paper, PPMLAC they assume apart from the initial stated 5 instruction, rest in executed in the normal domain. From what we understand, a Trusted Chip is an architecturally isolated chip that is also capable of executing standard chip ISA, hence in the simulation model we haven't isolated the execution environments.

Few other preconditions like fixed public and private keys for RSA were assumed.

5.2 Trusted Chip architecture of the Slave

In the *slave.hpp* file the class slave defines a chip of a slave-Bob. We anticipate more vulnerabilities in this side of the system as most of the computing happens in Bob's end.

As seen in figure 4 the slave side architecture includes the same set of registers and memory as the master, however there are additional registers: *architecure registers* and *random registers*. These registers play an important part in the one-padding and secure multiplication in the PPMLAC protocol stated.

5.3 Running the Final Assembly

Assembly program of the two protocol stated in the [2] is given below. The two protocols in the paper, namely Protocol 1, which states the rules of the multiplication and Protocol 2 states the rules for secure key initialization.

```
int main(){
    master alice;
    slave bob;

    //Initializing the network
    //Protocol 2
    alice.outQnd(0);
    bob.send_key(0);
    alice.ld(1,0);
    alice.RSA_encrypt(0,0,1);
    alice.RSA_encrypt(1,1,1);
    alice.send_data(0,0);
    alice.send_data(1,1);

    bob.read(0,0);
    bob.read(1,1);
    bob.RSA_decrypt(0,2);
    bob.RSA_decrypt(1,3);
    bob.TRNG(0);
    bob.add(2,0);
    bob.store_temp(0,2);
    bob.store_temp(1,3);
    bob.RSA_encrypt(0,0,1);
    bob.send_info(0,0);

    alice.readtemp(0,0);
    alice.RSA_decrypt(0,1);
    alice.intialize_seed(1);

    //Intial Conditions
    alice.mem[0] = 3;
    alice.mem[1] = 6;
    bob.mem[0] = 3;
    bob.mem[1] = 2;

    //Multiplication
    //Protocol 1
    alice.intialize_seed(1);
    alice.outRnd(0);
    alice.outRnd(1);
    alice.subract(2,0,0);
    alice.subract(3,1,1);
    alice.send_data(2,0);
    alice.send_data(3,1);
    alice.outQnd(2);
    //Bob
    bob.intialize_seed(2);
    bob.setRnd(0);
    bob.setRnd(1);
    bob.setRnd(2);
    bob.read(0,0);
    bob.read(1,1);
    bob.add(2, 0, 0);
    bob.add(3, 1, 1);
    bob.load(0, 2);
    bob.load(1,3);
    bob.mult();
    bob.store(2,2);

    return 0;}
```

Apart from instructions defined initially in this section we include few other instructions in the model for both Master and Slave.

Slave Instructions

- **send_key** b : This Instruction sends the public key of the slave to the network.
- **read** a, b : This instruction reads data from $network[b]$ to $temprory\_register[a]$.
- **TRNG** a : This instruction generates a true random number using the TRNG to the $temprory\_register[a]$.
- **add** a, b : This instruction adds values at $temproryregister[a] = temprory\_register[a] + temprory\_register[b]$.
- **initialize_seed** a : Initializes the CSPRNG with seed as the value in $temproryregister[a]$
- **store_temp** a, b : Stores $memory[a] = temprory\_register[b]$.
- **send_info** a, b : Send data in $temporary\_register[a]$ to the network buffer $network[b]$. It is important to note that this ISA can only be called during protocol 2.
- **RSA_encrypt** a, b, c : Here $mem[a]$ is the message, and $mem[c]$ is the public key of the master and the encrypted cipher text is stored in $(temprory\_register[b]$.
- **RSA_decrypt** a, b : Here $temprory\_register[a]$ is the cipher texted, which is decryted with the slave's hidden key and then is stored to the $temprory\_register[b]$.

Master Instructions

- **subract** a, b, c : Performs $temporary\_register[a] = memory[b] - temporary\_register[c]$
- **send_data** a, b : Send data in $temporary\_register[a]$ to the network buffer $network[b]$.
- **readtemp** a, b : Moves data to $temporary\_register[a]$ from the network.

- **initialize_seed** a : Initializes the CSPRNG with seed as the value in $temprory\_register[a]$
- **RSA_encrypt** a, b, c : Here $mem[a]$ is the message, and $mem[c]$ is the public key of the master and the encrypted cipher text is stored in ($temprory\_register[b]$.
- **RSA_decrypt** a, b : Here $temprory\_register[a]$ is the cipher texted, which is decrypted with the slave's hidden key and then is stored to the $temprory/_{r}egister[b]$.

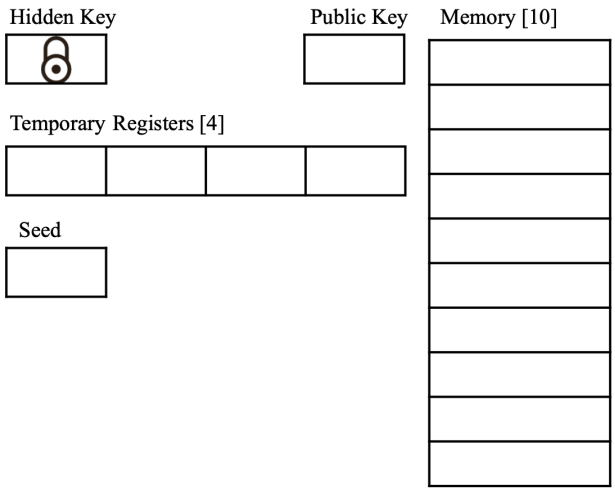

Figure 3: Register Architectur of the Trusted Chip of Master (Alice)

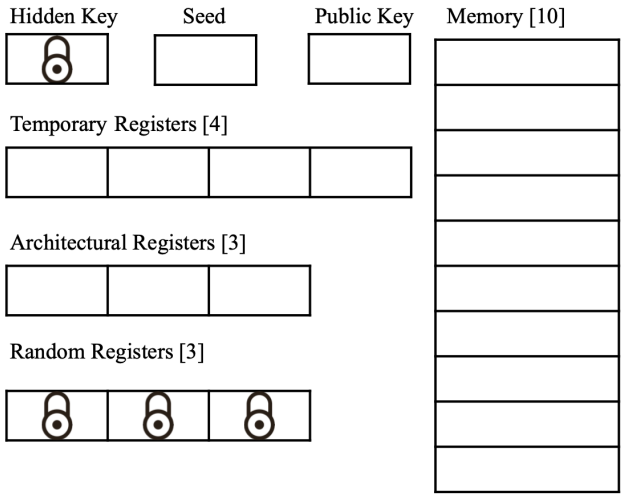

Figure 4: Register Architecture of Trusted Chip of Slave (Bob)

6 POSSIBLE VULNERABILITY

As PPMLAC goes for a protocol which reduces communication overhead, Alice never receives data from Bob except for the final output that is one-time padded. Because of which to prove the system is secure, it is sufficient to prove that Bob doesn't learn anything about Alice's data.

Possible vulnerabilities analysed from the simulation model:

- **Issue with the Function being computed** : Let us take a system where there are 3 Alice, $Alice_1$, $Alice_2$ and $Alice_3$. They together want to calculate a function λ in Bob's chip. These kinds of scenarios are common when Machine Learning Models are trained in a cloud servers. In a scenario where Bob along with $Alice_2$ and $Alice_3$ join together to steal the plain text supplied from $Alice_1$. Let us say the function λ is a multiplication function, $Alice_2$ and $Alice_3$ can use plain texts such as 1 that will leak the plan text value of $Alice_1$. Security should be guaranteed in a MPC system even if n-1 parties are dishonest in a n party system. This can be illustrated from figure 5. Here, initially if Bob gives a value of 6, the answer is 36. With this information Bob can't predict Alice's share, as he has no idea about the number of parties in the system. However, if he knows the number of parties in the system and all the parties apart from Alice give a share of *unity*, the system becomes vulnerable as the output will be Alice's share.

```
chokа@Chokas-Air ISA-Model % ./test
Enter the number you want to multiply with Alice's number secretly (You are Bob): 6
Final Answer = 36
chokа@Chokas-Air ISA-Model % ./test
Enter the number you want to multiply with Alice's number secretly (You are Bob): 1
Final Answer = 6
```

Figure 5: Using an constant unity for multiplication can leak information.

- **Security of the seed supplied to the CSPRNG** : We believe the entire security of the proposed MPC system relies on the synchronization of the CSPRNGs on both the master and slave sides, ensuring they operate at the same speed. It's also important to note that if the seed is leaked, the security of the system is compromised. An attacker could induce faults in one side's CSPRNG, causing discrepancies in the behavior of both the master and slave chips. This would lead to incorrect results. While the attacker may not gain access to the data, they can still cause erroneous outputs. This protocol lacks measures to prevent or detect such attacks, which could prove very costly when training machine learning models. As seen in 6, faults in CSPRNG will result in wrong outputs. In the illustrated case, Alice's share is figure 6 and the expected values are 54, 24 and 6. The effectiveness of this attack comes from the fact that PPMLAC [2] has no security measure to make sure the CSPRNG units in both systems are synchronized throughout the process.

7 OPENTITAN

OpenTitan is an open-source hardware Root of Trust (RoT) project aiming to provide a secure foundation for building trustworthy silicon root of trust chips. A root of trust is a critical component in computer systems, establishing a secure foundation for booting, verifying software integrity, and protecting sensitive operations.

```
choka@Chokas-Air ISA-Model % ./test
Enter the number you want to multiply with Alice's number secretly (You are Bob): 9
Final Answer = 6
choka@Chokas-Air ISA-Model % ./test
Enter the number you want to multiply with Alice's number secretly (You are Bob): 4
Final Answer = -4
choka@Chokas-Air ISA-Model % ./test
Enter the number you want to multiply with Alice's number secretly (You are Bob): 1
Final Answer = -10
```

Figure 6: Program output after inserting a fault in the seed of the CSPRNG

- *Key Features*: OpenTitan includes a hardware root of trust, which is the foundation of its security architecture. This root of trust ensures the integrity and authenticity of the device's boot process and critical operations.
- *Subsystems*: OpenTitan is made of Ibex Core which acts as its host CPU, furthermore, it has various subsystems to provide hardware root of trust, secure boot, and cryptographic functions.

7.1 Setup

For this project we used Google Cloud Platform to setup OpenTitan. This provided us with scalable computing resources and easy access to the collaboration tools. To run OpenTitan software, we needed to both build the software components and simulate the hardware they run on. This required setting up Verilator and Vivado. After completing the setup, we successfully passed all tests from the OpenTitan Toolchain, ensuring the correct setup of the simulation environment.

OpenTitan has multiple kinds of test environments. The basic is a *smoke test*, which is used to test the functionality of a particular IP. Then they have a *system test*, which will aid in performing a full system test.

OpenTitan utilizes Bazel to build the software on simulated hardware and then verifies whether the code functions as expected or not. Bazel is a powerful build tool that allows for efficient and reproducible builds, ensuring that software builds correctly on the simulated hardware environment. This process is essential for validating the functionality and integrity of the software within the context of the OpenTitan hardware design.

When running a C test code, the Toolchain first initializes the code in its ROM and then executes it on the simulated hardware. This ensures that the software is properly integrated into the system and functions as expected within the simulated hardware environment provided by OpenTitan.

8 PROPOSED ARCHITECTURE FOR TESTING

To test the PPMLAC architecture using OpenTitan we need to add Alice and Bob's Chip inside the OpenTitan framework and then use software to initialize the protocol.

8.1 Hardware Design

The paper [2] describes the hardware architecture as shown in 8 and it is build using minimum number of units like multiplier, adder,

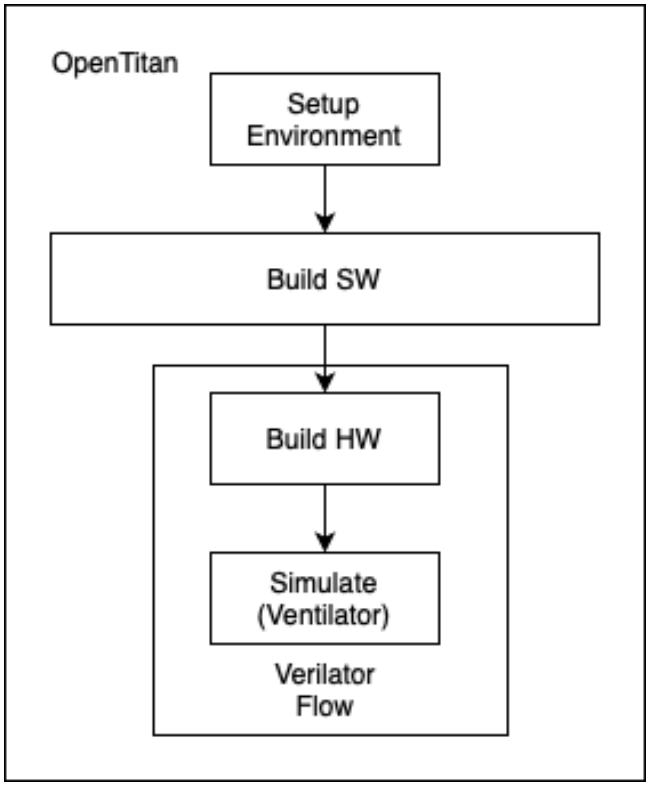

Figure 7: OpenTitan Setup Flow

True Random Number Generator and CSPRG so as to reduce the risk of side channel attacks. After making Bob's side chip we need to interface it with OpenTitan framework. And then use software to simulate the Alice's side. We can use software to simulate the Alice's side because communication is happening one way only. The hardware components are as follows:

- **OTBN**: OpenTitan contains a Big Number Accelerator called OTBN. It is a cryptographic co-processor which can be used for asymmetric operations like RSA. The design of this module is very simple so as to reduce the attack surface for instance, it has no interrupts and all computations are done in single cycle.
- **CSPRNG**: This is a Cryptographically Secure Random Number Generator provided by OpenTitan. It is designed with robust cryptographic principles to resist predictability and ensure randomness. This module provides a reliable source of randomness for cryptographic operations like true random number generation as well as deterministic random number generation.
- **Key**: Bob's chip contains two keys one is a public key and other one is his private key.
- **Secret Multiplier and ADDer**: This is a multiplier and ADDer whose inputs cannot be accessed from outside thus making them a secret multiplier and ADDer.

- **Random Register**: A register for storing random values generated by the CSPRNG. This register cannot be accessed from off-chip as well.

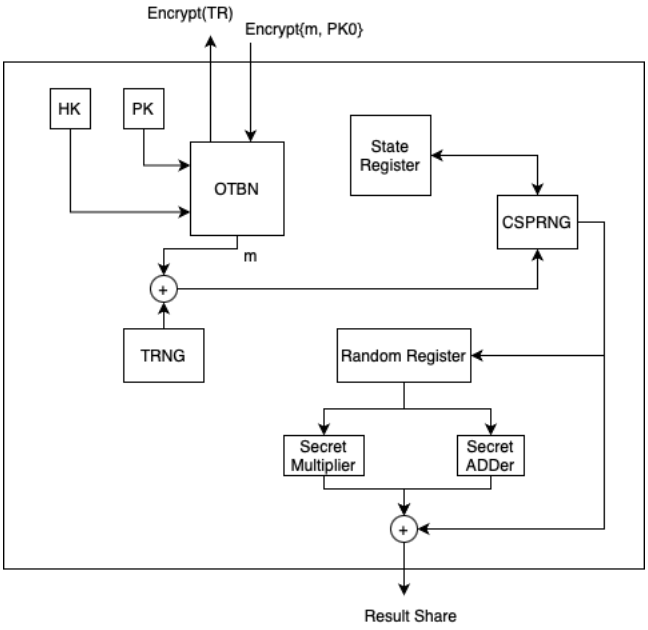

Figure 8: Bob's Side Architecture

8.2 Software Design

The software is used to initialize the secure chips and pass instructions for executing the PPMLAC protocol. It configures the hardware components of Bob's chip, establishes secure communication channels, and initiates the protocol. This involves setting up cryptographic parameters, initializing the TRNG and CSPRNG, and configuring the arithmetic units. Additionally, the software simulates the flow of data between Alice and Bob's Chips, ensuring that computations are performed in a functional correct manner.

9 RUNNING CODE ON OPENTITAN

To run a C code on the OpenTitan Toolchain, the code needed to be first loaded onto the ROM. While various smoketests were provided with OpenTitan (smoketests are programs which are used to verify functionality of IP blocks), running the example C code within the same framework was not straightforward. We had to thoroughly understand the BUILD files of the smoketests and then apply the necessary modifications to the BUILD files of the example code. This process was time-consuming, and making changes to the BUILD files was not straightforward. Significant amount of time was spent in deciphering the structure and dependencies of the existing BUILD files. Once we had a clear understanding, we edited the BUILD files of the example code to align with the framework used by the smoketests. This involved adjusting configurations, specifying correct dependencies, and ensuring compatibility with the toolchain's build process. After several iterations and testing phases, we successfully modified the BUILD files, enabling the example C code to run seamlessly within the OpenTitan framework. The code changes to BUILD file are as follows:

```
load("//rules/opentitan:defs.bzl", "opentitan_binary",
load("//rules:files.bzl", "exclude_files")
load("@rules_pkg//pkg:mappings.bzl", "pkg_files")

package(default_visibility = ["//visibility:public"])

opentitan_binary(
    name = "hello_world",
    testonly = True,
    srcs = ["hello_world.c"],
    copts = [
        "-Wno-date-time",
    ],
    exec_env = [
   "//hw/top_earlgrey:fpga_cw310",
   "//hw/top_earlgrey:sim_dv",
   "//hw/top_earlgrey:sim_verilator",
    ],
    deps = [
        ":hello_world_lib",
        "//sw/device/lib/base:mmio",
    ],
)

opentitan_test(
name = "hello_world_test",
verilator = verilator_params(
binaries = {
":hello_world": "firmware",
},
),
exec_env = {
"//hw/top_earlgrey:sim_verilator": None,
}
)

cc_library(
    name = "hello_world_lib",
    deps = [
    "//hw/top_earlgrey/sw/autogen:top_earlgrey",
    "//sw/device/examples:demos",
    "//sw/device/lib/arch:device",
    "//sw/device/lib/crt",
    "//sw/device/lib/dif:gpio",
    "//sw/device/lib/dif:pinmux",
    "//sw/device/lib/dif:uart",
    "//sw/device/lib/runtime:hart",
    "//sw/device/lib/runtime:log",
```

```
    "//sw/device/lib/runtime:print",
    "//sw/device/lib/testing:pinmux_testutils",
    "//sw/device/lib/testing/test_framework:
check",
    "//sw/device/lib/testing/test_framework:
ottf_ld_silicon_creator_slot_a",
    "//sw/device/lib/testing/test_framework:
ottf_start",
    "//sw/device/lib/testing/test_framework:
ottf_test_config",
    ],
)

pkg_files(
    name = "package",
    testonly = True,
    srcs = [":hello_world"],
    prefix = "earlgrey/examples",
    )
```

The command to run C codes on Opentitan Framework is ./bazelisk.sh run //sw/device/examples/hello_world:hello_world_test_sim_verilator

The figures 9 and 10 shows the successful run of the C code of Hello World and USB Dev code from OpenTitan example.

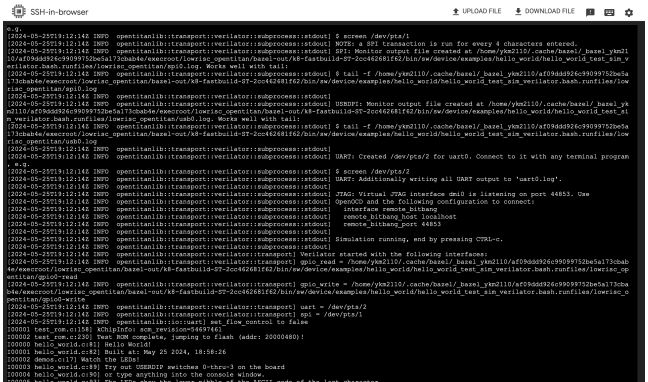

Figure 9: Hello World on OpenTitan

10 CHALLENGES FACED

During our project, we faced several challenges while setting up OpenTitan on Google Cloud Platform (GCP). Our lack of experience with toolkits like OpenTitan made the initial setup and usage particularly challenging. The outdated reference manual for OpenTitan required us to make manual changes to the JSON and top_level generation files to successfully generate the toplevel_earlgrey. While the initial steps, including setting up Verilator, were relatively straightforward, configuring Vivado proved to be more complex.

Our major challenge arose when attempting to run a C model on OpenTitan. In OpenTitan, the C code must be loaded into the ROM before running it on the simulated hardware, but the documentation lacks guidance on how to load the ROM. To address this, we had

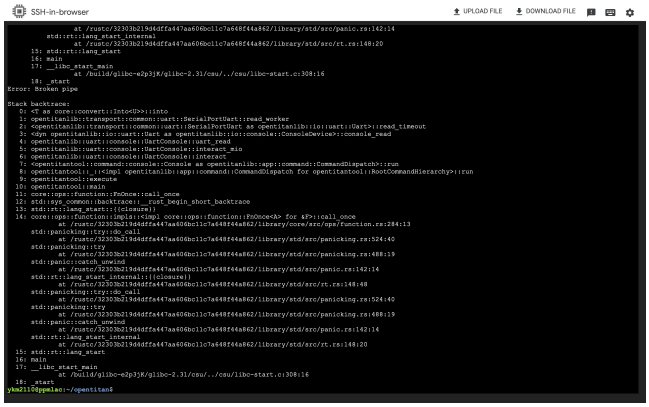

Figure 10: USB Dev

to make changes to the BUILD file, and through trial and error, we were able to run C code on the OpenTitan framework.

After running the C code on OpenTitan, we attempted to run our ISA model on the toolchain. Through exhaustive testing, we discovered that to run custom C code on the toolchain, the code must model hardware-level transactions. For instance, if the OBTN module is invoked, this must be explicitly specified in the code.

11 CONCLUSION

During this project we understood the PPMLAC architecture and developed a software program to simulate the architecture. To test our program, we set up the OpenTitan framework on GCP and verified that all the IPs of OpenTitan were running as expected through various smoke tests. Additionally, we successfully loaded C code into the ROM and tested it on the toolchain, which required modifying several BUILD files within OpenTitan.

We also attempted to run our ISA model on OpenTitan. Through testing, we discovered that running custom C code necessitates making changes to our model to accurately invoke hardware-level transactions in the code. This step is essential for the code to function correctly on the toolchain.

REFERENCES

[1] Emma Dauterman, Danny Lin, Henry Corrigan-Gibbs, and David Mazières. 2023. Accountable authentication with privacy protection: The Larch system for universal login. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. USENIX Association, Boston, MA, 81–98. https://www.usenix.org/conference/osdi23/presentation/dauterman

[2] Xing Zhou, Zhilei Xu, Cong Wang, and Mingyu Gao. 2022. PPMLAC: high performance chipset architecture for secure multi-party computation. In *Proceedings of the 49th Annual International Symposium on Computer Architecture* (New York, New York) *(ISCA '22)*. Association for Computing Machinery, New York, NY, USA, 87–101. https://doi.org/10.1145/3470496.3527392