Alcatraz: Secure Remote Computation via Sequestered Encryption in Minimally Trusted Hardware

Albert Lu[1,2]

[1] Phillips Exeter Academy, Exeter, NH, USA

[2] MIT PRIMES, Cambridge, MA, USA

Instructors: Sacha Servan-Schreiber (MIT)*, Jules Drean (MIT)*

* Current affiliation: Tinfoil, Inc. (https://tinfoil.sh)

Alcatraz: Secure Remote Computation via Sequestered Encryption in Minimally Trusted Hardware

Albert Lu[1,2]

[1]MIT PRIMES
[2]Phillips Exeter Academy

Abstract

This paper introduces "Alcatraz," a new architecture that enables secure remote computation with minimal trust in the hardware. In Alcatraz, sensitive data is always encrypted, except when it is inside a small, trusted circuit, which is composed of an Arithmetic Logic Unit (ALU) gated by a decryption and encryption engine. By design, the internal states of the trusted circuit is inaccessible from any software, and unencrypted data is never exposed outside the trusted circuit. Thus it is extremely difficult for any attacker to gain information about the sensitive data by observing or attacking other parts of the processor and computer, such as registers or caches, or by exploiting any microarchitectural side channels.

We implemented Alcatraz on a field-programmable gate array (FPGA), and verified with a formal proof that the circuit is secure at the wire-level, which is stronger than the register-transfer-level (RTL) security proved previously. Wire-level verification has the benefit that it's much closer to the physical reality, i.e., the timing and level of signals on the wire, that may be observed by attackers.

We apply Alcatraz to single-server private information retrieval and estimate based on benchmark that Alcatraz achieves 7× to 21× speedup when compared with the current state-of-the-art approach for private information retrieval. Our approach also reduces the communication size by five orders of magnitude.

1 Introduction

With the rapid development of cloud computing, increasingly large amounts of information is being exchanged and processed on remote computers. However, this trend has also brought about its own risks. Because cloud computing is a shared resource, processing sensitive information in the cloud means that it may be stolen by malicious attackers [6]. Any vulnerability in the hardware or software can undermine the confidentiality of sensitive data.

Microarchitectural side channels are routes via which sensitive information may leak. These "channels" exist because the hardware states below the instruction-set architecture may become data dependent (e.g., timing differences due to cache miss). Attackers can exploit these channels to learn about sensitive information they otherwise do not have access to, or potentially obtain the secrets outright. Microarchitectural side channel attacks pose threats to all computing environments, and have been shown to compromise RSA private keys [5]. There are even more attacks that take advantage of side-channels inside the processor itself, such as cross-privilege data gathering [50] and speculative data gathering [39].

A specific but common family of microarchitectural side channel attacks exploits timing differences in runtime. Numerous timing-based attacks are able to exploit these minute differences to infer secret data [57, 34]. These techniques can also be combined with other attacking techniques, such as transient execution [29].

One approach to address these hardware vulnerabilities is to reduce the attack surface and keep sensitive data within a small, trusted chip (a Trusted Execution Environment; **Figure 1, Left**). This technique has been applied to cryptography, such as the use of TPMs (Trusted Platform Modules) [46]. Although Trusted Execution Environments offer improvement in security by isolating sensitive operations within a secure enclave, they still require trusting a software and hardware stack, which may be susceptible to exploitation. It has been shown that TPMs still suffer from timing and lattice side-channel attacks [40].

Another approach to address these hardware vulnerabilities is Fully Homomorphic Encryption (FHE) [22]. Fully Homomorphic Encryption allows a computer to do direct computation on the ciphertext without needing to decrypt it. As a result, any potential leakage would still be encrypted, and thus not a problem. However, FHE is too slow to be practical [1] (**Table 1**).

	FHE	**TEE**	**Alcatraz**
Security	Cryptographic	Full hardware trust Vulnerable to side channels	Minimal hardware trust
Efficiency	Slow	Fast	Fast
Expressivity	Can only compute logical circuits	Can run arbitrary programs	Can only compute logical circuits

Table 1: Comparison of approaches to secure remote computation.

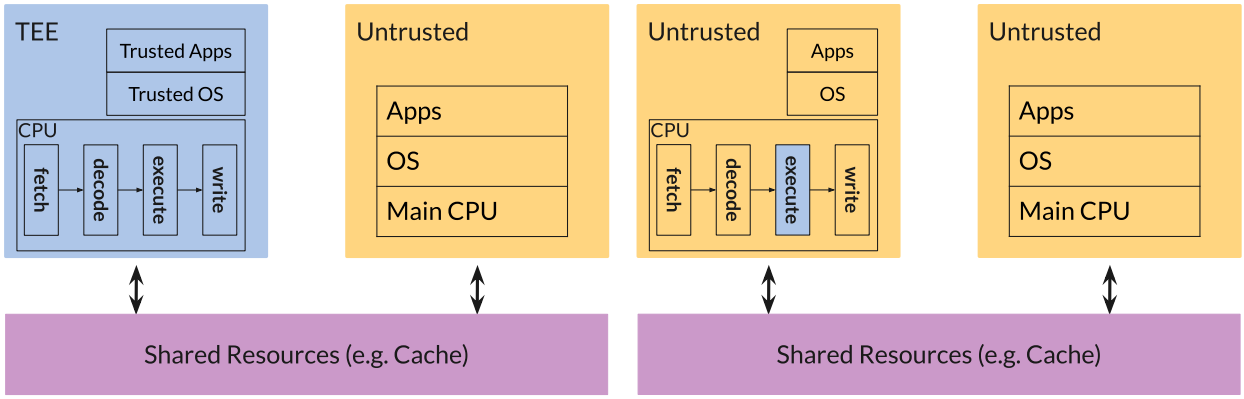

Figure 1: Left: A Trusted Execution Environment (TEE) requires a trusted software and hardware stack. Data residing in the untrusted hardware components (shown in orange) are encrypted. They may become decrypted in the trusted hardware components (shown in blue). **Right**: In contrast, our approach only requires trusting a small circuit within the CPU execution pipeline stage. As a result, our approach has much smaller attack surface than TEE.

In this work, we develop Alcatraz, which is inspired by both Trusted Execution Environments and FHE, as a solution to secure remote computation. Alcatraz is a new architecture, where we address microarchitectural side channel attacks by restricting all non-encrypted sensitive data to a special Arithmetic Logic Unit (ALU) gated by encryption engines. Any sensitive data outside this ALU is always encrypted (**Figure 1, Right**). This means that as long as the adversary[1] does not have access to the internal states of the ALU, they will not be able to infer any sensitive information.

Furthermore, we provide a formally verified proof that the special ALU is secure against timing-based side channel attacks at the wire level. Our proof rests on the fact that the behavior of the encrypted ALU's input/output wires completely captures any possible timing side-channels at a clock-cycle accurate level.

Using the Knox framework [8], we verify that the behavior of the circuit's input/output wires doesn't leak more information than what is described in the specification, and thus is free of any timing leakage (see Sections 3.3 and 4.4.3 for more details).

Compared to the register-transfer-level proof by Tan et al. [55] for Sequestered Encryption (SE), gate-level abstraction is closer to the circuit's actual physical layout. This creates significantly larger expressions representing the circuit, meaning that significant speedups are required to create a complete formal proof of security (see sections 4.1 and 2.7 for more details).

1.1 Our contributions

We build Alcatraz, a new architecture for secure computation, which is based on sequestered encryption (SE) [12]. The idea of sequestered encryption is to remove all sensitive plaintext values from architectural and microarchitectural states that could be potentially accessed by an attacker.

[1]For details on our threat model, see Section 3.1.

The central piece of our new architecture is a hardware module that can perform arithmetic or logical operations on encrypted data. We provide a full Verilog implementation of the hardware module.

Along with that, we also provide a proof of the hardware's correctness and security properties (i.e., free of timing side channels) at the wire-level. Our proof is based on the concept of information-preserving refinement (IPR) and the Knox framework of Athalye et al. [8]. We crafted efficient proofs for the correctness and security of our implementation.

We integrated Alcatraz with a RISC-V core [21] and a vector coprocessor [47], and applied it to a private information retrieval [14, 30] task. We achieved 7× to 21× speedup against state-of-the-art FHE implementations [38].

2 Background

In this section, we cover some basic background knowledge for this paper.

2.1 Symmetric-key Encryption and Authenticated Key Exchange

In Alcatraz, we use symmetric-key encryption to protect sensitive data. Symmetric-key encryption utilizes the same key for both encryption and decryption, making it much faster than asymmetric-key encryption scheme. The challenge of symmetric-key encryption is that the key needs to be known by both parties. To solve the problem, we look at an authenticated key exchange, which allows two parties to agree on a common key.

The Diffie–Hellman key exchange is a of establishing a shared secret between two parties without revealing it. However, Kara et al. [25] notes that the Diffie–Hellman key exchange is still susceptible to man-in-the-middle (MitM) attacks. To combat these attacks, both parties have to send additional information to confirm that the original information has not been tampered with. This is called authenticated key exchange [20, 31], which requires a public-key digital signature scheme, such as ECDSA [13].

2.2 Field Programmable Gate Arrays (FPGAs)

Field Programmable Gate Arrays, or FPGAs, are configurable integrated circuits that can be re-programmed. They consist of programmable logic blocks connected by a communal grid, meaning they are very versatile. To achieve that versatility, FPGAs sacrifice performance and thus may not be the best choice if a circuit's design is fixed and needs to run many times repeatedly [48]. For Alcatraz, we use FPGAs as a prototyping platform (see Section 4.4).

2.3 Microarchitectural Side Channels

A microarchitectural side channel is a type of side channel where information leaks through a hardware feature in the computer, especially through the micro-architecture. For example, cache timing

attacks take advantage of the significant time difference between accessing a piece of data located in the cache versus located in the main memory. Execution time, power supply/consumption, and timing of memory accesses are all examples of microarchitectural side channels (see also surveys in [54, 35]). The existence of side channels is usually due to the existence of microarchitectural mechanisms that were created for performance optimization (branch predictors, memory hierarchies, etc.)

Side channel attacks Side-channel attacks utilize these microarchitectural side channels to get information. For example, if a program's execution duration depends on the number of 1-bits in the secret key, then an adversary can learn that information through measuring the program's duration. Adversaries can do similar things with power consumption attacks and electromagnetic side channels [53].

These information leakages can give an attacker substantial information which can aid them in recovering sensitive data such as a private key. For example, Karimi et al. [26] show that timing-based side channel attacks can steal encryption keys. In our work, we will mitigate all timing-based side channel attacks.

2.4 Homomorphic Encryption

Homomorphic encryption is an encryption function $E(x)$ with the special property that allows for $E(f(x, y))$ to be computed based on $E(x)$ and $E(y)$, where f is an arbitrary simple function of x and y. E must also follow the normal properties of an encryption function, hiding the message from computationally-bounded adversaries that do not have access to the secret key.

In homomorphic encryption, it is shown that the two fundamental operations, homomorphic addition and homomorphic multiplication, correspond to XOR and AND operations in Boolean logic. Implementing these two operations is sufficient to support fully homomorphic encryption [22].

Fully homomorphic encryption is sometimes considered the holy grail of cryptography, as it could enable countless secure applications. Indeed, computation can be delegated to untrusted parties as sensitive data is never exposed and always encrypted under the private key. As long as the private key is kept safe, the data can never be stolen or extracted.

Feasibility Fully homomorphic encryption suffices in an ideal world. However, in the real world, implementations are often too slow to be practical [42]. As such, many variants, (e.g., partially homomorphic encryption [41]) are developed, which are limited by the amount of multiplications they can perform.

2.5 Formal Verification

When cryptographic schemes are devised, there is usually a mathematical proof that the scheme itself is secure. However, in the process of implementing the scheme into actual code, there may be bugs that introduce side-channels, or even errors in the implementation that could lead to security breaches.

In order to combat this, "formal verification" tools can be used. Formal verification tools make it possible to prove or disprove the correctness of a system's implementation, with respect to a previously-defined specification, using formal methods and automated mathematical reasoning [28]. It proves that the code correctly implement the system according to the specification. Furthermore, it can be extended to show not only correctness but also security, and even resistance against some side-channel attacks, like timing. In our work, we use the Knox framework by Athalye et al. [8] to complete our formal verification.

2.6 Satisfiability Modulo Theories

Satisfiability Modulo Theories (SMT) analyzes the satisfiability of some formulas, given a background theory. This background theory fixes the meaning of predicates and function symbols, such as $<, +, \oplus$. A satisfiability modulo theory solver (e.g., Z3 by Microsoft [19]) contains many specialized solving algorithms for specific background theories, which can be combined to determine the satisfiability of a Boolean formula [10]. When trying to verify a property in formal verification, tools will often show it is equivalent to verifying the satisfiability of an SMT formula. An SMT solver can then be used to verify the property automatically.

2.7 Sequestered Encryption

Biernacki et al. [12] introduces the hardware technique called Sequestered Encryption. They assume an adversary that has complete software access, being able to launch software attacks and compromise the operating system. However, the adversary cannot launch physical attacks, like differential power analysis or the "electron-microscope attack".

This technique isolates the sensitive plaintext data into a small trusted hardware unit, and also ensures no sensitive plaintext values exist in any software-accessible architectural state, significantly reducing the attack surface an adversary can reach. This trusted hardware unit is known as the "SE unit".

Tan et al. [55] provides two implementations. First, they introduce a "stateless atomic implementation", which always sandwiches an operation with a decrypt and encrypt operation. However, the problem with that is that there are many extraneous encrypt-decrypt pairs, which are not necessary and increase the latency of operations. To combat this problem, Tan et al. introduces SE-OPT, which is the stateful optimized implementation. To get rid of the extraneous encrypt-decrypt pairs, they extend the SE unit to include a decryption cache, which stores ciphertext-plaintext pairs. Specifically, when an instruction is given to the SE unit, the ciphertext value is used to index the

decryption cache, which returns the corresponding decrypted plaintext (if it is present in said decryption cache). If the ciphertext is not present, it is decrypted (as usual). This design introduces short-term memory of recent decryption operations. Notice that decryption cache latency is dependent on the whether the ciphertext is present or not, which is not a secret. That also upholds their security goals related to timing-based side channels.

3 Overview

We develop "Alcatraz", an architecture for secure remote computation with minimal trust in the hardware. Within Alcatraz, the basic Arithmetic Logic Unit (ALU) is a simple decrypt-operation-encrypt sandwich. We adopt the Advanced Encryption Standard (AES), a symmetric key encryption algorithm, for both our encryption and decryption.

3.1 Our Threat Model

In this section, we describe and discuss our threat model. We first present our security goals. Then, we list the capabilities that our adversary has. Finally, we describe the root of trust, which specifies the components that we assume to be trusted.

Security Goals In our work, we will formally verify that our Alcatraz implementation is resistant to timing-based side channel attacks (i.e., does not leak sensitive information through timing-based side channels). More specifically,

- It should not leak sensitive data through architectural states and/or timing-based microarchitectural side channels.
- Any secure instruction should not leak sensitive data through microarchitectural states and/or I/O signals.

Attacker Capabilities In this work, our adversary aims to gain information about sensitive information (plaintexts) in polynomial time. We assume that our adversary possesses the following capabilities:

- The attacker can observe and/or change digital signals outside Alcatraz, including signals in the Alcatraz's wire-level I/O.
- The attacker can run any program (including malformed ones) on Alcatraz.
- The attacker can measure and analyze program execution time via precise timers [17].

At the same time, we also assume that our adversary does not have the following capabilities:

- The attacker **cannot** read and/or change the **intermediate** states in Alcatraz (i.e., states within Alcatraz that are not part of the I/O), as well as the physical characteristics of the chip.
- The attacker **cannot** use physical side channels such as electromagnetic radiation [2], temperature [24], or power [37].

Root of Trust Our work assumes that the hardware manufacturer and Alcatraz's ALU can be trusted. This means we assume the initial private key is secure and the circuit is correctly made. Everything else (e.g., software) could potentially be compromised.

3.2 Utilizing Secure Enclaves

The blueprint of Alcatraz is based off secure enclaves like the Trusted Platform Module (TPM). Currently, many solutions using TPMs have been developed to combat microarchitectural side channel attacks. Designated TPM coprocessors are used to store the corresponding private keys.

These TPMs have the basic cryptographic actions such as encryption, signatures, hashing (this is known as lightweight cryptography), as well as a hardware random number generator that is used for key generation. It can also generate a report of its hardware and software, bind information, and "seal" information, giving TPM its attestation abilities.

In return for a wide variety of usage, however, TPMs require a relatively large trusted area. As it turns out, they are still vulnerable to side-channel timing attacks [40] and man-in-the-middle attacks [46, 16].

To enhance security, we adopt the recently developed sequestered encryption (SE) method [55], which has been shown to be more efficient than Fully Homomorphic Encryption (FHE) [44].

The objective of homomorphic encryption is to enable operation on ciphertexts in untrusted environments. Although FHE can directly operate on sensitive data without ever decrypting them, it is not computationally efficient given the current implementations [4]. An FHE add operation can take thousands of clock cycles to complete [3]. In contrast, with Alcatraz, operations are done on plaintexts in a trusted circuit.

3.3 Proving Security

To fully prove wire-level security against our adversary, we utilize a new framework, Knox. Introduced by Athalye et al. [8], it can be used on Hardware Security Modules (HSMs). Given an HSM's circuit and specification, Knox can verify the functional correctness of the circuit, as well as formally verify that it has no timing-based leakage. They consider a powerful adversary that has direct access to the wire-level input/output of the HSM, with the ability to set logic levels on the input wires and read logic levels on the output wires at every clock cycle.

Gate-level abstraction and wire-level security In Knox, we abstract circuits to the gate-level. Compared with the register transfer level (i.e. the level where Hardware Description Languages such as Verilog operates in), the gate level abstraction is closer to a circuit's actual physical layout. Knox is capable of proving an HSM is secure regardless of the wire-level input the HSM receives, hence wire-level security.

Athalye et al. [8, 9] introduces the concept of Information-Preserving Refinement (IPR), which is that a given HSM circuit implements its provided specifications and leaks no further information through wire-level behavior. The goal of IPR is to establish an equivalence between the physical implementation and specification through a driver (for the implementation) and an emulator (for the specification).

To prove IPR, Knox models the specification and implementation as state functions, relates the two state functions with a refinement relationship, and proves three properties of the relationship: functional equivalence, physical equivalence, and initialization.

Physical equivalence and timing side-channel in HSM For a hardware security module, all it exposes to the host computer (and any potential attackers) are its input/output wires. Thus the behavior of the HSM's I/O wires should capture any timing channel at a cycle-accurate level. Athalye et al. [8] show that if the physical equivalence can be established between an HSM's circuit and an emulator (on top of the specification), it implies the circuit doesn't have timing side-channels.

For the authenticated key exchange, we use the already-verified ECDSA signature generator [9] and the authenticated exchange scheme from [45].

We create a specification of our circuit according to the NIST FIPS 197 standard of AES. Utilizing this specification and a Verilog description of our circuit, we can use the Knox framework to formally verify wire-level security.

4 Our Solution: Alcatraz

Here, we look at the contributions our solution, Alcatraz, brings.

4.1 Security Properties at the Wire-Level

The original works on SE [12, 55] did its security verification at the register-transfer level (RTL). Among the common hardware abstraction levels in digital circuit design, RTL is often where the high level circuit design is done. On the other hand, RTL is a leaky abstraction of the physical circuit, and there could be side channels not captured by a RTL description of the hardware, and thus can never be detected with a RTL security proof.

When developing a security proof for Alcatraz, we chose to use a recently introduced framework, Knox [8] from section 3.3, which is built upon the notion of "Information-Preserving Refinement"

(IPR). The proof's goal is to show that, when observing its output or manipulating its input, the circuit's behavior doesn't leak more information than indicated by its functional specification. As a consequence, security proofs constructed in the Knox framework can verify at the wire-level that a circuit is correct and does not have any security vulnerabilities such as the timing side channel.

Wire-level security Wire-level security proof shows that for a circuit, regardless of what an adversary can do with I/O wires, no sensitive information is leaked.

Advantages Wire-level security presents multiple advantages over RTL-level security. Wire-level security can detect bugs present on the wire-level, such as DFT (design for test) bugs in gate-level simulation (GLS). However, RTL-level security proof cannot detect such problems because they happened below RTL's abstraction level.

4.2 Architecture

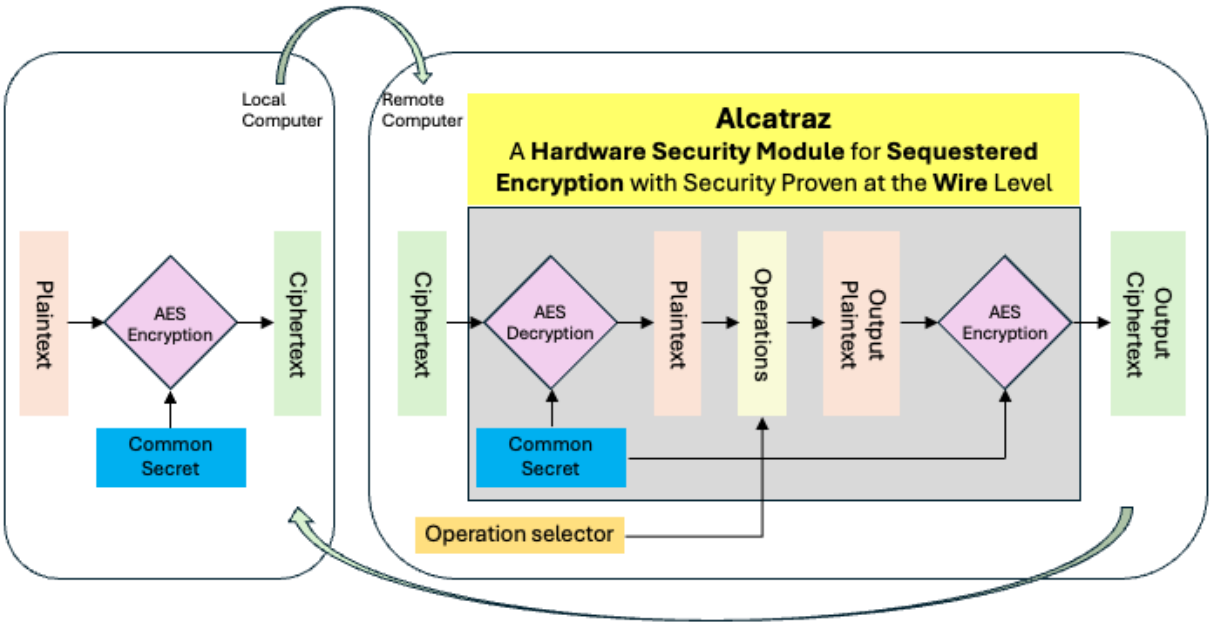

Figure 2: Architecture of Alcatraz.

We implemented Alcatraz as an "encrypted" ALU attached to the "execute" stage of the CPU pipeline on a computer (**Figure 2**). When a user wishes to use Alcatraz to perform secure computation remotely, they use a special instruction that tells the CPU that there is sensitive information involved. The CPU of the remote computer then re-directs that instruction to the encrypted ALU. On the other hand, instructions that are considered non-sensitive can be directed to the regular ALU. When the instruction is executed, ciphertext will be sent into Alcatraz, along with an

operation selector telling the HSM which operation should be performed on the ciphertext. Once decrypted, the plaintext is operated on, then the resulting plaintext will be encrypted before exiting Alcatraz.

4.3 Key Provisioning

In our paper, we use AES, a symmetric key encryption algorithm, as our encryption/decryption engines. This allows for faster encryption times than using public/private key encryption, which is critical because every operation that uses our ALU will have to go through a decryption and encryption operation. This will help reduce the latency of operations in Alcatraz.

In order to perform AES, we require a common key that is known by both parties. In order to obtain a common key without relying on a trusted out-of-band communication channel, we use an authenticated key exchange protocol. One suitable protocol is the signed Diffie–Hellman key exchange [27]. We use the protocol designed by Pan et al. [45] (**Figures 3 and 4**), which has additionally proven tight security of the scheme.

For the digital signature involved, we use elliptic curve cryptography (ECDSA), whose security has been verified by Athalye et al. [9]. We combine ECDSA with the authenticated Diffie–Hellman key exchange by Pan et al. [45] to securely create a common secret key for AES. This allows us to securely bring a key into Alcatraz without additional external sources of trust.

The shared value $\mathrm{SV} = (pk_{\mathrm{RU}}, pk_{\mathrm{HSM}}, X, Y, \sigma_{\mathrm{RU}}, \sigma_{\mathrm{HSM}})$ is something that can be evaluated by both parties on their own. If a third party has compromised the communication in some section, the shared value SV will not match up, and the common secret will not match up, which means the remote user and Alcatraz will be unable to communicate and have to re-initiate the key exchange protocol.

x is randomly chosen from field $\mathbb{F}$	$x \xleftarrow{\$} \mathbb{F}$
x is assigned the value V	$x \leftarrow V$
ECDSA Digital Signature	$\mathrm{ECDSA}(pk, x)$

Figure 3: Notations used in Figure 4.

Remote User		Alcatraz
$x \xleftarrow{\$} \mathbb{F}, X = g^x$		$y \xleftarrow{\$} \mathbb{F}, Y = g^y$
$\sigma_{\mathrm{RU}} \leftarrow \mathrm{ECDSA}(sk_{\mathrm{RU}}, X)$	$\xrightarrow{(X, \sigma_{\mathrm{RU}})}$	
	$\xleftarrow{(Y, \sigma_{\mathrm{HSM}})}$	$\sigma_{\mathrm{HSM}} \leftarrow \mathrm{ECDSA}(sk_{\mathrm{HSM}}, (X, Y))$
$S_{\mathrm{RU}} = \mathrm{H}(\mathrm{SV}, Y^x)$		$S_{\mathrm{HSM}} = \mathrm{H}(\mathrm{SV}, X^y)$

Figure 4: Scheme proposed by Pan et al. [45] using ECDSA.

To avoid sharing a secret every time Alcatraz is called, for every user, we generate one shared secret and save it. This allows us to reuse the secret with that one user, making authenticated

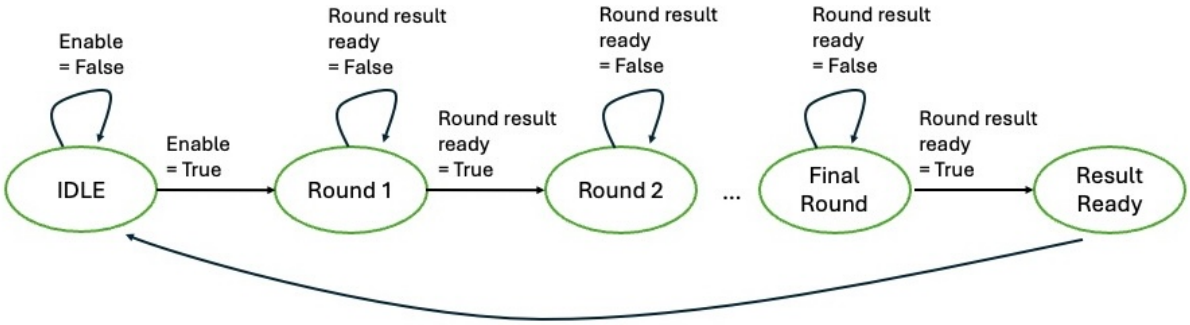

Figure 5: Finite State Machine of the AES encipher module.

key-exchange a one-time operation and lowering computation costs.

4.4 Implementation

We implemented Alcatraz as a circuit on FPGA, using Verilog hardware description language. We also wrote a functional specification of Alcatraz in Racket. We used Yosys' SMT-LIBv2 backend [51] to compile Alcatraz's circuit into an SMT file, which was then converted by `#lang yosys` [7] into a Rosette model. Along with that, we also wrote our own driver and emulator to run in Knox for verification.

To implement AES, we based both the specification and hardware implementation on the NIST FIPS 197 standard of AES [43]. Our implementation focuses on AES-128 (i.e. key length is equal to 128 bits). It is straightforward to extend our implementation to longer key lengths, such as AES-192 and AES-256.

4.4.1 Hardware Circuit

AES-128 is based on three core subfunctions, which are Encipher, Decipher, and KeyExpansion. At a high level, our implementation of the whole AES circuit consists of three Verilog submodules, each corresponding to one of the subfunctions. For the ease of exposition, we do not distinguish between the two terms (submodule vs subfunction) when describing our implementation.

We implement each subfunction as a finite state machine (**Figure 5**) so that we can create state-dependent hints to speed up proofs (see Section 4.4.4). Specifically, we introduce two state variables, `state` and `next_state`, where `next_state` tells the FSM what the next state is. We use the same idea for all other variables we need to keep track of, like input/output and the `result_ready` wire.

For example, in Encipher we always start the FSM from the IDLE state, where the circuit's output is reset to 0 (**Figure 5**). While in this state, we keep checking the `enable` input wire at each clock cycle, and stay in the IDLE state as long as `enable` is `false`. When `enable` is set to

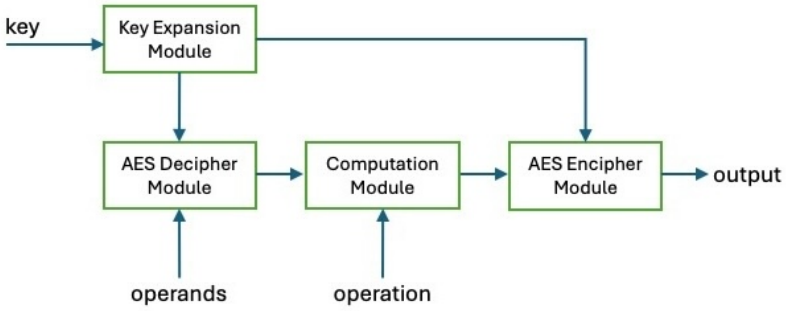

Figure 6: Organization of Alcatraz Circuit. In addition to the inputs shown, all modules have a `clock` and an `enable` input.

`true`, we will move the FSM to "Round 1". In this state, we call a subfunction OneRound, whose inputs we feed based on the round number and previous results.

Once OneRound updates the wire `round_result_ready` to `true`, we take the result immediately and record it in a register. This is necessary because the valid result is only kept on the output wire for one clock cycle. After that, `round_result_ready` is set to false and the result is erased by OneRound. In Encipher, each round's calculation is based on the result from the last round, thus each round's result must be saved for later use.

We repeat this for every state until we reach the final round (the 10-th round for AES-128). Once we complete the final round, we set `result_ready` (of Encipher) to `true`. Similar to OneRound, the wire `result_ready` is only set to `true` for one clock cycle, and the valid result is only maintained for one cycle before getting wiped. As such, Encipher must take the result and save it in a register during the clock cycle so the result (the ciphertext) can be sent back to the user.

Furthermore, rather than executing the three subfunctions in order, due to the expensiveness of KeyExpansion, we can simply store the expanded key along with the shared secret, and have both Encipher and Decipher use the stored expanded key as shown in **Figure 6**. Furthermore, if the same key is re-used multiple times (e.g., if a user uses Alcatraz multiple times), then the KeyExpansion function does not need to be re-evaluated called multiple times; instead we can directly look it up.

Pipelining To enhance performance, we pipeline both the Encipher and Decipher module according the states in **Figure 5**, for a total of 20 stages.

4.4.2 Functional Specification

In our functional specification, we implemented the Galois Field multiplication as lookup tables. This is because the polynomial division was too time costly, so we precomputed the necessary multiplications and hard-coded them as tables. This is equivalent to doing the GF(256) multiplication,

```
(ite* (⊢ (bveq (bv #x00 8) (extract 23 16 k$cef..)) (bv #x63 8))
      (⊢ (bveq (bv #x01 8) (extract 23 16 k$cef..)) (bv #x7c 8))
      (⊢ (bveq (bv #x02 8) (extract 23 16 k$cef..)) (bv #x77 8))
      (⊢ (bveq (bv #x03 8) (extract 23 16 k$cef..)) (bv #x7b 8))
      ...) ...) ...)
```

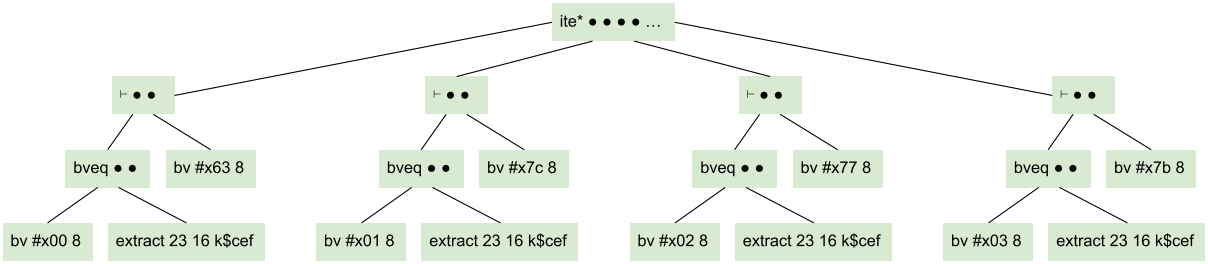

Figure 7: (above) Symbolic term representing a register holding the value `sbox[key[23:16]]`. **(below)** the expression tree for the same symbolic term, where four of the 256 subtrees are shown.

which does not impact the validity of the specification. We then implemented the OneRound and InvOneRound as recursive functions with a counter. When the counter hit 10, the number of rounds in AES-128, the function would then return the result to the parent function. All our specifications are tested with unit tests in Racket.

4.4.3 Automated Verification of Correctness and Security

To verify the correctness of our hardware implementation, we created a small program in Knox (the driver) that calls the hardware to perform operations defined in the specification. Then we used Knox to verify that the output from the driver (on top of the hardware) cannot be distinguished from the output from the specification ("functional equivalence").

To verify the security of our hardware implementation, we created another small program in Knox (the emulator) that calls the specification to mimic wire-level behavior of the hardware. Then we used Knox to prove that the output from the emulator (on top of the specification) cannot be distinguished from the output from the hardware ("physical equivalence").

In either case, Knox will need to symbolically simulate the circuit and keep track of its state over time. Based on the symbolic states, Knox formulates the logical assertions for functional equivalence and phsyical equivalence, and then invokes Rosette, which ultimately calls the Z3 SMT prover, to prove that the assertions hold.

A circuit's symbolic state can be thought of as the collection of current contents of all its registers and wires, represented as symbolic terms. As an example, **Figure 7** shows the symbolic state of a register, where the symbolic expression corresponds to `sbox[key[23:16]]`, i.e. using the second byte of the key as an index for the S-box.

In **Figure 7** the free variable `k$cef..` represents the content of the input wire `k` (the key) at a specific clock cycle. The symbolic term for indexing into the `sbox` array is

```
(ite* (⊢ condition1 value1) (⊢ condition2 value2) ...)
```

which is similar to the `switch` statement in `C++`. When represented as an expression tree, the `ite*` node will have 256 subtrees because `k$cef..` may take 1 of the 256 possible values (`0x00` to `0xff`). As another example, the output of the key expansion module is represented as a symbolic term whose free variables correspond to either the module's inputs or some internal registers.

4.4.4 Speeding Up Proofs with Hints

When a circuit gets more complicated, so too does the symbolic terms capturing its states. Consequently, the size of these terms may blow up, which makes it increasingly slow to check assertions about these terms. This is especially true in circuits whose registers depend on each other over many clock cycles.

As an example, the symbolic term for the key expansion module output corresponds to a tree of more than 1 million nodes, where a node is either `ite*` or some bit vector operation (e.g. `extract` for taking a subvector, or `bvxor` for computing the bitwise xor of two vectors). Even with an automated prover, it becomes very inefficient to directly compare expressions of such size and verify the functional and phsyical equivalence. In fact, when running on a Macbook Pro M2 computer, the Z3 prover cannot finish the correctness proof after running for 4 hours due to the huge term size.

One way to improve the efficiency of the proofs is to utilize additional information about the symbolic terms' structure and dependencies. In the Knox framework, Athalye et al. [8] introduce a concept of "hints", which help speed up verification. Hints are designed to be untrusted, so at worst if a hint is incorrect the automated prover will fail to complete the proof. If the hints are correct but insufficient, the verification process may still take a long time. In any case, adding hints will *not* lead to an incorrect proof.

Among the eight types of hints provided in Knox, we used the following six in our proofs:

- `CONCRETIZE` is a hint suggesting a symbolic term may be proved to have a concrete value. If successfully proved, the said term will be replaced by a concrete value. Otherwise, for example when a term depends on other symbolic terms whose values are unknown, then the term will stay unchanged. This hint is helpful in concretizing values that are already known by the prover, but still expressed as symbolic terms.

- `CASE-SPLIT` is a hint that separates a symbolic term into possible cases. This allows for possible concretization of the symbolic term, which may help reducing the term size. One example is we applied CASE-SPLIT on the enabling wire to splits into two cases (`en`=0 or `en`=1), followed by a `CONCRETIZE` hint on the same wire. As a result, we can eliminate `en` from all subsequent states.

- `REPLACE` is a hint to replace a complicated term with a simplified term (if the prover can prove they are equivalent).

- `REMEMBER`, `SUBSTITUTE`, and `CLEAR` allow us to temporarily abstract away a register and later substitute back it's original term.

Furthermore, we created proof "tactics" based on the state of the FSM of the circuit. A tactic is a snippet of Knox code that invokes some hints and/or updates auxiliary data structures used by hints.

When writing the tactics, the idea is to instruct Knox to abstract away the registers (via `REMEMBER`) before each clock cycle (i.e. treating them as a black box instead of a complicated term), so the prover only needs to focus on the register operations happen within the cycle. Thus, the prover only has to prove a simpler assertion. After a clock cycle is finished, the tactics will instruct Knox to either restore the register's symbolic term (via `SUBSTITUTE`), or replace it with a simpler yet equivalent term (via `REPLACE`). We write our tactics to be aware of the FSM state, because these hints rely on the FSM states to know which register to replace or substitute.

Finally, we observed the prover can still be slow even after adding hints and tactics, and it was due to the order of operations in our specification differing too much from that in the circuit. Based on this finding, we carefully write our specification so that the order of operations in the specification closely align with that in the circuit.

With our optimizations, our final proof was able to complete under 1 minute, a speed up of at least two orders of magnitude.

5 Evaluation

We evaluated the performance of Alcatraz in an important class of secure remote computation problems, Private Information Retrieval.

5.1 Private Information Retrieval

A Private Information Retrival (PIR) protocol is a cryptographic protocol that allows a user to retrieve a specific item from databases without revealing to the server any information on which item is being retrieved. This ensures that the user's query remains private from the database server [14, 30]. PIR protocols are a fundamental building block for larger privacy-ensuring schemes such as oblivious transfer [23], secret sharing schemes [11], certificate transparency [36, 49], and password-breach alerting [33, 56]. More recently, Colombo et al. [15] introduced the first authenticated multi-server PIR scheme, ensuring the security of the user as long as at least one server is honest. Corrigan-Gibbs and Kogan [18] also devised a scheme to allow for sub-linear database lookups without sacrificing server-side storage requirements.

For our project, we implemented a single-server PIR scheme using Alcatraz.

5.2 Evaluation Setup

We follow the common practice to represent PIR queries in one-hot encoding. To perform retrieval, each bit in the query is multiplied with the corresponding entry in the database (i.e. the first query

bit is multiplied with the first entry, etc), and the resulting products are all summed together to produce the final answer. Note that we do not need to worry about overflow because only one of the (decrypted) query bits is set to 1.

To ensure security, queries are encrypted by AES128 before leaving the client. All the above-mentioned multiplication and summation operations are carried out in an encrypted ALU on the server.

In our implementation, we integrated the encrypted ALU with an Ibex RISC-V core [21] and a pipelining vector coprocessor (Vicuna) [47].

Parallel Processing To take advantage of data parallelism, we integrated the encrypted ALU with Vicuna to utilize its support of vector registers, where each register is set to the max supported width of 2048 bits.

We added two types of custom RISC-V instructions to perform encrypted ALU computation:

- **sumprod**(**k**) `vd, v1, v2`
- **sumall**(**k**) `vd, v1, v2`

The main idea is to store 16 128-bit ciphertexts per register, which allows us to perform sum-of-16-products in one instruction. It also allows us to add 32 ciphertexts in one instruction, giving us a massive improvement in performance. The definitions for the custom instructions are given in **Algorithms 1 and 2**.

Input:

`k`: an integer ($0 \leq k < 128$), specifying which 16-bits of `v1` to process

`v1`: the i-th, ..., $(i + 2047)$-th bits of the query (encrypted in AES128)

`v2`: 16 blocks of 128-bit plaintext (the j-th, ..., $(j + 15)$-th entries of the database), to be multiplied with the query bits

Output:

write encrypted sum of products to (`k` mod 16)-th 128-bit block in `vd`

```
1 m = k >> 3;
2 n = k & 0b111 ;
3 p = k & 0b1111 ;
4 /* get the m-th ciphertext */
5 c = v1[m * 128 : m * 128 + 127];
6 d = Decrypt_AES128(c);
7 /* perform sum-of-product operation between 16 query bits and 16 entries */
8 e = d[n * 16] * v2[0 : 127] + ... + d[n * 16 + 15] * v2[1920 : 2047];
9 vd[p * 128 : p * 128 + 127] = Encrypt_AES128(e);
```

Algorithm 1: sumprod(`k`, `v1`, `v2`): calculate the sum of products between 16 bits of the encrypted query vector (`v1`) and the corresponding 16 database entries (`v2`). `k` specifies which 16 bits of the query to process, and what location in `vd` to write the output. `v1` and `v2` are passed in registers. `k` is encoded in `funct7`, a 7-bit field in the RISC-V instruction format.

Input:

`k`: an integer ($0 \leq k < 16$), specifying the output location in `vd`

`v1`: 16 blocks of 128-bit data (encrypted in AES128)

`v2`: 16 blocks of 128-bit data (encrypted in AES128)

Output: encrypted sum of all 32 blocks to `k`-th 128-bit block in `vd`

```
1 for i = 0...15 do
2 |   d_i = Decrypt_AES128(v1[(i * 128) : (i * 128) + 127]);
3 |   e_i = Decrypt_AES128(v2[(i * 128) : (i * 128) + 127]);
4 end
5 f = d_0 + ... + d_15 + e_0 + ... + e_15;
6 vd[k * 128 : p * 128 + 127] = Encrypt_AES128(f);
```

Algorithm 2: sumall(`k`, `v1`, `v2`): calculate the sum of 32 128-bit blocks (16 from `v1` and 16 from `v2`). Inputs are encrypted by AES128. `k` is encoded by 4 bits in `funct7`, a field in the RISC-V instruction format.

Our PIR algorithm Using the new instructions, we can perform PIR by computing sum of products between the query vector and the database entries, and outputting the final (encrypted) sum, which is the retrieved entry (**Figure 8**).

We assume the database has k entries, where k is a multiple of 2048. (If not, we can pad it by adding extra dummy entries.) For clarity, our algorithm assumes each database entry is a 128-bit data block. For databases where each entry has m 128-bit blocks, we perform the same algorithm m times, each time outputting the i-th ($0 \leq i < m$) encrypted block of the final answer.

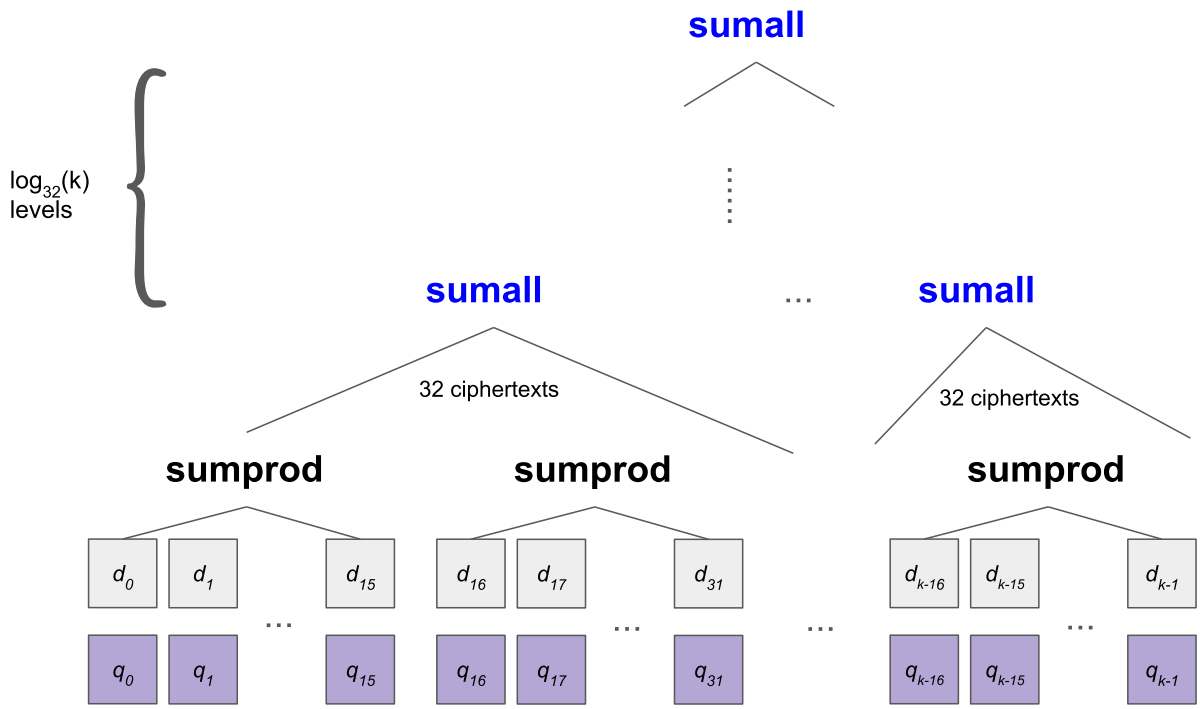

Figure 8: Computation tree for the PIR algorithm. $q_0, ..., q_{k-1}$ represents bits in the query. $d_0, ..., d_{k-1}$ represents the entries in the database, where each entry is a 128-bit block.

Reducing pipeline stalling Our integration includes a modified instruction decoder in the pipelining vector coprocessor, which dispatches the custom instructions to the encrypted ALU. Through experimentation, we identified two default configuration settings that caused unnecessary pipeline stalling:

- The eXtension Interface between the main core (Ibex) and the coprocessor (Vicuna) allowed only 8 outstanding instructions.
- Vicuna's own instruction queue also allowed only 8 outstanding instructions.

We increased both settings from 8 to 16, and observed improved pipeline utilization (cycle-per-instruction for encrypted instruction improved by 4.3%).

5.3 Benchmark

We compare our implementation to the state-of-the-art FHE implementations for several PIR tasks[38]. We implemented PIR algorithm in `C++` and inline assembly, and ran the benchmark

in the Verilator simulator [52]. We measured the performance by taking advantage of Ibex's built-in performance counter to obtain the clock cycle counts. To estimate the actual time, we assumed our hardware running at a clock frequency of 667 MHz, shown to be achievable for AES on FPGA by Lee et al. [32].

Our benchmark includes three datasets whose sizes are the same as those described in [38] (**Table 2**). Our results show that Alcatraz achieves up to 21.3× speedup in computation time over the current state-of-the-art FHE-based PIR protocol.

	Database size (entries)	Entry size	Computation time (s)		Speedup
			Spiral [38]	Alcatraz	
Dataset 1	2^{20}	256B	0.85	**0.040**	21.3×
Dataset 2	2^{18}	30KB	8.99	**1.174**	7.66×
Dataset 3	2^{14}	100KB	2.38	**0.245**	9.71×

Table 2: Comparison of Alcatraz with SpiralStream [38] for Single-Server Private Information Retrieval

6 Conclusion

Microarchitectural side-channel attacks enable an adversary to gain information about sensitive data. These attacks have been shown to recover secrets such as RSA private keys. Hardware bugs and vulnerabilities exploited in such attacks are difficult to detect and prevent, yet they are increasing important because more and more computing is happening remotely in the cloud.

In this paper, we assume a threat model where a powerful adversary has all but physical access to a remote computer. Under this model, we introduce Alcatraz as a solution to secure remote computation. The central piece is a secure ALU which is sandwiched between a decryption step and an encryption step. All sensitive data remain encrypted when they are outside the secure ALU. When performing computations, sensitive data enter the secure ALU as encrypted ciphertext, and the computation results are encrypted before leaving the secure ALU.

We have verified that Alcatraz is secure at the wire-level. In particular, we verified an AES implementation in Verilog, as well as our minimal arithmetic logic unit. Furthermore, Alcatraz brings in the encryption key needed by AES via an authenticated key exchange using ECDSA. The key provisioning process reduces the need for trust in an outside channel for a shared secret. The secure ALU, combined with key provisioning, allows us to create a chip that can add secure remote computation capabilities to existing computers.

We implemented a single-server PIR task using Alcatraz. We utilize the full 2048-bit registers allowed in the Vicuna coprocessor to increase parallelization. We managed to achieve a 7× to 21× speedup when compared to current state-of-the-art FHE implementations.

In terms of query size, for all three benchmark datasets our approach only requires sending just a single number encrypted in AES128, which is 16 bytes (128 bits). When compared with the

query sizes reported for Spiral (8MB to 16MB), our results represent a five-orders-of-magnitude improvement.

7 Acknowledgements

This work was done in the MIT PRIMES research program. I want to thank Slava Gerovitch and Srini Devadas for creating opportunities in the MIT PRIMES CS program. I also thank my mentors, Sacha Servan-Schreiber and Jules Drean, both from MIT CSAIL, for their guidance over the past three years.

I also would like to thank Anish Athalye for helping me with the details of the Knox framework, Thomas Bourgeat for helping with my questions on computer architecture, and Stella Lau for helping with my questions about power-based side channels.

References

[1] Abbas Acar, Hidayet Aksu, A Selcuk Uluagac, and Mauro Conti. A survey on homomorphic encryption schemes: Theory and implementation. *ACM Computing Surveys (Csur)*, 51(4): 1–35, 2018.

[2] Dakshi Agrawal, Bruce Archambeault, Josyula R Rao, and Pankaj Rohatgi. The EM side-channel(s): Attacks and assessment methodologies. In *Cryptographic Hardware and Embedded Systems-CHES 2002: 4th International Workshop Redwood Shores, CA, USA, August 13–15, 2002 Revised Papers 4*, pages 29–45. Springer, 2003.

[3] Rashmi Agrawal, Lake Bu, Alan Ehret, and Michel A Kinsy. Fast arithmetic hardware library for rlwe-based homomorphic encryption. *arXiv preprint arXiv:2007.01648*, 2020.

[4] Ahmad Al Badawi, Bharadwaj Veeravalli, Jie Lin, Nan Xiao, Matsumura Kazuaki, and Aung Khin Mi Mi. Multi-GPU design and performance evaluation of homomorphic encryption on GPU clusters. *IEEE Transactions on Parallel and Distributed Systems*, 32(2):379–391, 2020.

[5] Alejandro Cabrera Aldaya, Cesar Pereida García, Luis Manuel Alvarez Tapia, and Billy Bob Brumley. Cache-timing attacks on RSA key generation. *Cryptology ePrint Archive*, 2018. URL https://eprint.iacr.org/2018/367.

[6] Naseer Amara, Huang Zhiqui, and Awais Ali. Cloud computing security threats and attacks with their mitigation techniques. In *2017 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC)*, pages 244–251. IEEE, 2017.

[7] Anish Athalye, Adam Belay, M Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. Notary: A device for secure transaction approval. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 97–113, 2019.

[8] Anish Athalye, M. Frans Kaashoek, and Nickolai Zeldovich. Verifying hardware security modules with Information-Preserving refinement. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 503–519, Carlsbad, CA, July 2022. USENIX Association. ISBN 978-1-939133-28-1. URL https://www.usenix.org/conference/osdi22/presentation/athalye.

[9] Anish Athalye, Henry Corrigan-Gibbs, Frans Kaashoek, Joseph Tassarotti, and Nickolai Zeldovich. Modular verification of non-leakage for hardware security modules with parfait. In *SOSP 2024*, 2024. to appear.

[10] Clark Barrett and Cesare Tinelli. Satisfiability modulo theories. *Handbook of model checking*, pages 305–343, 2018.

[11] Amos Beimel, Yuval Ishai, Eyal Kushilevitz, and Ilan Orlov. Share conversion and private information retrieval. In *2012 IEEE 27th Conference on Computational Complexity*, pages 258–268. IEEE, 2012.

[12] Lauren Biernacki, Meron Zerihun Demissie, Kidus Birkayehu Workneh, Fitsum Assamnew Andargie, and Todd Austin. Sequestered encryption: A hardware technique for comprehensive data privacy. In *2022 IEEE International Symposium on Secure and Private Execution Environment Design (SEED)*, pages 73–84, 2022. doi: 10.1109/SEED55351.2022.00014.

[13] Joppe W Bos, J Alex Halderman, Nadia Heninger, Jonathan Moore, Michael Naehrig, and Eric Wustrow. Elliptic curve cryptography in practice. In *Financial Cryptography and Data Security: 18th International Conference, FC 2014, Christ Church, Barbados, March 3-7, 2014, Revised Selected Papers 18*, pages 157–175. Springer, 2014.

[14] Benny Chor, Eyal Kushilevitz, Oded Goldreich, and Madhu Sudan. Private information retrieval. *Journal of the ACM (JACM)*, 45(6):965–981, 1998.

[15] Simone Colombo, Kirill Nikitin, Henry Corrigan-Gibbs, David J Wu, and Bryan Ford. Authenticated private information retrieval. In *32nd USENIX security symposium (USENIX Security 23)*, pages 3835–3851, 2023.

[16] Mauro Conti, Nicola Dragoni, and Viktor Lesyk. A survey of man in the middle attacks. *IEEE communications surveys & tutorials*, 18(3):2027–2051, 2016.

[17] Bart Coppens, Ingrid Verbauwhede, Koen De Bosschere, and Bjorn De Sutter. Practical mitigations for timing-based side-channel attacks on modern x86 processors. In *2009 30th IEEE symposium on security and privacy*, pages 45–60. IEEE, 2009.

[18] Henry Corrigan-Gibbs and Dmitry Kogan. Private information retrieval with sublinear online time. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 44–75. Springer, 2020.

[19] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[20] Whitfield Diffie, Paul C Van Oorschot, and Michael J Wiener. Authentication and authenticated key exchanges. *Designs, Codes and cryptography*, 2(2):107–125, 1992.

[21] Islam Elsadek and Eslam Yahya Tawfik. RISC-V resource-constrained cores: A survey and energy comparison. In *2021 19th IEEE International New Circuits and Systems Conference (NEWCAS)*, pages 1–5, 2021. doi: 10.1109/NEWCAS50681.2021.9462781.

[22] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the forty-first annual ACM symposium on Theory of computing*, pages 169–178, 2009.

[23] Yael Gertner, Yuval Ishai, Eyal Kushilevitz, and Tal Malkin. Protecting data privacy in private information retrieval schemes. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, pages 151–160, 1998.

[24] Michael Hutter and Jörn-Marc Schmidt. The temperature side channel and heating fault attacks. In *Smart Card Research and Advanced Applications: 12th International Conference, CARDIS 2013, Berlin, Germany, November 27-29, 2013. Revised Selected Papers 12*, pages 219–235. Springer, 2014.

[25] Mostefa Kara, Abdelkader Laouid, Muath AlShaikh, Ahcène Bounceur, and Mohammad Hammoudeh. Secure key exchange against man-in-the-middle attack: Modified Diffie–Hellman protocol. *Jurnal Ilmiah Teknik Elektro Komputer dan Informatika*, 7(3):380–387, 2021.

[26] Elmira Karimi, Zhen Hang Jiang, Yunsi Fei, and David Kaeli. A timing side-channel attack on a mobile GPU. In *2018 IEEE 36th International Conference on Computer Design (ICCD)*, pages 67–74. IEEE, 2018.

[27] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography, Second Edition*. Chapman & Hall/CRC, 2nd edition, 2014. ISBN 1466570261.

[28] Christoph Kern and Mark R Greenstreet. Formal verification in hardware design: a survey. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 4(2):123–193, 1999.

[29] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. Spectre attacks: Exploiting speculative execution. *Communications of the ACM*, 63(7):93–101, 2020.

[30] Eyal Kushilevitz and Rafail Ostrovsky. Replication is not needed: Single database, computationally-private information retrieval. In *Proceedings 38th annual symposium on foundations of computer science*, pages 364–373. IEEE, 1997.

[31] Brian LaMacchia, Kristin Lauter, and Anton Mityagin. Stronger security of authenticated key exchange. In *International conference on provable security*, pages 1–16. Springer, 2007.

[32] Useok Lee, Ho Keun Kim, Young Jun Lim, and Myung Hoon Sunwoo. Resource-efficient fpga implementation of advanced encryption standard. In *2022 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1165–1169. IEEE, 2022.

[33] Lucy Li, Bijeeta Pal, Junade Ali, Nick Sullivan, Rahul Chatterjee, and Thomas Ristenpart. Protocols for checking compromised credentials. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1387–1403, 2019.

[34] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. Last-level cache side-channel attacks are practical. In *2015 IEEE symposium on security and privacy*, pages 605–622. IEEE, 2015.

[35] Xiaoxuan Lou, Tianwei Zhang, Jun Jiang, and Yinqian Zhang. A survey of microarchitectural side-channel vulnerabilities, attacks, and defenses in cryptography. *ACM Comput. Surv.*, 54(6), jul 2021. ISSN 0360-0300. doi: 10.1145/3456629. URL https://doi.org/10.1145/3456629.

[36] Wouter Lueks and Ian Goldberg. Sublinear scaling for multi-client private information retrieval. In *International Conference on Financial Cryptography and Data Security*, pages 168–186. Springer, 2015.

[37] Rita Mayer-Sommer. Smartly analyzing the simplicity and the power of simple power analysis on smartcards. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 78–92. Springer, 2000.

[38] Samir Jordan Menon and David J. Wu. SPIRAL: Fast, high-rate single-server PIR via FHE composition. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 930–947, 2022. doi: 10.1109/SP46214.2022.9833700.

[39] Daniel Moghimi. Downfall: Exploiting speculative data gathering. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 7179–7193, Anaheim, CA, August 2023. USENIX Association. ISBN 978-1-939133-37-3. URL https://www.usenix.org/conference/usenixsecurity23/presentation/moghimi.

[40] Daniel Moghimi, Berk Sunar, Thomas Eisenbarth, and Nadia Heninger. TPM-FAIL: TPM meets timing and lattice attacks. In *Proceedings of the 29th USENIX Security Symposium*, 2020.

[41] Ciara Moore, Máire O'Neill, Elizabeth O'Sullivan, Yarkın Doröz, and Berk Sunar. Practical homomorphic encryption: A survey. In *2014 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 2792–2795. IEEE, 2014.

[42] Michael Naehrig, Kristin Lauter, and Vinod Vaikuntanathan. Can homomorphic encryption be practical? In *Proceedings of the 3rd ACM workshop on Cloud computing security workshop*, pages 113–124, 2011.

[43] National Institute of Standards and Technology (NIST). Advanced encryption standard (AES). Technical Report 197-upd1, Department of Commerce, Washington, D.C., 2001. URL https://doi.org/10.6028/NIST.FIPS.197-upd1.

[44] Monique Ogburn, Claude Turner, and Pushkar Dahal. Homomorphic encryption. *Procedia Computer Science*, 20:502–509, 2013.

[45] Jiaxin Pan, Chen Qian, and Magnus Ringerud. Signed (group) Diffie–Hellman key exchange with tight security. *Journal of Cryptology*, 35(4):26, 2022.

[46] Bryan Parno. Bootstrapping trust in a "trusted" platform. In *Proceedings of the 3rd Conference on Hot Topics in Security*, HOTSEC'08, USA, 2008. USENIX Association.

[47] Michael Platzer and Peter Puschner. Vicuna: A timing-predictable RISC-V vector coprocessor for scalable parallel computation. In *33rd euromicro conference on real-time systems (ECRTS 2021)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2021.

[48] Jonathan Rose, Abbas El Gamal, and Alberto Sangiovanni-Vincentelli. Architecture of field-programmable gate arrays. *Proceedings of the IEEE*, 81(7):1013–1029, 1993.

[49] Mark D Ryan. Enhanced certificate transparency and end-to-end encrypted mail. *Cryptology ePrint Archive*, 2013.

[50] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. Zombieload: Cross-privilege-boundary data sampling. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 753–768, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450367479. doi: 10.1145/3319535.3354252. URL https://doi.org/10.1145/3319535.3354252.

[51] David Shah, Eddie Hung, Clifford Wolf, Serge Bazanski, Dan Gisselquist, and Miodrag Milanović. Yosys+nextpnr: an open source framework from verilog to bitstream for commercial FPGAs, 2019. URL https://arxiv.org/abs/1903.10407.

[52] Wilson Snyder. Verilator and systemperl. In *North American SystemC Users' Group, Design Automation Conference*, volume 79, pages 122–148, 2004.

[53] François-Xavier Standaert. *Introduction to Side-Channel Attacks*, pages 27–42. Springer US, Boston, MA, 2010. ISBN 978-0-387-71829-3. doi: 10.1007/978-0-387-71829-3_2. URL https://doi.org/10.1007/978-0-387-71829-3_2.

[54] Jakub Szefer. Survey of microarchitectural side and covert channels, attacks, and defenses. *Journal of Hardware and Systems Security*, 3(3):219–234, 2019.

[55] Qinhan Tan, Yonathan Fisseha, Shibo Chen, Lauren Biernacki, Jean-Baptiste Jeannin, Sharad Malik, and Todd Austin. Security verification of low-trust architectures. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, pages 945–959, 2023.

[56] Kurt Thomas, Jennifer Pullman, Kevin Yeo, Ananth Raghunathan, Patrick Gage Kelley, Luca Invernizzi, Borbala Benko, Tadek Pietraszek, Sarvar Patel, Dan Boneh, et al. Protecting accounts from credential stuffing with password breach alerting. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1556–1571, 2019.

[57] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: a high resolution, low noise, L3 cache side-channel attack. In *Proceedings of the 23rd USENIX Conference on Security Symposium*, SEC'14, page 719–732, USA, 2014. USENIX Association. ISBN 9781931971157.