AI VOID FIELD GUIDE

Understanding HEIR: Building Private AI with Homomorphic Encryption

Explore how HEIR, an open-source compiler, enables private AI inference using homomorphic encryption. Learn core FHE concepts, set up your environment, and build privacy-preserving machine learning applications.

AI VOID

2026-08-18 15:54 IST

Contents

CHAPTER 01

The Privacy Imperative: Why We Need Private AI

Imagine harnessing the full power of AI to analyze highly sensitive data – like medical records, financial transactions, or personal communications – without ever exposing the raw information. This isn't science fiction; it's the promise of Private AI, a rapidly evolving field becoming an urgent necessity in our data-driven world.

This chapter begins our deep dive into making Private AI a tangible reality. We'll explore the fundamental "why" behind protecting data during AI computation, introduce the groundbreaking cryptographic concept of Homomorphic Encryption (HE), and meet HEIR, an open-source compiler designed to bridge the gap between complex cryptographic theory and practical AI applications. By the end, you'll grasp the critical need for Private AI and the foundational concepts that underpin its development.

The Data Privacy Challenge in AI

Artificial Intelligence models are insatiable learners; they thrive on vast amounts of data. The more data they process, the more accurate and powerful they become. However, a significant portion of the data that could drive truly transformative AI applications—across healthcare, finance, personal analytics, and beyond—is inherently sensitive. This creates a profound dilemma: how do we unlock AI's potential without compromising individual privacy or violating stringent data protection regulations?

The Growing "Trust Gap" and Its Consequences

Traditional AI approaches typically require sensitive data to be decrypted, processed, and often stored in plain text at some point. This exposure introduces several critical risks and limitations:

- **Data Breaches:** Centralizing plain text data, even with robust security measures, creates an attractive target for malicious actors. A single breach can have catastrophic consequences.
- **Regulatory Hurdles:** Strict data protection laws, such as GDPR, CCPA, and HIPAA, impose severe restrictions on how sensitive data can be collected, processed, and shared. Non-compliance carries heavy penalties.

- **User Mistrust:** Individuals are increasingly concerned about how their personal data is used. This erosion of trust leads to reluctance in sharing information, hindering advancements that could benefit society.
- **Collaboration Barriers:** Organizations often face insurmountable legal or ethical barriers when attempting to share proprietary or sensitive datasets, even for mutually beneficial AI projects.

These challenges collectively contribute to a "trust gap" that significantly impedes the full potential of AI. Without reliable methods to ensure data privacy throughout the AI lifecycle, many innovative and impactful applications remain out of reach.

Understanding Homomorphic Encryption: Computing on Secret Data

What if you could ask a trusted assistant to perform complex calculations on a sensitive document, but they could only see a scrambled, unreadable version of it? And when they returned the scrambled result, you could decrypt it to find the correct answer, knowing your assistant never saw the original content? That's the core idea behind **Homomorphic Encryption (HE)**.

📌 **Key Idea:** Homomorphic Encryption is a cryptographic technique that allows computations to be performed directly on encrypted data. The result of these computations, when decrypted, is identical to the result of performing the same operations on the original, unencrypted data.

Let's unpack this "magic" into a simple flow:

1. **Encrypt Data:** Your sensitive data is encrypted before it leaves your secure environment.
2. **Offload Computation:** This encrypted data is sent to an untrusted computing environment (e.g., a cloud server or an AI service).
3. **Process Encrypted Data:** An AI model or other computation processes the encrypted data. Crucially, the model never sees the original, plain text values; it only interacts with their cryptographically transformed versions.
4. **Receive Encrypted Result:** The computation yields an encrypted output.
5. **Decrypt Result:** You, and only you (or other authorized parties holding the decryption key), can decrypt this result to reveal the meaningful outcome.

The power of HE lies in its ability to keep data encrypted throughout the computation process, eliminating the need to trust the computing environment with your plain text information.

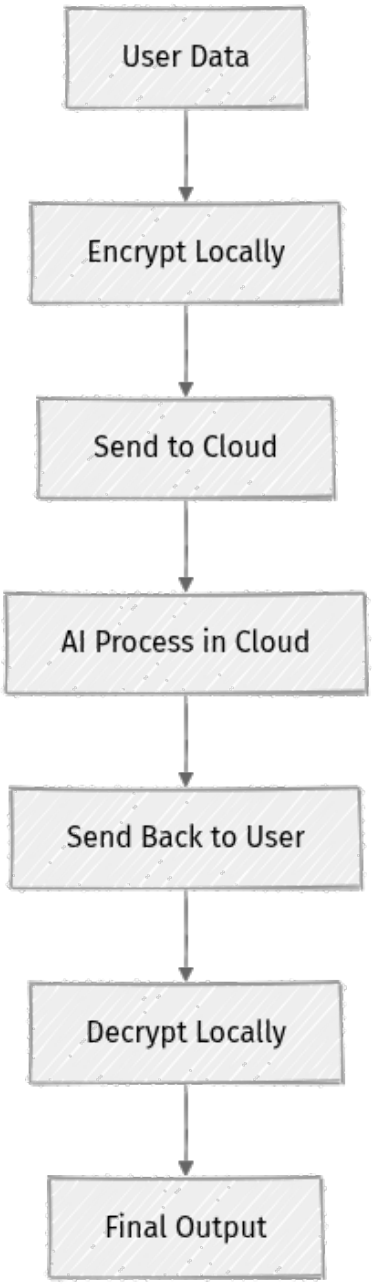
From Partial to Fully Homomorphic Encryption (FHE)

The journey of homomorphic encryption has been a significant cryptographic endeavor:

- **Partially Homomorphic Encryption (PHE):** These schemes allow only one type of mathematical operation (e.g., only addition or only multiplication) to be performed an unlimited number of times on encrypted data. While useful for specific tasks, their scope is limited.
- **Somewhat Homomorphic Encryption (SHE):** SHE schemes improve upon PHE by allowing a limited number of both addition and multiplication operations on encrypted data. This is more versatile but still constrained by the number of operations allowed before the encryption becomes too "noisy" to decrypt correctly.
- **Fully Homomorphic Encryption (FHE):** This is the **holy grail** of homomorphic encryption! FHE schemes enable arbitrary computations (any program, any combination of additions and multiplications) on encrypted data, an unlimited number of times. This breakthrough means you can theoretically run entire, complex AI models on encrypted inputs without ever decrypting them.

FHE is the technology that makes truly private AI powerful and practical, allowing sophisticated algorithms to operate on sensitive data with robust privacy guarantees.

Here's a simplified visual representation of the FHE workflow:



Introducing HEIR: The Compiler for Private AI

While FHE offers incredible potential, working with it directly is notoriously complex. Cryptographic operations differ significantly from standard programming paradigms, requiring deep expertise. This is precisely where **HEIR** steps in.

HEIR (Homomorphic Encryption Intermediate Representation) is an open-source compiler project that aims to make Fully Homomorphic Encryption practical and accessible for developers. Think of HEIR as a sophisticated translator: it takes your high-level computational logic and transforms it into a series of highly optimized FHE-compatible cryptographic operations.

What Does HEIR Do?

HEIR provides an "end-to-end FHE compilation" framework. This means it handles the intricate details of converting general computations into FHE-compatible circuits that can run efficiently on encrypted data. Its primary goal is to empower developers to write applications that leverage FHE without requiring them to become FHE cryptographic experts themselves.

Checked on 2026-08-18: HEIR is an actively developed open-source project. Its GitHub repository (https://github.com/heir-compiler/HEIR) indicates that, while the project is progressing rapidly, certain integrations, particularly between its middle-end and back-end components, are still under active refinement. For developers seeking to generate executables, the project currently directs users to utilize `format_assistant/h` within the repository. This signifies that HEIR is at an exciting, evolving stage, with its capabilities and documentation continually expanding.

Google's Alignment with Private AI

While HEIR is an independent open-source initiative, its goals align closely with Google's broader commitment to privacy-preserving technologies and responsible AI development. Google has been a significant contributor and advocate for various privacy-enhancing technologies, including FHE, differential privacy, and federated learning. Their efforts often span fundamental research, open-source contributions, and the development of frameworks that make these advanced technologies more accessible. HEIR represents a crucial step towards democratizing access to advanced cryptographic techniques for a wider developer community.

⚡ **Quick Note:** HEIR aims to abstract away the cryptographic complexities, allowing you to focus on the logic and design of your AI application, rather than the nuances of the underlying FHE scheme.

Practical Implications & Potential Use Cases

The ability to compute on encrypted data using FHE, facilitated by enabling tools like HEIR, ushers in a new era of privacy-preserving applications across numerous industries:

- **Healthcare:**
  - **Private Diagnostics:** AI models can analyze encrypted patient data (e.g., genetic sequences, medical images) to identify diseases or predict risks, without ever exposing sensitive health information to cloud providers or AI services.
  - **Drug Discovery:** Pharmaceutical companies can securely collaborate on encrypted datasets to accelerate research and development, without revealing proprietary compound structures or patient trial results.
- **Finance:**
  - **Fraud Detection:** Banks can detect fraudulent patterns by analyzing encrypted transaction data, maintaining customer privacy while significantly improving security and reducing financial crime.
  - **Credit Scoring:** Lenders can calculate credit scores based on encrypted financial histories, ensuring fairness and privacy for applicants.
- **Government & Public Sector:**
  - **Secure Data Sharing:** Government agencies can analyze aggregated, encrypted demographic data for policy-making and resource allocation without compromising individual citizen privacy.
  - **Law Enforcement:** Targeted analysis on encrypted data could assist investigations while rigorously adhering to strict privacy mandates.
- **Personalized AI:**
  - **Private Recommendations:** Recommendation engines can analyze encrypted user preferences to suggest products or content, without the platform explicitly learning or storing the user's explicit tastes.
  - **Secure Biometrics:** Authentication systems can verify encrypted biometric data, enhancing security while preventing the exposure of sensitive biological identifiers.

These examples merely scratch the surface. The potential for FHE, especially with tools like HEIR making it more accessible, is immense across any domain where AI interacts with sensitive information.

🧠 Important: It's not a silver bullet (yet!)

While incredibly promising, FHE and tools like HEIR are still maturing. A significant challenge remains the performance overhead; computations on encrypted data are inherently slower and consume more resources than on plain text. However, continuous research and compiler optimizations are rapidly improving efficiency, making FHE increasingly viable for practical applications. Understanding these tradeoffs is crucial for designing effective Private AI solutions.

Step-by-Step Implementation: Preparing Your Environment for HEIR

Before we dive into writing FHE-enabled code in later chapters, let's get your development environment ready by setting up HEIR. This initial setup will provide a foundation for hands-on experimentation.

Prerequisites

To build HEIR, you'll need a C++ compiler and CMake, a cross-platform build system generator.

1. **C++ Compiler:** Ensure you have a modern C++ compiler (like g++ or Clang) installed.

```
# Check if g++ is installed
g++ --version
# If not installed on Ubuntu/Debian:
# sudo apt update && sudo apt install build-essential
```

1. **CMake:** Install CMake. As of **2026-08-18**, we recommend CMake version **3.29.0** or newer.

```
# Check CMake version
cmake --version
# If not installed on Ubuntu/Debian:
# sudo apt update && sudo apt install cmake
```

1. Clone the HEIR Repository

First, you need to get the HEIR source code onto your local machine.

```
git clone https://github.com/heir-compiler/HEIR.git
```

This command downloads the entire HEIR project from its GitHub repository.

2. Navigate to the HEIR Directory

Change your current directory to the newly cloned HEIR project folder.

```
cd HEIR
```

You are now inside the main HEIR project directory.

3. Create a Build Directory

It's a best practice to build software out-of-source. This keeps your source code clean and separates build artifacts.

```
mkdir build
```

This creates a new directory named `build` where all compiled files will reside.

4. Configure the Build System with CMake

Now, navigate into the `build` directory and use CMake to configure the project.

```
cd build
cmake ..
```

- `cd build`: Moves you into the `build` directory.
- `cmake ..`: This command tells CMake to look for the `CMakeLists.txt` file in the parent directory (`..`, which is the HEIR root directory) and generate the necessary build files (e.g., Makefiles on Linux/macOS, Visual Studio solutions on Windows) within the current `build` directory.

You should see output indicating CMake is detecting your system, compilers, and configuring the project.

5. Build HEIR

Finally, compile the HEIR project. This step will take some time, depending on your system's specifications.

```
make -j$(nproc)
```

- `make`: Invokes the build system (e.g., `make` on Unix-like systems).
- `-j$(nproc)`: This is an optimization that tells `make` to use all available CPU cores (`nproc` returns the number of processing units) to speed up the compilation process.

Upon successful completion, you will have built the HEIR compiler components. You won't run a full FHE program yet, but you've successfully prepared the environment for future hands-on exercises.

Mini-Challenge: Pondering Private AI in Your World

Let's ground these abstract concepts in a practical context.

Challenge: Think about an industry or application you are familiar with (e.g., social media, smart homes, online retail, personal fitness trackers). Identify one specific scenario within that domain where using AI with sensitive data currently presents a significant privacy, ethical, or regulatory challenge. Then, briefly describe how Homomorphic Encryption might be applied to mitigate or solve that problem.

Hint: Consider data that is highly personal or proprietary, which an AI could analyze to provide value, but where direct access to the plain text data is problematic. Focus on the flow of data and where encryption would make a difference.

What to observe/learn: This exercise encourages you to connect the abstract concept of FHE to real-world problems, helping you appreciate its transformative potential beyond theoretical discussions. There's no single "right" answer; the goal is creative problem-solving and critical thinking.

Common Pitfalls & Troubleshooting

As HEIR is a cutting-edge tool in an evolving field, you might encounter some specific challenges.

- **1. Performance Overhead:**
  - **Pitfall:** Expect computations on encrypted data to be significantly slower and more resource-intensive than on plain text. This is an inherent characteristic of FHE.
  - **Troubleshooting:** For initial experiments, start with small datasets and simple operations. Be aware that scaling up will require careful optimization and potentially specialized hardware in production. Don't be discouraged by long runtimes for complex tasks; this is an area of active research and improvement.
- **2. Active Development Status:**
  - **Pitfall:** HEIR is an open-source project in active development. Features, APIs, and documentation might change rapidly. The integration between the middle-end and back-end is still being refined.
  - **Troubleshooting:** Always refer to the official HEIR GitHub repository's `README.md` and issues page for the latest updates and specific instructions. If you encounter issues, check if they are known or have been addressed in recent commits. Be prepared for documentation to evolve.
- **3. CMake Configuration Issues:**
  - **Pitfall:** CMake can sometimes fail to configure if required dependencies (like a C++ compiler) are missing, or if your CMake version is too old.
  - **Troubleshooting:** Double-check that all prerequisites (especially CMake version and a C++ compiler) are correctly installed and accessible in your system's PATH. Ensure you run `cmake ..` from inside the `build` directory. If errors persist, review the CMake output carefully for missing packages or configuration problems.

Summary

In this foundational chapter, we've explored the critical need for Private AI in an increasingly data-sensitive world. We've learned that:

- The traditional approach to AI, requiring plain text data, creates a "trust gap" and significant privacy challenges.
- **Homomorphic Encryption (HE)** is a revolutionary cryptographic technique that allows computations to be performed directly on encrypted data.
- **Fully Homomorphic Encryption (FHE)** is the most powerful form, enabling arbitrary and unlimited computations on encrypted data, which is essential for complex AI tasks.
- **HEIR** is an open-source compiler (actively developed as of 2026-08-18) that aims to make FHE practical for developers by translating high-level computational logic into FHE-compatible operations.
- Private AI, powered by FHE and tools like HEIR, holds immense potential to revolutionize industries from healthcare to finance by enabling secure, privacy-preserving data analysis.
- We also took the first practical step by setting up the HEIR development environment, preparing us for hands-on work.

We've successfully set the stage for understanding why Private AI is essential and how foundational technologies like FHE and HEIR contribute to its realization. In the next chapter, we'll delve deeper into HEIR's architecture and begin to explore its components, preparing you for more hands-on exploration.

References

- HEIR Compiler GitHub Repository
- A Guide For HEIR Experiment Evaluation (Original Version)
- CMake Official Website
- Google AI Blog: Advancing Private AI
- Google AI Blog: Homomorphic Encryption

This page is AI-assisted and reviewed. It references official documentation and recognized resources where relevant.

CHAPTER 02

Unpacking Homomorphic Encryption: HE, FHE, and Their Superpowers

In today's data-driven world, the ability to use powerful AI models often clashes with the fundamental need for data privacy. Imagine being able to run complex AI analyses on sensitive customer data, medical records, or financial transactions without ever decrypting that data. Sounds like magic, right? This is the promise of Homomorphic Encryption.

This chapter will demystify Homomorphic Encryption (HE) and its powerful cousin, Fully Homomorphic Encryption (FHE). We'll explore how these cryptographic techniques enable computations on encrypted data, opening new frontiers for private AI. You'll understand the core differences between various HE schemes and see why FHE is a game-changer for privacy-preserving machine learning. Finally, we'll introduce HEIR, an open-source compiler that aims to make FHE practical for developers.

As HEIR is a cutting-edge tool still in active development, this chapter will focus on the fundamental concepts and its conceptual role in the FHE ecosystem. While we won't delve into a line-by-line coding implementation yet, we will outline the conceptual steps involved in using such a compiler to prepare you for future practical application.

Why Privacy in AI Matters

The tension between leveraging data for AI insights and protecting user privacy is one of the most significant challenges in modern technology. Companies and organizations want to derive value from vast datasets, but regulations like GDPR and HIPAA, alongside ethical considerations, demand stringent privacy safeguards.

Traditional methods often involve decrypting data, processing it, and then re-encrypting the results. This creates a "vulnerable window" where sensitive information is exposed, even if only briefly, to the processing environment. This exposure is a significant risk for data breaches and misuse. Homomorphic Encryption offers a radical alternative: compute directly on the encrypted data, eliminating the need for decryption during processing.

The Magic of Homomorphic Encryption (HE)

At its heart, Homomorphic Encryption (HE) is a form of encryption that allows computations to be performed on ciphertext (encrypted data), producing an encrypted result that, when decrypted, matches the result of operations performed on the plaintext (original data).

📌 **Key Idea:** You can "calculate" on locked boxes without ever opening them.

Think of it like this: Imagine you have a special calculator inside a locked vault. You can slide in encrypted numbers, tell the calculator what operations to perform (add, multiply, etc.), and it slides out an encrypted result. You never see the original numbers, and the calculator never sees them either. Only you, with the right key, can decrypt the final result to see the answer.

What Problem Does HE Solve?

HE fundamentally addresses the problem of **data privacy during computation**. It allows cloud providers, AI service providers, or any third party to perform operations on sensitive data without ever gaining access to the raw, unencrypted information. This is crucial for scenarios involving:

- **Cloud Computing:** Securely processing data on untrusted servers.
- **Healthcare:** Analyzing patient data for research without compromising individual privacy.
- **Finance:** Performing fraud detection or risk assessments on encrypted financial transactions.
- **Confidential AI Inference:** Running AI models on private user inputs.

Evolution of Homomorphic Encryption: PHE, SHE, and FHE

Homomorphic Encryption isn't a single technology but a family of cryptographic schemes that have evolved over time. Understanding this evolution helps appreciate the power of Fully Homomorphic Encryption (FHE).

Partially Homomorphic Encryption (PHE)

The earliest forms of HE, known as Partially Homomorphic Encryption (PHE), allow for only one type of mathematical operation to be performed an unlimited number of times on encrypted data.

- **Example:** You might have an encryption scheme that allows you to add encrypted numbers as many times as you want, but you cannot multiply them. Or vice-versa.
- **Why it exists:** PHE schemes are relatively simple and efficient. They solve specific problems where only one type of operation is needed (e.g., summing votes or calculating averages).
- **Limitations:** Their restricted functionality limits their applicability for complex computations like those found in AI models, which often require both additions and multiplications.

Somewhat Homomorphic Encryption (SHE)

Building upon PHE, Somewhat Homomorphic Encryption (SHE) schemes allow for both addition and multiplication operations on encrypted data, but only for a limited number of times.

- **The "Noise" Problem:** HE schemes inherently generate "noise" with each operation. This noise grows with every computation. If the noise becomes too large, the encrypted data can no longer be decrypted correctly. SHE schemes can handle a limited number of operations before the noise overwhelms the signal.
- **Why it exists:** SHE is more powerful than PHE, enabling more complex algorithms. However, the depth of computation (number of sequential operations) is constrained.
- **Limitations:** The limited number of operations still makes SHE challenging for arbitrary computations, especially deep neural networks, which involve many layers of additions and multiplications.

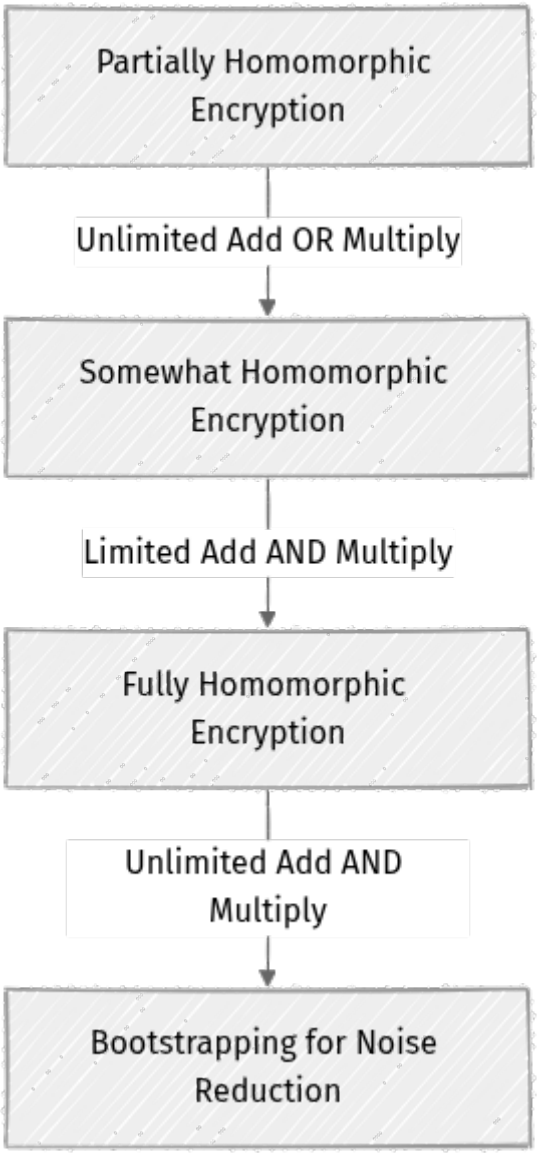
Fully Homomorphic Encryption (FHE)

Fully Homomorphic Encryption (FHE) is the "holy grail" of homomorphic encryption. It allows for an unlimited number of both addition and multiplication operations (and thus, any arbitrary computation) on encrypted data, without ever needing to decrypt it.

- **The Breakthrough: Bootstrapping:** The key innovation that made FHE possible is a technique called "bootstrapping." Bootstrapping effectively "refreshes" the ciphertext by reducing the noise, allowing for an unlimited number of operations. It's like periodically cleaning up the noise in our "locked vault calculator" so it can keep working indefinitely.
- **Why it exists:** FHE unlocks the full potential of privacy-preserving computation for any algorithm, including complex AI models.
- **Challenges:** While incredibly powerful, FHE operations are significantly more computationally intensive and slower than plaintext operations, and even slower than PHE/SHE. This is an active area of research and optimization.

🧠 **Important:** Bootstrapping is the critical technique that transforms Somewhat Homomorphic Encryption into Fully Homomorphic Encryption by managing the inherent "noise" that accumulates during computations.

Here's a quick visual summary of the progression:



The Power of FHE for Private AI Inference

FHE is transformative for AI because it enables **private AI inference**. This means you can use an AI model to make predictions or classifications on data that remains encrypted throughout the entire process.

Consider a scenario where a hospital wants to use an advanced diagnostic AI model hosted by a third-party cloud provider. Using FHE:

1. The hospital encrypts a patient's medical data (e.g., scan results, symptoms).
2. The encrypted data is sent to the cloud provider.
3. The AI model, running on the cloud provider's servers, performs its computations directly on the encrypted patient data.

4. The model outputs an encrypted diagnostic result.
5. This encrypted result is sent back to the hospital.
6. The hospital decrypts the result to see the diagnosis.

At no point does the cloud provider or the AI model itself ever see the patient's raw medical data, ensuring maximum privacy. This makes FHE critical for building trust and enabling AI in highly regulated or sensitive domains.

Introducing HEIR: An End-to-End FHE Compiler

While FHE offers incredible privacy benefits, implementing it directly can be complex and error-prone. This is where compilers like HEIR come into play.

HEIR (Homomorphic Encryption Intermediate Representation) is an open-source compiler designed to make Fully Homomorphic Encryption practical for developers. It acts as a bridge, allowing developers to write high-level code that can then be compiled into operations suitable for FHE execution.

- **What is HEIR?** It's an "end-to-end FHE compiler." This means it aims to take a program, analyze it, and transform it into a series of operations that can be performed efficiently on encrypted data using FHE schemes.
- **Why is it important?** HEIR abstracts away much of the underlying cryptographic complexity of FHE. Instead of manually managing noise, bootstrapping, and specific FHE scheme parameters, developers can focus on the application logic.
- **Google's Role:** HEIR is an open-source project, initially developed by Google, aiming to advance the state of practical FHE. It represents a significant step towards making FHE accessible to a broader developer community.
- **Intermediate Representation (IR):** HEIR uses its own Intermediate Representation (IR) specifically tailored for FHE programs. This IR allows the compiler to optimize the computations for homomorphic execution, potentially reducing the performance overhead.

HEIR's Significance

HEIR's goal is to enable developers to build privacy-preserving AI applications with relative ease. It allows researchers and engineers to experiment with FHE without becoming FHE cryptography experts themselves. By providing a compilation pipeline, HEIR seeks to:

- **Simplify Development:** Reduce the barrier to entry for FHE.
- **Optimize Performance:** Translate high-level code into efficient FHE operations.
- **Enable Broader Adoption:** Make FHE a viable option for real-world private AI solutions.

Preparing for HEIR: Understanding the Compiler's Role and Current State

As of **2026-08-18**, HEIR is an active and evolving open-source project. This means its documentation and capabilities are still under development, and specific version numbers for stable releases might not be readily available in the traditional sense.

⚡ **Quick Note:** When working with cutting-edge tools like HEIR, expect rapid changes and evolving best practices. Always refer to the official HEIR GitHub repository for the most up-to-date information.

Core Setup Requirements (for Building HEIR)

To work with or contribute to HEIR, you would typically need a development environment capable of building C++ projects.

- **CMake:** HEIR uses CMake for configuring its build system.
    - **Installation:** Ensure you have CMake installed on your system.

```
        # On Ubuntu/Debian
        sudo apt update && sudo apt install cmake -y

        # On macOS (with Homebrew)
        brew install cmake
```

```
- **Version:** As of 2026-08-18, a recent stable version of CMake (e.g.,
3.20+) is generally recommended for modern C++ projects. You can check your
version with `cmake --version`.
```

Understanding HEIR's Current State for Implementation

It's important to note a critical practical limitation mentioned in the HEIR repository:

> "The integration between Middle-End and Back-End is not yet well-implemented. If you require an executable, please use `format_assistant/h`." — HEIR GitHub README

This statement is crucial. It indicates that while HEIR provides the foundational compiler infrastructure, generating fully functional, end-to-end FHE executables directly from arbitrary high-level code might still be a work in progress. For developers, this means that a direct, step-by-step code-along for a complete application isn't practical at this specific stage of HEIR's development.

Instead, the value lies in understanding the conceptual workflow and the role HEIR plays in enabling FHE.

Conceptual Steps to Using an FHE Compiler (like HEIR)

While a concrete code implementation is not yet fully stable or documented for HEIR's end-to-end use, we can outline the conceptual "step-by-step" process for how a developer would interact with an FHE compiler like HEIR once it reaches maturity. This helps to build an understanding of the compilation pipeline.

1. **Define Your Private Computation:**
    - **What you do:** You start by defining the specific function or algorithm you want to execute privately (e.g., a Euclidean distance calculation, a simple neural network layer, an inner product).
    - **HEIR's role:** HEIR would expect this logic in a format it can understand – perhaps a specific subset of C++, a domain-specific language (DSL), or an intermediate representation (IR) like MLIR.
    - **Example (conceptual):** Imagine writing a function `compute_distance(a, b)` that takes two vectors and calculates their Euclidean distance.

2. **Compile to FHE Intermediate Representation (IR):**
    - **What you do:** You pass your high-level computation to the HEIR compiler's front-end.
    - **HEIR's role:** The compiler parses your code and translates it into its specialized FHE Intermediate Representation (IR). This IR is designed to represent operations in a way that's amenable to homomorphic execution and optimization.
    - **Why it matters:** This step abstracts away the complexities of FHE schemes, allowing you to focus on the algorithm.
3. **Optimize for Homomorphic Execution:**
    - **What you do:** (This step is largely automated by the compiler).
    - **HEIR's role:** The compiler's middle-end analyzes the FHE IR to apply various optimizations. This includes strategies to minimize noise growth, schedule bootstrapping operations efficiently, and select optimal FHE parameters (e.g., polynomial degree, prime moduli) for performance and security.
    - **Why it matters:** FHE is computationally expensive. Optimization is critical to make it practical for real-world applications.
4. **Generate FHE Backend Code:**
    - **What you do:** You instruct the compiler to generate the low-level FHE operations.
    - **HEIR's role:** The compiler's back-end takes the optimized FHE IR and translates it into calls to a specific FHE library (e.g., SEAL, HElib, TFHE). This generated code will perform the actual homomorphic additions, multiplications, and bootstrapping.
    - **Why it matters:** This is the bridge between your high-level logic and the cryptographic primitives.

5. **Execute on Encrypted Data:**
    - **What you do:** In your application, you would encrypt your sensitive input data using the chosen FHE library's encryption functions. Then, you execute the FHE backend code generated by HEIR on this encrypted input.
    - **HEIR's role:** While HEIR generates the code, the execution happens within your application, leveraging the underlying FHE library.
    - **Why it matters:** This is the moment of private AI inference – computation happens entirely on ciphertext.
6. **Decrypt the Encrypted Result:**
    - **What you do:** Once the homomorphic computation is complete, you receive an encrypted result. Using your secret key, you decrypt this result to obtain the final plaintext output.
    - **HEIR's role:** HEIR's job is done; it provided the compilation. Decryption is a function of the FHE library and your application.
    - **Why it matters:** Only the authorized party with the key ever sees the unencrypted outcome.

This conceptual workflow highlights how HEIR aims to streamline the development process for FHE applications, allowing developers to focus on the logical aspects of their private AI models rather than the intricate cryptographic details.

Mini-Challenge: Designing a Private AI Scenario

Let's solidify your understanding of FHE's potential. No coding required, just thinking!

Challenge: Imagine you are a startup developing a new health monitoring app. Your app collects very sensitive user biometric data (heart rate, sleep patterns, activity levels). You want to offer a premium feature: an AI model that analyzes this data to provide personalized health risk assessments. How would you design this feature to ensure maximum user privacy using FHE?

Think about:

- What data would be encrypted?
- Who holds the encryption key?
- Where would the AI model run?

- What would be the input to the AI model (encrypted or plaintext)?
- What would be the output of the AI model?
- Who decrypts the final result?

Hint: Focus on the flow of data and who has access to the plaintext at each stage.

Common Pitfalls & Troubleshooting in FHE Development

Working with FHE, even with compilers like HEIR, presents unique challenges that developers should be aware of.

⚠️ What can go wrong: Performance Overhead

FHE operations are significantly slower and consume more resources than equivalent plaintext operations. Expect execution times to be orders of magnitude longer (e.g., seconds or minutes vs. milliseconds for simple operations).

- **Troubleshooting:**
  - **Optimization:** FHE compilers like HEIR aim to optimize, but careful algorithm design is still crucial. Simplify your models.
  - **Hybrid Approaches:** Sometimes, only the most sensitive parts of a computation are done homomorphically, while less sensitive parts are done in plaintext (with careful isolation).
  - **Hardware Acceleration:** Dedicated FHE accelerators are an active area of research to improve performance.

⚠️ What can go wrong: Complexity of FHE Schemes

While HEIR abstracts away some complexity, understanding the underlying FHE schemes (e.g., CKKS for approximate real numbers, BFV/BGV for exact integers) can be beneficial for optimizing performance and choosing the right scheme for your specific task.

- **Troubleshooting:**
  - **Specialization:** Different FHE schemes are better suited for different types of operations. Choosing the wrong one can lead to inefficiency or incorrect results.
  - **Learning Curve:** Be prepared for a steeper learning curve compared to traditional cryptography.

⚠️ What can go wrong: "Noise" Management and Bootstrapping Cost

The "noise" inherent in FHE operations and the computationally expensive "bootstrapping" process are fundamental challenges. If not managed correctly, noise can lead to incorrect decryption. Bootstrapping can add significant latency (e.g., hundreds of milliseconds to seconds per refresh).

- **Troubleshooting:**
    - **Parameter Selection:** Choosing correct FHE parameters (e.g., polynomial degree, prime moduli, number of levels) is crucial and impacts both security and performance. Incorrect parameters can lead to security vulnerabilities or unmanageable noise.
    - **Compiler Role:** Compilers like HEIR are designed to automate some of this parameter management and bootstrapping scheduling, but vigilance and understanding are still required.

⚠️ What can go wrong: Early Stage Tooling

As of 2026-08-18, HEIR and other FHE tooling are still evolving. This means documentation might be incomplete, APIs might change frequently, and active community support might be nascent compared to mature technologies.

- **Troubleshooting:**
    - **Community Engagement:** Engage with the HEIR community on GitHub, report issues, and contribute. Your feedback helps shape the future.
    - **Refer to Source:** When in doubt, the source code and official repository README are your best friends.
    - **Expect Iteration:** Be prepared for breaking changes and the need to adapt your code as the tools mature.

Summary

In this chapter, we've journeyed into the fascinating world of Homomorphic Encryption, unlocking the secrets behind computing on encrypted data.

Here are the key takeaways:

- **Homomorphic Encryption (HE)** allows computations on ciphertext, producing an encrypted result that decrypts to the same value as if computed on plaintext.

- **Partially Homomorphic Encryption (PHE)** permits unlimited operations of one type (e.g., additions).
- **Somewhat Homomorphic Encryption (SHE)** allows a limited number of both additions and multiplications.
- **Fully Homomorphic Encryption (FHE)** is the ultimate goal, enabling an unlimited number of both additions and multiplications through a technique called **bootstrapping**.
- **FHE is crucial for Private AI Inference**, allowing AI models to process sensitive data without ever seeing it in plaintext.
- **HEIR** is an open-source, end-to-end FHE compiler developed by Google to simplify and optimize FHE development.
- Working with HEIR (as of 2026-08-18) means engaging with a cutting-edge tool in active development, requiring attention to its evolving state and inherent FHE challenges like performance and noise management. The "step-by-step" process is currently conceptual, reflecting the compilation pipeline rather than direct code implementation.

You now have a solid conceptual foundation for understanding how FHE makes private AI a reality. In the next chapter, we'll dive deeper into the architecture of HEIR and begin to explore how it translates high-level operations into FHE-compatible code. Get ready to see how this powerful compiler aims to bring FHE to the masses!

References

- HEIR GitHub Repository
- A Guide For HEIR Experiment Evaluation (Original Version) - HEIR GitHub
- Homomorphic Encryption - Wikipedia
- What is Homomorphic Encryption? - IBM Research
- What is Fully Homomorphic Encryption? - Zama

This page is AI-assisted and reviewed. It references official documentation and recognized resources where relevant.

CHAPTER 03

Introducing HEIR: An Open-Source Compiler for FHE-Powered AI (Current Status)

In an increasingly data-driven world, the tension between leveraging powerful AI models and protecting sensitive user information is growing. How can we perform complex computations on data without ever exposing the underlying sensitive details? This chapter introduces you to a groundbreaking solution: Homomorphic Encryption (HE) and HEIR, an open-source compiler designed to make privacy-preserving AI a practical reality.

We'll dive into the core concepts of Homomorphic Encryption, understand why Fully Homomorphic Encryption (FHE) is particularly challenging, and then explore HEIR's role as an "end-to-end FHE compiler." You'll learn about its architecture, its current development status (as of 2026-08-18), and how it aims to bridge the gap between AI models and secure, encrypted computation. This knowledge is crucial for any developer looking to build the next generation of privacy-first AI applications.

The Privacy Conundrum in AI

Imagine you have a powerful AI model that can detect diseases from medical images, but the images contain highly sensitive patient data. Or perhaps you want to run a fraud detection model on financial transactions, but privacy regulations prevent you from directly accessing raw customer data. The traditional approach requires decrypting data, processing it, and then re-encrypting it, creating a vulnerable window where data is exposed.

This exposure is a significant hurdle for deploying AI in sensitive domains like healthcare, finance, or government. It's a problem that Homomorphic Encryption seeks to solve by allowing computations directly on encrypted data.

Homomorphic Encryption (HE) vs. Fully Homomorphic Encryption (FHE)

Before we meet HEIR, let's clarify the fundamental cryptographic concepts it builds upon. Understanding these distinctions is key to appreciating HEIR's purpose.

Homomorphic Encryption (HE)

📌 **Key Idea:** Homomorphic Encryption allows computations on encrypted data without decrypting it first.

Think of it like a secure processing facility. You send encrypted data to this facility. Inside, operations are performed on the encrypted data. The facility never sees the original sensitive information, only its scrambled form. When the encrypted result is returned to you, only you can decrypt it to reveal the plaintext outcome.

However, traditional HE schemes often have limitations:

- **Limited Operations:** Some schemes might only support addition, or only multiplication, but not both. This restricts the types of programs you can run.
- **Finite Operations:** They might only allow a fixed number of operations before the "noise" in the ciphertext grows too large. This "noise" is an inherent part of HE and, if unchecked, can make decryption impossible.

Fully Homomorphic Encryption (FHE)

This is where the "Fully" comes in, and it's a game-changer for AI.

🧠 **Important:** FHE allows any arbitrary computation to be performed on encrypted data, an unlimited number of times, without ever decrypting it. It achieves this through a process called "bootstrapping," which essentially "refreshes" the ciphertext to reduce noise.

This means you can run an entire AI inference model – which involves many additions, multiplications, and other complex operations – entirely on encrypted inputs. The model's weights themselves can also be encrypted, adding another layer of privacy. The user provides encrypted data, the cloud server processes it while it remains encrypted, and returns an encrypted result. Only the user can decrypt the final output.

The challenge? FHE is computationally intensive. Operations on encrypted data are significantly slower and require more memory than operations on plaintext data. This is why "compilers" like HEIR are so crucial: they aim to optimize FHE computations to make them practical.

Introducing HEIR: The FHE Compiler

HEIR (Homomorphic Encryption Intermediate Representation) is an open-source project that aims to be an end-to-end compiler for Fully Homomorphic Encryption programs. It's designed to take a program written for plaintext execution (like an AI model) and transform it into an optimized FHE program that can run securely on encrypted data.

What Problem Does HEIR Solve?

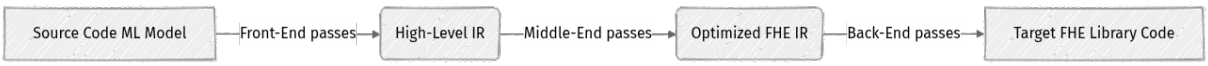
HEIR tackles the complexity and inefficiency of FHE directly. Manually translating an AI model into an FHE-compatible circuit and then optimizing it for performance is incredibly difficult and error-prone. HEIR aims to automate this process, allowing developers to focus on their AI logic rather than the intricate details of FHE cryptography.

⚡ **Real-world insight:** Google is involved in the development of HEIR. While the project is open-source and community-driven, Google's contributions signify a commitment to advancing the practical application of FHE for private AI, particularly in cloud environments where data privacy is paramount.

How HEIR Works (Conceptual Flow)

At its core, HEIR functions much like a traditional compiler, but with a specific focus on FHE. It translates high-level code into an optimized form suitable for FHE execution.

Here's a simplified view of its conceptual stages:



1. **Front-End:** This stage takes your original program (e.g., an ML model defined in a common format or a high-level language) and converts it into an initial, high-level Intermediate Representation (IR). Think of IR as a standardized, abstract way to represent your program's operations.
2. **Middle-End:** Here, the real FHE magic begins. This stage performs FHE-specific optimizations on the IR. This might involve reordering operations, batching data to utilize FHE's parallel computation capabilities, or selecting specific FHE schemes and parameters to improve efficiency and manage noise.
3. **Back-End:** Finally, the optimized FHE IR is translated into concrete code that can be executed by an existing FHE library (e.g., SEAL, HElib, Concrete). These libraries handle the actual cryptographic operations.

This layered approach allows HEIR to integrate with various FHE libraries and target different hardware, providing flexibility and future-proofing as FHE technology evolves.

HEIR's Current Status (as of 2026-08-18)

It's important to understand that HEIR is an actively developed, cutting-edge project. As such, its capabilities and documentation are continually evolving. This means that while it's a powerful research and development tool, its full end-to-end capabilities are still maturing.

The official HEIR GitHub repository provides the most up-to-date information. As of our check on 2026-08-18, the repository notes a critical limitation for developers:

⚠️ **What can go wrong:** The integration between HEIR's Middle-End and Back-End is currently not yet well-implemented. This means that while HEIR can process and optimize the Intermediate Representation, generating a fully executable FHE program that links directly to an FHE library might require manual intervention or specific tools. The documentation specifically mentions that "If you require an executable, please use `format_assistant/h`." This indicates that direct, seamless compilation to a runnable FHE program is still under active development.

This limitation means that while HEIR is a powerful tool for understanding and optimizing FHE computation graphs, it's not yet a "black box" compiler where you feed in an AI model and get a runnable encrypted binary without further steps. Developers should expect to engage with the generated IR and potentially use helper tools or manual integration with FHE libraries for full end-to-end execution.

Step-by-Step Implementation: Setting Up HEIR (Conceptual Build)

While generating a fully executable FHE program is still evolving, you can still clone and build the HEIR compiler itself to explore its components and contribute to its development. This will give you the tools to work with HEIR's Intermediate Representation.

Prerequisites

To build HEIR, you'll need `CMake`, a cross-platform build system. CMake helps manage the compilation process for complex projects.

- **CMake:** Ensure you have CMake version **3.20 or newer** installed.
    - On Ubuntu/Debian:

```
        sudo apt update && sudo apt install cmake
```

```
- On macOS (with Homebrew):
```

```
        brew install cmake
```

```
- For other systems, refer to the [official CMake documentation](https://
cmake.org/download/) for installation instructions.
```

Cloning the HEIR Repository

First, let's get the source code from GitHub. This is where the entire HEIR project resides.

```
# Clone the HEIR repository from GitHub
git clone https://github.com/heir-compiler/HEIR.git

# Navigate into the newly cloned directory
cd HEIR
```

This command downloads the entire project to your local machine, creating a directory named `HEIR`.

Building HEIR

Now, let's configure and build the project using CMake. This process will compile the HEIR compiler itself, along with its various passes and tools that operate on FHE IR.

1. **Create a build directory:** It's good practice to build outside the source directory to keep things clean.

```
    mkdir build
    cd build
```

1. **Configure the project with CMake:** This command tells CMake to look for the `CMakeLists.txt` file in the parent directory (`..`) and generate platform-specific build files (e.g., Makefiles on Linux/macOS, Visual Studio solutions on Windows).

```
cmake ..
```

```
You might see output indicating that CMake is finding dependencies and
configuring the build environment. If there are missing dependencies (like
LLVM/MLIR components), CMake will usually report them here.
```

1. **Build the project:** This command starts the actual compilation process using the build files generated by CMake.

```
cmake --build .
```

```
This might take a few minutes, depending on your system's specifications and
the number of available processor cores. You'll see compiler output as it
builds the various components of HEIR.

Upon successful completion, you will find HEIR executables and libraries
within your `build` directory. For example, `build/bin/heir-opt` might be one
of the key tools, allowing you to run various optimization passes on FHE IR.
```

Mini-Challenge: Exploring the IR

Now that you've successfully built HEIR, let's take a closer look at its internal structure. Understanding where different components live can help demystify how compilers work.

Challenge: Navigate into the `build/lib` and `build/bin` directories. Can you identify where the Intermediate Representation (IR) definitions might be located, or where the compiler's optimization passes are implemented?

Hint:

1. Start by looking at the original source directories: `HEIR/lib/Dialect` and `HEIR/lib/Transforms`.
2. Then, observe how these source components are organized and compiled into the `build/lib` directory (e.g., `.so` or `.dylib` files).
3. In `build/bin`, look for executables that suggest they can process or optimize IR, like `heir-opt`.

What to Observe/Learn: This exercise helps you understand the modular nature of a compiler. You'll see how different components are compiled into libraries and executables, giving you a glimpse into where the "brains" of the FHE optimization live. This hands-on exploration reinforces the conceptual flow we discussed earlier.

Common Pitfalls & Troubleshooting

Working with an experimental compiler like HEIR can present unique challenges. Here are some common issues and how to approach them.

1. **Dependency Hell:**
    - **Problem:** CMake errors indicating missing required libraries (e.g., specific versions of LLVM, MLIR, or other development tools).
    - **Solution:** Carefully read the error messages. HEIR often relies on specific versions of MLIR/LLVM, which can be tricky to manage. Always check the `HEIR/README.md` and `HEIR/docs` within the cloned repository for the exact dependency requirements and recommended installation methods. You might need to install these manually or via your system's package manager, sometimes even building them from source if specific versions are required.
2. **"Not Yet Well-Implemented" Backend:**
    - **Problem:** You've compiled HEIR, but you're struggling to generate a fully executable FHE program from your AI model that runs end-to-end.
    - **Solution:** Remember HEIR's current development status (as of 2026-08-18). The project is a powerful tool for generating and optimizing FHE IR, but its direct, seamless compilation to runnable FHE programs is still an area of active research. Focus on using HEIR to understand and manipulate the IR, and then manually integrate that IR with a specific FHE library (like SEAL or HElib) using helper tools such as `format_assistant/h` as mentioned in the HEIR documentation. This approach requires a deeper understanding of FHE libraries.

3. **Evolving API and Documentation:**
    - **Problem:** Code examples or documentation you find online from blogs or older tutorials are outdated and no longer work with the latest HEIR version.
    - **Solution:** Always refer to the official HEIR GitHub repository as the primary source of truth. The `main` branch is the most current, and the `docs` directory within the repository will have the latest guides and examples. Given the rapid pace of development in this field, frequent checks against the official source are essential.

Summary

In this chapter, we've taken a significant step into the world of privacy-preserving AI by introducing HEIR.

Here are the key takeaways:

- We distinguished between **Homomorphic Encryption (HE)**, which allows limited operations on encrypted data, and **Fully Homomorphic Encryption (FHE)**, which enables arbitrary, unlimited computations on encrypted data.
- We learned that **HEIR** is an open-source compiler designed to bridge the gap between AI models and FHE, aiming to make secure, encrypted computation practical by optimizing the complex FHE operations.
- We discussed HEIR's **conceptual architecture**, involving Front-End, Middle-End, and Back-End stages that transform high-level programs into FHE-compatible code.
- Crucially, we acknowledged HEIR's **current development status** (as of 2026-08-18), noting that while powerful, its end-to-end compilation to executable FHE programs is still evolving and may require manual integration with FHE libraries.
- Finally, we walked through the **basic steps to build HEIR** from its source code and explored its structure through a mini-challenge, preparing you to dive deeper into its capabilities.

The journey into privacy-preserving AI with FHE is just beginning. In the next chapter, we'll start exploring how to define simple computations and how they might be represented within HEIR's Intermediate Representation, laying the groundwork for more complex AI applications.

References

1. HEIR GitHub Repository: https://github.com/heir-compiler/HEIR
2. A Guide For HEIR Experiment Evaluation (Original Version): https://github.com/heir-compiler/HEIR/raw/refs/heads/main/README.md
3. CMake Official Documentation: https://cmake.org/documentation/

This page is AI-assisted and reviewed. It references official documentation and recognized resources where relevant.

CHAPTER 04

Setting Up Your HEIR Development Environment (2026-08-18)

Setting Up Your HEIR Development Environment (2026-08-18)

Welcome back, privacy-preserving AI enthusiast! In our previous chapters, we peeled back the layers of homomorphic encryption (HE) and fully homomorphic encryption (FHE), discovering how they enable computation on encrypted data. We then introduced HEIR, an ambitious open-source compiler designed to make FHE practical for AI inference. Now, it's time to roll up our sleeves and get HEIR running on your machine.

This chapter will guide you through the essential steps to set up your HEIR development environment. We'll cover everything from installing necessary prerequisites like CMake and a C++ compiler to cloning the HEIR and LLVM projects, and finally building the HEIR compiler itself. By the end, you'll have a working HEIR environment, ready for your first foray into private AI inference.

Setting up a compiler from source can sometimes feel like a complex puzzle, but we'll break it down into the smallest, most manageable pieces. Remember, HEIR is an actively developed project, so while these instructions are accurate as of 2026-08-18, slight adjustments might be needed in the future.

Why a Proper Setup Matters for Private AI

A correctly configured development environment is the bedrock of any successful project, especially when diving into complex domains like homomorphic encryption and compiler development. For HEIR, an FHE compiler built on the LLVM/MLIR infrastructure, a precise setup ensures that:

- **Dependencies are Met:** HEIR relies on specific versions and configurations of tools like CMake, C++ compilers, and the MLIR framework. An incorrect setup can lead to cryptic compilation errors and wasted debugging time.
- **Optimal Performance:** Compiler builds can be resource-intensive. A streamlined setup prevents unnecessary rebuilds and ensures you're using efficient build systems like Ninja, which significantly speeds up development cycles.

- **Future Compatibility:** As HEIR evolves, having a clean, modular setup makes it easier to update components and adapt to new versions without breaking your existing work. This is crucial for an experimental project.

Think of it like preparing a specialized workshop for a delicate engineering task. You wouldn't start building a precision instrument without ensuring all your tools are present, sharp, and calibrated. This meticulous preparation saves significant time and frustration down the line.

Core Concepts: Understanding the HEIR Build Architecture

Before we start typing commands, let's understand the landscape of the HEIR build process. HEIR isn't a standalone tool; it's deeply integrated into the broader LLVM ecosystem. Understanding this architecture helps demystify the setup steps.

What is LLVM and MLIR?

LLVM (Low-Level Virtual Machine) is a collection of modular and reusable compiler and toolchain technologies. It's famous for its flexible intermediate representation (IR) and its ability to optimize code for various architectures. LLVM provides the foundation for many modern compilers, including Clang.

MLIR (Multi-Level Intermediate Representation) is an extension of the LLVM philosophy. It's designed to address the complexity of modern compilers by allowing multiple IRs at different levels of abstraction. This means you can represent code at a very high, domain-specific level (like FHE operations) and gradually lower it to more general, machine-level instructions. HEIR leverages MLIR to represent FHE programs, enabling sophisticated, FHE-specific optimizations before targeting specific HE backends.

📌 **Key Idea:** HEIR uses MLIR as its foundation, allowing it to represent and optimize FHE programs in a structured, multi-level way, benefiting from LLVM's robust compiler infrastructure.

The Role of CMake and Build Systems

CMake is a cross-platform, open-source build system generator. Instead of directly compiling code, CMake reads configuration files (`CMakeLists.txt`) and generates native build tool files (like Makefiles or Ninja build files) that are then used by your chosen build system (e.g., `make` or `ninja`) to compile the source code.

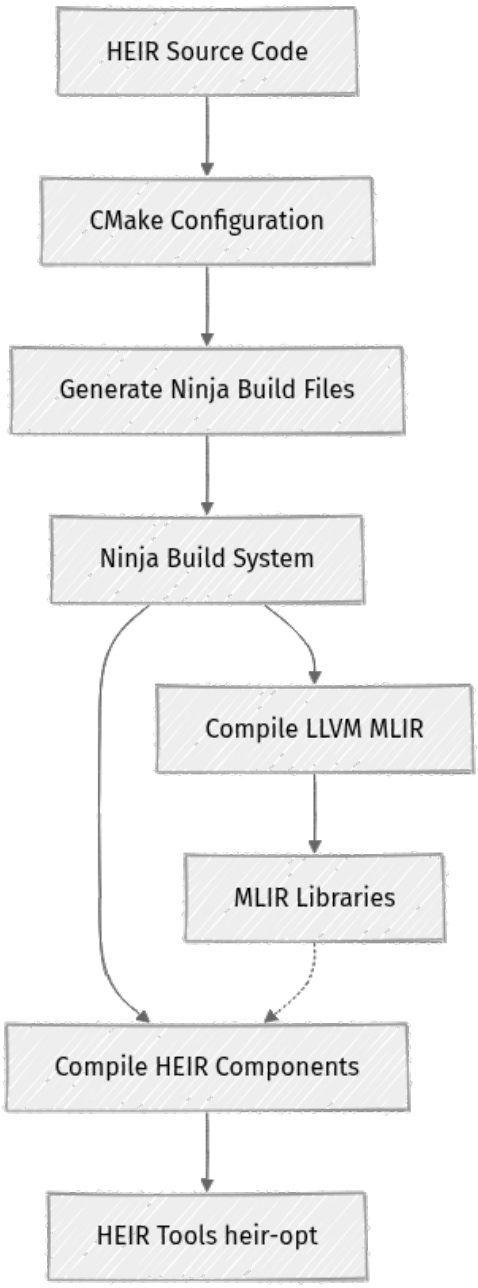
This abstraction is crucial for large, complex projects like HEIR and LLVM, which need to support various operating systems, hardware architectures, and compiler toolchains. You'll run CMake once to configure the build, and then use `ninja` (our recommended build tool) repeatedly to compile and link the project as you make changes.

HEIR's Current Development State

It's important to reiterate that HEIR is an actively developed, experimental project. As of 2026-08-18, the integration between its "Middle-End" (MLIR-based FHE optimization) and "Back-End" (targeting specific FHE libraries) is still evolving. The official GitHub repository explicitly states that if you require an executable from HEIR, you might need to use `format_assistant/h` (a specific utility within the project). This means our focus will be on successfully building the core HEIR compiler components, which will include tools like `heir-opt` (the HEIR optimizer), rather than achieving a fully end-to-end FHE compilation to an executable in this early stage.

🧠 **Important:** While we'll build the HEIR compiler, its full end-to-end FHE compilation capabilities are still under active development. Expect to work with intermediate representations and tools like `heir-opt` to apply FHE-specific optimizations, rather than directly compiling to a final FHE-encrypted executable just yet.

Here's a simplified visual of the HEIR build process, showing how these components interact:



Step-by-Step Implementation: Preparing Your Environment

Let's begin by ensuring your system has all the necessary tools. We'll focus on Linux (Ubuntu/Debian-based distributions) and macOS, as these are common environments for compiler development.

Step 1: Install Git (Version Control)

Git is essential for cloning the HEIR and LLVM repositories from GitHub.

Why Git? It allows us to download the source code, track changes, and manage updates easily from the project's official repository.

- **For Ubuntu/Debian (or similar Linux distributions):**

```
    sudo apt update
    sudo apt install git -y
```

- **For macOS:** Git is usually pre-installed with Xcode Command Line Tools. For a more up-to-date version or if you don't have Xcode, you can install it via Homebrew (recommended).

```
    /bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/
install/HEAD/install.sh)"
    brew install git
```

Verify Installation:

```
git --version
```

You should see an output like `git version 2.45.2` (or newer, as of 2026-08-18).

Step 2: Install CMake (Build System Generator)

CMake is crucial for generating the build files that will compile HEIR. We'll aim for a recent stable version.

Why CMake? It abstracts the build process, making it consistent across different operating systems and compiler toolchains, simplifying complex project setups like HEIR's.

- **For Ubuntu/Debian:**

```
    sudo apt install cmake -y
```

```
⚡ **Quick Note:** The `apt` package manager might provide an older stable
version. While generally sufficient for HEIR, for the absolute latest,
consider downloading from the official CMake website or building from source
if you encounter specific compatibility issues.
```

- **For macOS:**

```
brew install cmake
```

Verify Installation:

```
cmake --version
```

You should see `cmake version 3.29.3` (or newer, as of 2026-08-18).

Step 3: Install a C++ Compiler (Clang or GCC)

HEIR and LLVM are predominantly written in C++, so you'll need a robust C++ compiler. Clang is often preferred for LLVM development due to its tight integration, but GCC works perfectly fine. We recommend a C++17 compatible compiler or newer.

Why a C++ Compiler? It translates the C++ source code into executable machine instructions, making the compiler tools runnable on your system.

- **For Ubuntu/Debian:** Install `build-essential`, which includes GCC and G++.

```
sudo apt install build-essential -y
```

```
To specifically install Clang and related tools (often newer than GCC on some
systems):
```

```
sudo apt install clang lld -y
```

- **For macOS:** Install Xcode Command Line Tools. This provides Clang and other essential development utilities.

```
xcode-select --install
```

Verify Installation:

```
g++ --version
clang --version
```

You should see versions like `g++ (Ubuntu 11.4.0-1ubuntu1~22.04) 11.4.0` or `Apple clang version 15.0.0 (clang-1500.3.9.4)` (or newer, as of 2026-08-18).

Step 4: Install Ninja (Fast Build System)

Ninja is a small, fast build system that significantly speeds up compilation, especially for large projects like LLVM and HEIR. It's often much faster than traditional `make`.

Why Ninja? It's designed for speed, minimizing overhead in the build process by focusing on parallelism and efficient dependency tracking.

- **For Ubuntu/Debian:**

```
sudo apt install ninja-build -y
```

- **For macOS:**

```
brew install ninja
```

Verify Installation:

```
ninja --version
```

You should see `1.11.1` (or newer, as of 2026-08-18).

Step 5: Install Python (for LLVM/MLIR Scripts)

Python is used for various utility scripts within the LLVM/MLIR ecosystem, including build automation and testing.

Why Python? Many build and testing scripts, as well as some parts of the LLVM infrastructure, are written in Python. Having it installed ensures these utilities function correctly.

- **For Ubuntu/Debian:**

```
sudo apt install python3 python3-pip -y
```

- **For macOS:** Python 3 is usually pre-installed. You can also install it via Homebrew for a managed installation:

```
    brew install python
```

Verify Installation:

```
python3 --version
```

You should see `Python 3.10.12` (or newer, as of 2026-08-18).

Step-by-Step Implementation: Building HEIR

Now that your system is prepared, let's get HEIR compiled. This process involves cloning two repositories: `llvm-project` (which contains MLIR) and `HEIR`. We'll build `llvm-project` first, specifically MLIR, then use its output to compile HEIR.

Step 1: Clone the LLVM Project Repository

We need the `llvm-project` repository to build MLIR, which HEIR depends on.

```
# Create a dedicated directory for your HEIR development
mkdir heir_dev && cd heir_dev

# Clone the LLVM project. This is a very large repository (several GB).
# It will take some time depending on your internet connection.
git clone https://github.com/llvm/llvm-project.git
```

This command downloads the entire LLVM project source code. Be patient!

Step 2: Build MLIR from LLVM Project

Next, we'll configure and build MLIR within the `llvm-project`. This step is crucial as HEIR relies on the MLIR libraries and tools.

```
# Create a build directory for LLVM components, separate from the source
mkdir llvm-project/build && cd llvm-project/build

# Run CMake to configure the build.
# We explicitly enable MLIR and use Ninja as the generator.
cmake -GNinja -DLLVM_ENABLE_PROJECTS=mlir -DLLVM_TARGETS_TO_BUILD="X86;AArch64
" -DCMAKE_BUILD_TYPE=Release ..
```

Let's break down these important CMake flags:

- `-GNinja`: Specifies Ninja as the build system. This is generally faster for large projects.

- `-DLLVM_ENABLE_PROJECTS=mlir`: Tells CMake to build only the MLIR project from the vast `llvm-project` repository. This saves a lot of compilation time and disk space.
- `-DLLVM_TARGETS_TO_BUILD="X86;AArch64"`: Specifies which target architectures to build. `X86` is generally sufficient for most development on desktop/laptop systems. `AArch64` is for ARM-based systems like Apple Silicon Macs. You can adjust this based on your specific needs.
- `-DCMAKE_BUILD_TYPE=Release`: Builds in release mode, which optimizes for performance. This is ideal for a compiler you intend to use.

⚠️ **What can go wrong:** If CMake fails here, carefully review the error messages. Common issues include missing CMake or C++ compiler installations, or an incorrect path if you've deviated from the `mkdir build && cd build` pattern.

After CMake finishes configuring (which might take a minute or two), compile MLIR:

```
ninja
```

This step will take a significant amount of time (potentially 30 minutes to an hour or more, depending on your system's specs) and consume considerable CPU and memory resources, as it compiles the entire MLIR framework. This is a good time to grab a coffee, or two!

Step 3: Clone the HEIR Repository

Once MLIR is built, navigate back to your `heir_dev` directory and clone the HEIR repository.

```
# Go back to the parent directory (heir_dev)
cd ../..

# Clone the HEIR compiler repository
git clone https://github.com/heir-compiler/HEIR.git
```

Step 4: Build HEIR

Now we'll build HEIR, linking it against the MLIR we just compiled. This step is critical for HEIR to find its necessary MLIR dependencies.

```
# Create a build directory for HEIR, separate from its source
mkdir HEIR/build && cd HEIR/build

# Run CMake to configure HEIR.
# CRITICAL: We need to tell HEIR where to find the MLIR build.
```

```
# The `MLIR_DIR` variable points to the `lib/cmake/mlir` directory within your
LLVM build.
# Adjust the path carefully if your directory structure is different.
cmake -GNinja -DMLIR_DIR=$(pwd)/../../llvm-project/build/lib/cmake/mlir ..
```

Let's look at the crucial `MLIR_DIR` flag:

- `-DMLIR_DIR=$(pwd)/../../llvm-project/build/lib/cmake/mlir`: This is the most vital part of the HEIR CMake command. It explicitly tells HEIR's CMake where to find the MLIR configuration files and libraries that were generated during the `llvm-project` build. The `$(pwd)/../../llvm-project/build/lib/cmake/mlir` path assumes `HEIR/build` and `llvm-project/build` are located as siblings within your `heir_dev` directory, as set up in our steps.
  - `$(pwd)` is your current directory (e.g., `~/heir_dev/HEIR/build`).
  - `..` goes up one level to `~/heir_dev/HEIR`.
  - `..` again goes up one level to `~/heir_dev`.
  - Then `/llvm-project/build/lib/cmake/mlir` points to the specific MLIR installation directory.

🧠 **Important:** Carefully verify the `MLIR_DIR` path. An incorrect path is the most common reason for HEIR's CMake configuration to fail. If you've placed `llvm-project` or `HEIR` in different locations, you'll need to adjust this path accordingly.

After CMake successfully configures, compile HEIR:

```
ninja
```

This compilation should be much faster than the MLIR build, as HEIR is a smaller project that builds on top of the already compiled MLIR.

Step 5: Verify Your HEIR Installation

After `ninja` completes, you should have the HEIR tools built and ready. The primary tool we'll look for is `heir-opt`, the HEIR optimizer.

```
# Check if heir-opt exists in the build/bin directory
ls bin/heir-opt
```

If you see `bin/heir-opt` listed, congratulations! You've successfully built the HEIR compiler. You can also try running it to confirm its basic functionality:

```
./bin/heir-opt --help
```

This command should print a list of available command-line options and passes for `heir-opt`, indicating that the executable is functional.

⚡ **Quick Note:** Remember the comment from the HEIR GitHub repository: "If you require an executable, please use `format_assistant/h`." This implies that direct, end-to-end compilation to a final FHE executable is not fully stable or streamlined yet. Your `heir-opt` tool allows you to apply HEIR's MLIR-based optimizations to FHE-specific intermediate representations, which is the core functionality you've just built and will be the focus of our initial explorations.

Mini-Challenge: Optimizing a Simple MLIR Fragment with HEIR

Let's put your newly built `heir-opt` to a small test. We'll create a very basic MLIR file and try to run `heir-opt` over it. While HEIR's full power comes with FHE-specific dialects, `heir-opt` can still process generic MLIR, allowing us to confirm the build's integrity.

Challenge:

1. Create a simple MLIR file named `simple.mlir` that defines a basic function.
2. Use your `heir-opt` executable to process this file and observe the output.

Instructions: First, create the `simple.mlir` file. You can place it directly in your `HEIR/build` directory, or in the parent `HEIR` directory (then adjust the path in the command).

```
// simple.mlir
func.func @my_simple_func(%arg0: i32) -> i32 {
  %0 = arith.addi %arg0, %arg0 : i32
  func.return %0 : i32
}
```

Now, run `heir-opt` on it. You can specify a pass to apply, or just let it print the IR by default, which confirms parsing and basic processing.

```
# Make sure you are in the HEIR/build directory to easily access ./bin/heir-
opt
# If not, adjust the path, e.g., ~/heir_dev/HEIR/build/bin/heir-opt
./bin/heir-opt ../simple.mlir
```

What to Observe/Learn:

- `heir-opt` should print the MLIR code, possibly with some default passes applied. You should see the content of your `simple.mlir` file echoed back, potentially with minor formatting changes.
- The primary goal here is to confirm that `heir-opt` can successfully parse and process an MLIR file without crashing. This confirms your basic HEIR build is functional and can interpret MLIR.
- In future chapters, we'll explore how to apply HEIR's specific FHE-related passes using flags like `--heir-some-fhe-pass`. For now, just confirming basic execution is a big step!

Common Pitfalls & Troubleshooting

Building complex software like a compiler can sometimes hit snags. Here are some common issues you might encounter and practical strategies to approach them:

- **"CMake Error: The source directory ... does not appear to contain CMakeLists.txt."**
  - **Problem:** You're running `cmake` from the wrong directory or pointing it to an incorrect source directory.
  - **Solution:** Ensure you are in the correct `build` directory (e.g., `heir_dev/llvm-project/build` or `heir_dev/HEIR/build`) and that `cmake ..` correctly points to the parent directory containing the `CMakeLists.txt` file for the project you're trying to build.
- **"Could not find a package configuration file provided by 'MLIR'..."**
  - **Problem:** HEIR's CMake couldn't find the MLIR installation. This is almost always due to an incorrect `DMLIR_DIR` path.
  - **Solution:** Double-check the `DMLIR_DIR` argument in your HEIR CMake command. Make sure it points exactly to `$(YOUR_LLVM_BUILD_DIR)/lib/cmake/mlir`. The `$(pwd)` trick is sensitive to your current working directory. If in doubt, use an absolute path for `MLIR_DIR`.

- **"fatal error: 'mlir/IR/BuiltinOps.h' file not found" (or similar header errors)**
    - **Problem:** The C++ compiler cannot locate the MLIR header files. This usually means MLIR wasn't built correctly, or the `MLIR_DIR` path is still off, preventing HEIR from finding its dependencies.
    - **Solution:** Revisit the MLIR build steps (Step 2 of "Building HEIR"). Ensure `ninja` completed successfully in `llvm-project/build` and that no errors were reported. If it did, then focus again on the `DMLIR_DIR` path for HEIR's CMake.
- **Compilation failures during `ninja` (many C++ errors)**
    - **Problem:** This can be due to an incompatible C++ compiler version, missing development libraries, or insufficient system resources (memory, CPU).
    - **Solution:**
        - Ensure your C++ compiler (GCC/Clang) is up-to-date and supports C++17 or newer.
        - Verify you have enough RAM and CPU cores. Building LLVM/MLIR is very memory-intensive; a machine with less than 8GB RAM might struggle, and 16GB+ is recommended.
        - Sometimes, cleaning the build directory (`rm -rf *` within the `build` folder) and re-running CMake and Ninja can resolve transient issues caused by corrupted build artifacts.
- **Disk Space Issues:**
    - **Problem:** The `llvm-project` repository and its build artifacts can consume a significant amount of disk space (tens of gigabytes).
    - **Solution:** Ensure you have ample free disk space (at least 50GB recommended) before starting the build. If you run out, `ninja` will fail.

Summary

Phew! You've successfully navigated the intricate process of setting up your HEIR development environment. This is a significant accomplishment in your journey toward understanding private AI. Let's quickly recap what we've achieved:

- **Understood HEIR's Architecture:** We clarified HEIR's deep reliance on the LLVM and MLIR frameworks as its foundational compiler infrastructure.

- **Installed Core Prerequisites:** You've equipped your system with essential development tools including Git, CMake, Ninja, a C++ compiler (Clang/GCC), and Python.
- **Built MLIR:** You successfully cloned the `llvm-project` and compiled the MLIR framework, which is a critical dependency for HEIR.
- **Compiled HEIR:** By correctly pointing HEIR's build system to your MLIR installation, you compiled the HEIR compiler components, including the `heir-opt` tool.
- **Verified Installation:** You confirmed the presence and basic functionality of `heir-opt` with a mini-challenge, ensuring your build is operational.
- **Identified Common Pitfalls:** You're now aware of potential build issues and equipped with troubleshooting strategies, making future development smoother.

You now have a functional HEIR environment, which is a significant achievement given the complexity of compiler development. This setup provides the foundation for exploring how HEIR processes and optimizes FHE programs, bringing us closer to practical private AI inference.

In the next chapter, we'll dive deeper into HEIR's specific MLIR dialects and how they represent FHE computations. We'll start to see how HEIR bridges the gap between high-level programming and privacy-preserving encrypted operations, enabling you to build truly private AI applications.

This page is AI-assisted and reviewed. It references official documentation and recognized resources where relevant.

References

- **HEIR GitHub Repository:** The primary source for HEIR's source code and development status.
    - https://github.com/heir-compiler/HEIR
- **LLVM Project:** Official documentation and source for the LLVM compiler infrastructure.
    - https://llvm.org/
- **MLIR Documentation:** Information on the Multi-Level Intermediate Representation framework.
    - https://mlir.llvm.org/
- **CMake Official Documentation:** Comprehensive guide for using the CMake build system.
    - https://cmake.org/documentation/
- **Ninja Build System:** Details on the fast build system used for LLVM and HEIR.
    - https://ninja-build.org/

CHAPTER 05

HEIR's Inner Workings: Understanding Its Architecture and Intermediate Representation

Introduction to HEIR's Compiler Architecture

Welcome back! In our previous chapters, we explored the fascinating world of Homomorphic Encryption (HE) and Fully Homomorphic Encryption (FHE), understanding how they empower us to perform computations directly on encrypted data. We also introduced HEIR (Homomorphic Encryption Intermediate Representation), Google's ambitious open-source project designed to be an end-to-end FHE compiler.

But how does HEIR actually do this magic? How does it take a standard program and transform it into one that can run securely on encrypted data, preserving privacy? This chapter will pull back the curtain, taking you on a journey into the heart of HEIR: its compiler architecture and the crucial role of its Intermediate Representation (IR).

Understanding HEIR's architecture is key to grasping its power and potential. It helps us see how complex FHE operations are managed, optimized, and eventually executed. By the end of this chapter, you'll have a clear mental model of how HEIR translates your privacy-preserving AI ideas into reality.

The Compiler's Lens: Deconstructing HEIR

At its core, HEIR is a compiler. Just like a C++ compiler translates your human-readable code into machine instructions, HEIR translates your high-level computation logic into a form executable by a homomorphic encryption scheme. This translation isn't simple; it involves several intricate stages.

What is a Compiler Architecture?

Think of a traditional compiler as a factory with specialized departments. Each department takes the output from the previous one, performs a specific task, and then passes it along.

Generally, compilers are divided into three main phases:

1. **Frontend:** This department understands your original source code (e.g., Python, C++). It checks for syntax errors, builds an abstract syntax tree (AST), and often converts the code into a generic, high-level Intermediate Representation (IR).
2. **Middle-End:** This is the optimization hub. It takes the IR from the frontend and performs various transformations to make the code faster, more efficient, or smaller, without changing its core logic.
3. **Backend:** This final department takes the optimized IR and generates the actual machine code or instructions for a specific target platform (e.g., x86, ARM).

HEIR's Specialized Compiler Stages

HEIR follows a similar, yet specialized, three-stage pipeline, tailored specifically for the unique challenges of Fully Homomorphic Encryption.

Frontend: Bridging to FHE

The HEIR frontend's job is to take a program written in a standard language or a more abstract computation graph and convert it into HEIR's FHE-aware Intermediate Representation. This initial step is where the compiler begins to understand your program's intent and identify operations that can be performed homomorphically.

Middle-End: FHE-Specific Optimizations

This is where the real FHE magic happens in terms of optimization. Homomorphic encryption schemes have specific computational costs and limitations (e.g., noise growth, bootstrapping). The HEIR middle-end is designed to apply FHE-specific transformations to the IR. This might include:

- **Operation Rewriting:** Replacing standard arithmetic operations with their FHE-compatible equivalents (e.g., turning a regular `add` into an `fhe.add`).
- **Noise Management:** Optimizing the order of operations to minimize noise accumulation, which is critical for FHE to maintain data integrity over many computations.
- **Bootstrapping Strategy:** Deciding when and how to perform bootstrapping (a noise-reduction technique) to enable deeper computations without losing precision.

- **Circuit Optimization:** Restructuring the computation to be more efficient for encrypted execution, potentially by reordering operations or combining them.

Backend: Targeting Cryptosystems

The HEIR backend is responsible for translating the optimized FHE-IR into concrete operations for a specific homomorphic encryption library or cryptosystem. This could involve generating code for libraries like SEAL, TFHE, or OpenFHE, ensuring that the operations are correctly implemented according to the chosen scheme's parameters and API.

📌 **Key Idea:** HEIR acts as an abstraction layer, allowing developers to write high-level code while handling the complex FHE-specific transformations and optimizations behind the scenes.

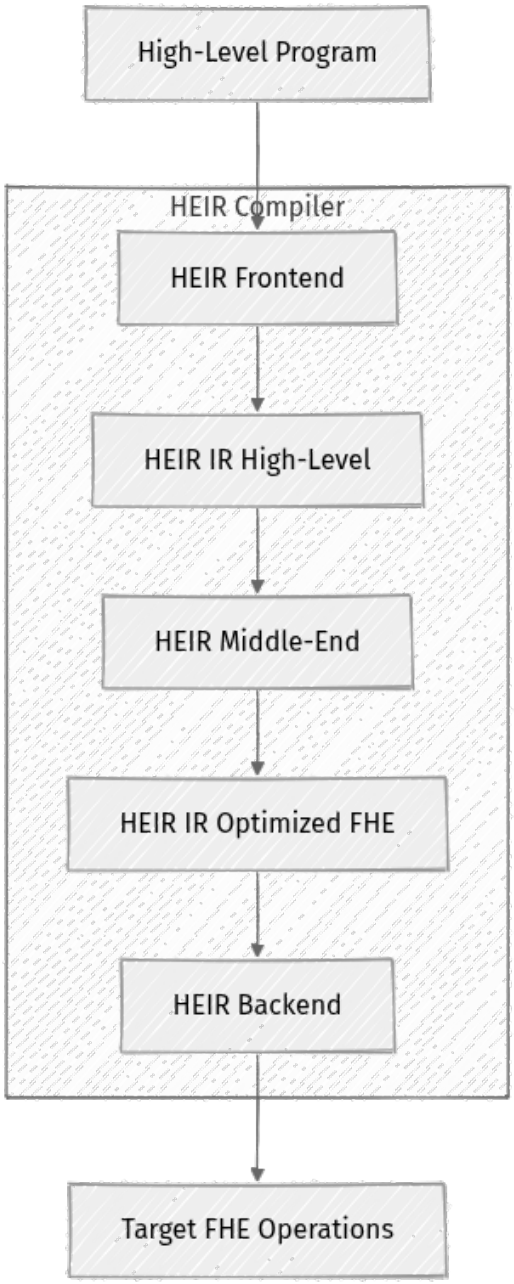


Figure: Simplified HEIR Compiler Pipeline

⚡ **Quick Note:** As of 2026-08-18, the HEIR project is actively under development. The GitHub repository notes that the "integration between Middle-End and Back-End is not yet well-implemented." This means that while the conceptual architecture is clear, practical end-to-end compilation to an executable FHE program might require using specific tools like `format_assistant/h` for certain stages, rather than a fully integrated pipeline. This is a common characteristic of cutting-edge research projects.

Intermediate Representation (IR): The Universal Translator

The concept of an Intermediate Representation (IR) is central to any sophisticated compiler, and especially for HEIR.

What is an IR?

An IR is an abstract form of your program that sits between the original source code and the final machine code. It's like a blueprint that describes the computation without being tied to a specific programming language or a specific hardware architecture.

Why is this useful?

- **Abstraction:** It frees the middle-end from worrying about the quirks of many different source languages and the backend from dealing with many different hardware architectures.
- **Optimization Target:** All optimizations can be applied to this single, unified representation, rather than having to write separate optimizers for each source language or target platform.
- **Modularity:** Different frontends can target the same IR, and different backends can consume the same IR.

HEIR's FHE-Aware IR

HEIR's Intermediate Representation is particularly interesting because it's designed to represent computations in a way that is aware of homomorphic encryption constraints. It's built upon the MLIR (Multi-Level Intermediate Representation) framework, which provides a flexible way to define different levels of abstraction for an IR.

This allows HEIR's IR to:

- **Represent FHE Primitives:** It can express operations like `add_encrypted`, `multiply_encrypted`, `rotate_encrypted_vector`, and `bootstrap` as first-class citizens.
- **Track FHE-Specific Properties:** The IR can carry metadata about encrypted values, such as their noise level, the encryption context, or the underlying plaintext type. This is crucial for the middle-end to make informed optimization decisions.

- **Enable Progressive Lowering:** MLIR's multi-level nature means the IR can start at a high, abstract level (e.g., "perform matrix multiplication") and gradually be "lowered" through successive transformations into more concrete, FHE-specific operations (e.g., "perform `dot_product` operations on encrypted vectors, followed by a `bootstrap`").

Conceptual Example: From High-Level to FHE-IR

Let's imagine a simple function that adds two numbers. In a high-level language, it's straightforward:

```
def add_numbers(a, b):
    return a + b
```

When this goes through HEIR, the IR might evolve:

Stage 1: High-Level IR (Conceptual)

Initially, the HEIR frontend might convert this into an IR that looks very similar to the original operation, but within the HEIR framework:

```
// Represents the initial operation
func @add_numbers(%arg0: i32, %arg1: i32) -> i32 {
  %0 = "std.add"(%arg0, %arg1) : (i32, i32) -> i32
  "std.return"(%0) : (i32) -> ()
}
```

This is a simplified, non-FHE specific representation, similar to what a general compiler might produce. It uses `i32` for 32-bit integers.

Stage 2: FHE-Aware IR (Conceptual)

Now, the HEIR middle-end steps in. If `arg0` and `arg1` are determined to be encrypted inputs, the middle-end would transform the `std.add` operation into an FHE-specific addition. It might also add annotations about the encryption context.

```
// After FHE lowering, assuming inputs are encrypted
func @add_numbers(%arg0: !fhe.ciphertext<i32>, %arg1: !fhe.ciphertext<i32>)
-> !fhe.ciphertext<i32> {
  // Use an FHE-specific add operation
  %0 = "fhe.add"(%arg0, %arg1) : (!fhe.ciphertext<i32>, !fhe.ciphertext<i32>)
-> !fhe.ciphertext<i32>
  // Potentially insert bootstrapping if noise is too high, or other
optimizations
  // %1 = "fhe.bootstrap"(%0) : (!fhe.ciphertext<i32>) -> !fhe.ciphertext<i32>
  "fhe.return"(%0) : (!fhe.ciphertext<i32>) -> ()
}
```

Notice how the type signature changes to `!fhe.ciphertext<i32>` (indicating an encrypted 32-bit integer) and the operation itself becomes `fhe.add`. This is a conceptual example of how HEIR's IR can represent FHE operations and track encrypted types.

🧠 **Important:** The actual syntax of HEIR's MLIR-based IR is much more detailed and complex, involving specific dialects for various FHE operations and parameters. This example is highly simplified to illustrate the concept of transformation.

Step-by-Step: Defining a Computation for HEIR's IR

While HEIR is a compiler, for this chapter focused on architecture and IR, we'll explore how you would define a simple FHE computation using its conceptual Intermediate Representation. This is how you'd express your privacy-preserving logic in a way that HEIR can understand and process.

Let's define a simple computation: multiplying an encrypted number by a plaintext constant, then adding an encrypted number. This illustrates both FHE-specific operations and interactions with unencrypted data.

Step 1: Declare the Function and Input Types

First, we need to declare a function within the HEIR IR. We'll specify its name, arguments, and return type. For encrypted values, we use the `!fhe.ciphertext` type. For plaintext constants, we'll use a standard integer type like `i32`.

Imagine a function `compute_private_value` that takes two encrypted integers (`%encrypted_x`, `%encrypted_y`) and a plaintext integer (`%constant`). It will return an encrypted integer.

```
// Define a function that takes two encrypted inputs and one plaintext
constant
func @compute_private_value(
    %encrypted_x: !fhe.ciphertext<i32>,
    %encrypted_y: !fhe.ciphertext<i32>,
    %constant: i32
) -> !fhe.ciphertext<i32> {
    // Operations will go here
}
```

Here, `func` declares a function, `@compute_private_value` is its name, and the arguments are explicitly typed. The `-> !fhe.ciphertext<i32>` indicates the return type.

Step 2: Perform Encrypted Multiplication with a Plaintext Constant

One common FHE operation is multiplying an encrypted value by a known, unencrypted constant. This operation is typically much faster and doesn't increase noise as much as encrypted-encrypted multiplication. HEIR's IR would represent this with a specific FHE operation, perhaps `fhe.mul_scalar`.

Let's multiply `%encrypted_x` by `%constant`.

```
// ... (previous function declaration) ...
func @compute_private_value(
    %encrypted_x: !fhe.ciphertext<i32>,
    %encrypted_y: !fhe.ciphertext<i32>,
    %constant: i32
) -> !fhe.ciphertext<i32> {
    // Multiply encrypted_x by the plaintext constant
    %multiplied_val = "fhe.mul_scalar"(%encrypted_x, %constant) : (!
fhe.ciphertext<i32>, i32) -> !fhe.ciphertext<i32>
    // ... next operation ...
}
```

Here, `%multiplied_val` is a temporary variable holding the result. The `"fhe.mul_scalar"` operation explicitly states that it takes an encrypted integer and a plaintext integer, and produces an encrypted integer.

Step 3: Perform Encrypted Addition

Next, we want to add the result of our multiplication (`%multiplied_val`) to our second encrypted input (`%encrypted_y`). This is a standard encrypted-encrypted addition.

```
// ... (previous function declaration and multiplication) ...
func @compute_private_value(
    %encrypted_x: !fhe.ciphertext<i32>,
    %encrypted_y: !fhe.ciphertext<i32>,
    %constant: i32
) -> !fhe.ciphertext<i32> {
    %multiplied_val = "fhe.mul_scalar"(%encrypted_x, %constant) : (!
fhe.ciphertext<i32>, i32) -> !fhe.ciphertext<i32>

    // Add the multiplied value to the second encrypted input
    %final_result = "fhe.add"(%multiplied_val, %encrypted_y) : (!
fhe.ciphertext<i32>, !fhe.ciphertext<i32>) -> !fhe.ciphertext<i32>
    // ... return value ...
}
```

The `"fhe.add"` operation takes two encrypted integers and returns an encrypted integer. This is a common FHE primitive.

Step 4: Return the Final Encrypted Value

Finally, the function needs to return the computed encrypted value.

```
// ... (previous operations) ...
func @compute_private_value(
    %encrypted_x: !fhe.ciphertext<i32>,
    %encrypted_y: !fhe.ciphertext<i32>,
    %constant: i32
) -> !fhe.ciphertext<i32> {
    %multiplied_val = "fhe.mul_scalar"(%encrypted_x, %constant) : (!
fhe.ciphertext<i32>, i32) -> !fhe.ciphertext<i32>
    %final_result = "fhe.add"(%multiplied_val, %encrypted_y) : (!
fhe.ciphertext<i32>, !fhe.ciphertext<i32>) -> !fhe.ciphertext<i32>

    // Return the final encrypted result
    "fhe.return"(%final_result) : (!fhe.ciphertext<i32>) -> ()
}
```

The `"fhe.return"` operation signifies the end of the FHE computation within this function.

⚡ **Real-world insight:** This textual IR is what HEIR's middle-end would consume and optimize. While you wouldn't typically write this by hand for complex programs, understanding its structure is crucial for debugging and understanding how HEIR processes your FHE logic. Tools like `mlir-opt` (part of the MLIR ecosystem) can be used to apply passes and observe IR transformations.

Mini-Challenge: Tracing an Encrypted Multiplication

Let's test your understanding of how an operation might evolve within HEIR.

Challenge: Imagine you have a high-level function `multiply_numbers(x, y)` that returns `x * y`. If both `x` and `y` are encrypted inputs, describe conceptually how the `multiply` operation might be represented in:

1. The initial, high-level IR stage (similar to `std.add`).
2. The FHE-aware IR stage, considering it's an encrypted multiplication.

Think about the types involved and the operation name. You don't need to write perfect MLIR syntax, just explain the conceptual changes.

Hint: Focus on the type annotations and the operation name in your conceptual IR snippets. How would they reflect the encrypted nature of the data? Remember the `!fhe.ciphertext` type and the FHE-specific operation prefix.

Common Pitfalls & Troubleshooting in Early FHE Compilers

Working with cutting-edge tools like HEIR, especially in its active development phase, comes with its own set of challenges.

1. **Navigating Development Stage Limitations:** As mentioned, HEIR is evolving. You might encounter features that are not fully integrated or documented. Expect to consult the source code and GitHub issues more frequently than with mature tools. The current integration between the middle-end and backend is still being built out (as of 2026-08-18).
    - **Troubleshooting:** Always check the latest `README.md` on the HEIR GitHub for updates on feature completeness and recommended workflows. Join any community forums or mailing lists if available.
2. **Complexity of FHE Operations:** Even with a compiler, understanding the underlying FHE operations and their constraints (like noise growth and bootstrapping) is crucial for designing efficient private AI applications. If your program doesn't compile or performs poorly, it might be due to FHE limitations rather than a compiler bug.
    - **Troubleshooting:** Review the basics of FHE scheme limitations (e.g., multiplicative depth, bootstrapping cost). Simplify your computation to isolate the problematic part. Consider if your algorithm is inherently FHE-friendly.
3. **Debugging Encrypted Computations:** Debugging a program that runs on encrypted data is inherently difficult because you cannot directly inspect intermediate values. Errors might manifest as incorrect final results or performance bottlenecks.
    - **Troubleshooting:** Start with small, unencrypted test cases to verify logic. Gradually introduce encryption, comparing results. HEIR's IR can be inspected to understand the transformations, which is a powerful debugging tool for compiler-level issues. Look for tools or passes within HEIR that allow for "plaintext evaluation" during development.

Summary

In this chapter, we've taken a deep dive into the internal architecture of HEIR, Google's FHE compiler, and demystified the concept of Intermediate Representation.

Here are the key takeaways:

- HEIR functions as a specialized compiler with a **Frontend**, **Middle-End**, and **Backend**, each tailored for FHE.
- The **Frontend** translates high-level programs into an initial IR, understanding the program's intent.
- The **Middle-End** applies crucial FHE-specific optimizations, such as noise management, bootstrapping strategy, and circuit restructuring.
- The **Backend** targets specific FHE cryptosystems, generating executable operations for libraries like SEAL or OpenFHE.
- **Intermediate Representation (IR)** is the core abstraction, acting as a universal blueprint for the program.
- HEIR's IR, built on MLIR, is **FHE-aware**, representing encrypted types (`!fhe.ciphertext`) and FHE-specific operations (e.g., `fhe.add`, `fhe.mul_scalar`), and enabling progressive lowering.
- We conceptually walked through defining a simple FHE computation directly in HEIR's IR, illustrating how operations like encrypted addition and scalar multiplication are expressed.
- Working with HEIR currently requires acknowledging its **active development status** and potential limitations in full end-to-end integration.

Understanding HEIR's architecture and IR is fundamental to leveraging its power for private AI. In the next chapter, we'll start getting our hands dirty with setting up the HEIR environment and exploring how to define simple computations for FHE in a more practical context.

References

- HEIR Compiler GitHub Repository
- A Guide For HEIR Experiment Evaluation (Original Version)
- MLIR (Multi-Level Intermediate Representation) Official Website

This page is AI-assisted and reviewed. It references official documentation and recognized resources where relevant.

CHAPTER 06

Your First Encrypted Computation: A 'Hello World' with HEIR

Welcome back, future privacy architect! In our previous chapters, we explored the fascinating world of homomorphic encryption (HE) and fully homomorphic encryption (FHE), understanding their power to enable secure computation on encrypted data. We also introduced HEIR, an open-source compiler framework designed to make building FHE-powered applications more accessible.

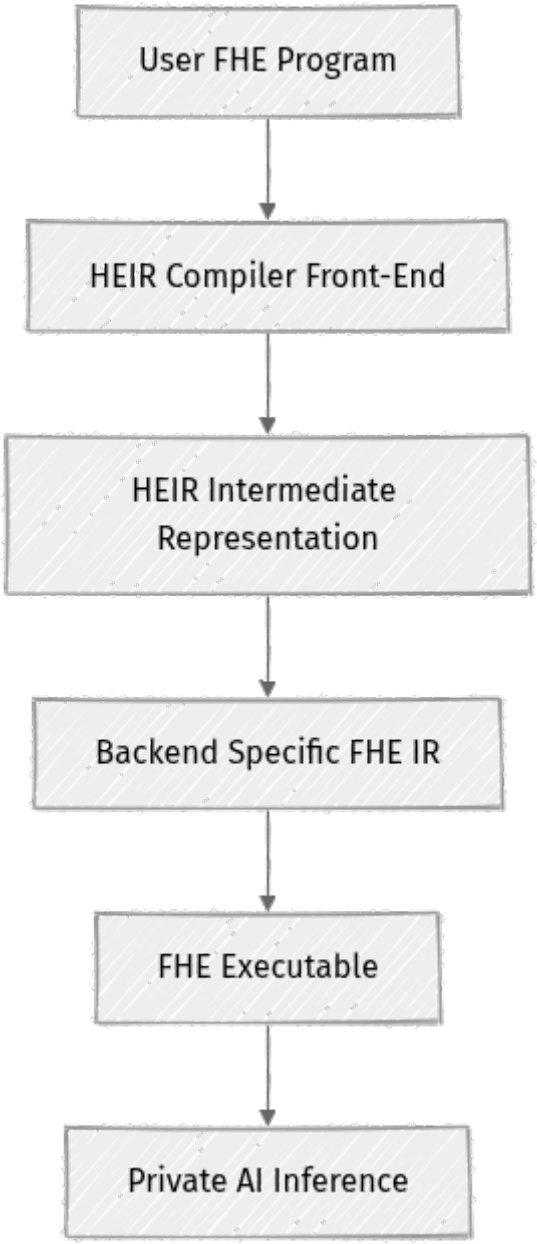
Today, we're rolling up our sleeves to perform our very first encrypted computation using HEIR. Think of this as our "Hello World" moment for private AI. While HEIR is still in active development, we'll set up a basic HEIR environment and walk through a simple, illustrative example to see FHE compilation in action. This hands-on experience will solidify your understanding of how HEIR translates abstract FHE principles into practical, privacy-preserving operations, even if the end-to-end execution aspects are still evolving.

HEIR's Role in the FHE Ecosystem

Before we dive into code, let's quickly recap HEIR's purpose. HEIR isn't an FHE library itself; rather, it's a compiler framework. This means it takes a description of a computation (often in an Intermediate Representation like MLIR), understands which parts need to operate on encrypted data, and then transforms that description into a form that can be executed by an FHE backend.

The goal is to streamline the complex process of FHE programming. Instead of manually handling low-level cryptographic operations, developers can define their desired computation, and HEIR handles the heavy lifting of compiling it into an FHE-compatible sequence. This greatly reduces the expertise required to build privacy-preserving applications.

Here's a simplified view of the HEIR compilation process:



📌 **Key Idea:** HEIR is a compiler framework, not a standalone FHE library. It bridges the gap between high-level computation descriptions and low-level FHE operations.

Setting Up Your HEIR Workbench

As of 2026-08-18, HEIR is an active research project, and its development is ongoing. This means specific stable releases with version numbers aren't widely published in the traditional sense. Instead, developers typically work with the `main` branch of the GitHub repository. It's an open-source project developed by a team of researchers and engineers.

Prerequisites: CMake

HEIR uses CMake to manage its build process. You'll need CMake installed on your system.

1. **Check CMake Version:** Open your terminal and run:

```
    cmake --version
```

```
You should see output similar to `cmake version 3.20.0` or newer. If CMake
isn't installed or is an older version, you'll need to install it. For Linux,
`sudo apt install cmake` (Debian/Ubuntu) or `sudo dnf install cmake` (Fedora)
usually works. For macOS, `brew install cmake` (with Homebrew) is common.

⚡ **Quick Note:** We recommend CMake version 3.20 or higher for best
compatibility with modern C++ projects.
```

Step 1: Clone the HEIR Repository

First, let's get the HEIR source code. We'll clone the official GitHub repository.

```
git clone https://github.com/heir-compiler/HEIR.git
```

This command downloads the entire HEIR project into a new directory named `HEIR` in your current location.

Step 2: Navigate and Create a Build Directory

Move into the newly cloned directory and create a `build` subdirectory. This is a standard practice for CMake projects to keep build artifacts separate from source code.

```
cd HEIR
mkdir build
cd build
```

Step 3: Configure and Build HEIR

Now, we'll use CMake to configure the build system and then compile HEIR.

1. **Configure with CMake:** From inside your `build` directory, run CMake:

```
    cmake ..
```

```
The `..` tells CMake to look for the `CMakeLists.txt` file in the parent
directory (the root of the HEIR repository). CMake will analyze your system
```

```
and generate build files (e.g., Makefiles on Linux/macOS).

⚠️ **What can go wrong:** If you encounter errors during this step, it's often due to missing dependencies (like specific C++ compilers, libraries, or Python packages). Check the HEIR GitHub repository's `README.md` for detailed dependency instructions if you run into issues.
```

1. **Build HEIR:** Once CMake configuration is successful, compile the project:

```
    cmake --build . --parallel $(nproc)
```

```
-   `cmake --build .` instructs CMake to build the project in the current directory.
-   `--parallel $(nproc)` (or `-j` followed by a number, e.g., `-j8`) tells the build system to use multiple CPU cores, which significantly speeds up compilation. `$(nproc)` automatically detects the number of available cores on Linux. On macOS, you might use `sysctl -n hw.ncpu` to get the core count.

This process can take a significant amount of time, depending on your system's specifications. Grab a coffee!

⚡ **Real-world insight:** Building complex compiler frameworks like HEIR from source is a common task in research and development environments. It ensures you have the latest features and can customize the build if needed.
```

Crafting Your First FHE Program: Adding a Constant

Since HEIR is a compiler, our "Hello World" won't be a simple `print()` statement. Instead, we'll define a very basic computation that HEIR can process. For this example, we'll define a function that takes an encrypted input `x` and adds a plaintext constant `5` to it, producing an encrypted output `x + 5`.

HEIR typically works with an Intermediate Representation (IR), often based on MLIR (Multi-Level IR). While writing full MLIR code can be complex, we can represent our simple operation conceptually.

Let's create a file named `add_constant.mlir` in the root `HEIR` directory (or a sub-directory you create for examples).

```
// add_constant.mlir
func.func @add_constant(%arg0: tensor<1xi32>) -> tensor<1xi32> {
  // Define a constant value to add
  %c5 = arith.constant 5 : i32
  %c5_tensor = tensor.from_elements %c5 : tensor<1xi32>

  // Perform the addition on the encrypted input and the constant
  %0 = arith.addi %arg0, %c5_tensor : tensor<1xi32>

  // Return the encrypted result
```

```
  func.return %0 : tensor<1xi32>
}
```

Let's break down this conceptual MLIR snippet:

- `func.func @add_constant(...)`: This declares a function named `add_constant`.
- `%arg0: tensor<1xi32>`: This specifies that our function takes one argument, `%arg0`, which is conceptually a 1-element tensor of 32-bit integers. In an FHE context, this `arg0` would represent our encrypted input.
- `-> tensor<1xi32>`: The function returns a 1-element tensor of 32-bit integers, representing our encrypted output.
- `%c5 = arith.constant 5 : i32`: We define a constant integer value `5`.
- `%c5_tensor = tensor.from_elements %c5 : tensor<1xi32>`: We convert our constant `5` into a 1-element tensor, matching the input type for our `addi` operation.
- `%0 = arith.addi %arg0, %c5_tensor : tensor<1xi32>`: This is the core operation. `arith.addi` performs an integer addition. It adds our encrypted input (`%arg0`) to our plaintext constant (`%c5_tensor`). Crucially, HEIR's job is to ensure this addition happens correctly while both are encrypted (or one is plaintext but handled securely by the FHE scheme).
- `func.return %0`: The function returns the result of the addition.

🧠 **Important:** This MLIR snippet is a simplified representation. In a real HEIR workflow, there would be more specific FHE-related operations and types to explicitly denote encrypted values and operations. The purpose here is to illustrate the computation HEIR would process.

Compiling with HEIR's `heir-opt` Utility

As noted in the HEIR GitHub README, the integration between the middle-end and back-end for generating full executables is still evolving. For practical experimentation, HEIR provides a helper utility: `heir-opt`. This tool helps process and prepare FHE programs by applying various optimization and transformation passes.

From the `HEIR/build` directory, you can invoke HEIR's tools. Let's process our `add_constant.mlir` through a simplified HEIR pipeline.

First, ensure you are in the `HEIR/build` directory.

```
cd path/to/HEIR/build
```

Now, let's use the HEIR compiler to process our MLIR file.

```
./bin/heir-opt ../add_constant.mlir --heir-simplify-arith -o add_constant_proc
essed.mlir
```

Let's break this down:

- `./bin/heir-opt`: This is the HEIR MLIR optimizer utility we just built.
- `../add_constant.mlir`: This specifies our input file. The `../` is because we're running it from the `build` directory.
- `--heir-simplify-arith`: This is an example of a pass HEIR might apply. It simplifies arithmetic operations. HEIR has many such passes that transform and optimize the IR for FHE.
- `-o add_constant_processed.mlir`: This specifies the output file where the processed MLIR will be written.

If successful, you will find a new file `add_constant_processed.mlir` in your `build` directory. Its content might look very similar to your input for such a simple pass, but in more complex scenarios, you'd see significant transformations as HEIR prepares the program for an FHE backend.

Regarding actual execution and the `format_assistant/h` mention in the README: The HEIR project currently focuses on the compilation framework and intermediate transformations. The `format_assistant/h` reference implies a specific path or tool to get an executable if one is available or desired for a particular FHE backend. As the backend integration is in flux, direct end-to-end execution from a simple MLIR file to a numeric result is not yet a ready-to-use feature for a "Hello World" in the traditional sense.

Instead, the "Hello World" here is successfully compiling and transforming your FHE program's representation using HEIR's tooling. This demonstrates that HEIR can parse, understand, and apply transformations to your FHE-aware code.

🔥 **Optimization / Pro tip:** Understanding the output of `heir-opt` with different passes is key to debugging and optimizing your FHE programs. Each pass performs a specific transformation, moving the program closer to an FHE-compatible form.

Mini-Challenge: Double the Constant!

Let's make a small modification to our `add_constant.mlir` file.

Challenge: Modify `add_constant.mlir` to add `10` instead of `5` to the encrypted input. Then, re-run the `heir-opt` command to process the updated file.

Hint: Look for the `arith.constant` operation in your `add_constant.mlir` file and change the literal value.

What to observe/learn: This challenge reinforces your understanding of how to define literal values in the MLIR-like syntax and how HEIR's `heir-opt` tool processes your changes. It shows the iterative nature of compiler development – defining your program, compiling, and checking the output.

Common Pitfalls & Troubleshooting

1. **CMake Configuration Errors:**
    - **Issue:** `CMake Error at CMakeLists.txt:X (find_package): Could not find a package configuration file...`
    - **Cause:** Missing development libraries (e.g., LLVM, Clang, specific Python headers) or an outdated CMake version.
    - **Solution:** Ensure you have all required dependencies installed as per the HEIR GitHub README. Update CMake to a recent stable version (3.20+).
2. **Compilation (Build) Errors:**
    - **Issue:** Many C++ compilation errors (`g++` or `clang++` output).
    - **Cause:** Often related to C++ compiler versions, missing C++ standard library components, or specific flags/features not supported by your compiler.
    - **Solution:** Verify your C++ compiler (g++ or clang++) is up-to-date. Ensure you're using a C++17 or C++20 compatible compiler. Review the HEIR build instructions for specific compiler recommendations.

3. **`heir-opt` Command Not Found or Permission Denied:**
    - **Issue:** `./bin/heir-opt: No such file or directory` or `Permission denied`.
    - **Cause:** You might not be in the `HEIR/build` directory, or the build process failed to create the `heir-opt` executable. If "Permission denied," the executable might not have execute permissions.
    - **Solution:** Double-check your current directory. Re-run `cmake --build .` to ensure compilation completed. If permission denied, try `chmod +x ./bin/heir-opt`.
4. **Misunderstanding HEIR's Current State:**
    - **Issue:** Expecting a direct numeric output from `heir-opt` for a "Hello World" (e.g., input `2` results in `7`).
    - **Cause:** HEIR is a compiler framework in active development. Its current focus is on the intermediate transformations and optimizations, not yet on providing a fully integrated, user-friendly runtime for all FHE backends that produces immediate numeric results.
    - **Solution:** Adjust expectations. The "Hello World" is about successfully compiling and transforming your FHE program's representation, demonstrating HEIR's core functionality as a compiler. Full end-to-end execution will evolve as the project matures and integrates with specific FHE libraries.

Summary

Congratulations! You've successfully navigated the initial steps of setting up the HEIR compiler and processed your first conceptual FHE program.

Here are the key takeaways from this chapter:

- **HEIR is a Compiler Framework:** It translates high-level FHE computations into lower-level, optimized forms for FHE backends.
- **Active Development:** As of 2026-08-18, HEIR is in active development, meaning the focus is on the compilation pipeline, and direct end-to-end execution might require specific helper tools or further integration.
- **Setup Involves CMake:** You learned to clone the repository, use CMake for configuration, and build the HEIR tools from source.
- **MLIR as Intermediate Representation:** FHE computations are described using an MLIR-like syntax, which HEIR then processes.

- **`heir-opt` for Transformations:** The `heir-opt` utility allows you to apply various passes to your MLIR code, demonstrating HEIR's ability to analyze and transform FHE programs.

In the next chapter, we'll delve deeper into the types of operations HEIR can handle and explore how more complex FHE computations, like those needed for private AI inference, are represented and compiled.

References

- HEIR Compiler GitHub Repository
- A Guide For HEIR Experiment Evaluation (Original Version) - HEIR GitHub README
- CMake Official Documentation
- MLIR Documentation

This page is AI-assisted and reviewed. It references official documentation and recognized resources where relevant.

CHAPTER 07

Building Practical Privacy: Implementing Euclidean Distance with HEIR

Welcome back, future privacy architect! Imagine you're building an AI application that needs to compare sensitive user data—like health metrics or financial patterns—without ever seeing the raw information. How do you find the "distance" or "similarity" between two data points if they're both encrypted? This chapter tackles exactly that challenge.

In our previous discussions, we explored the fundamentals of Homomorphic Encryption (HE) and Fully Homomorphic Encryption (FHE), understanding HEIR's crucial role as a compiler for FHE programs. Now, it's time to bring these powerful concepts to life with a practical example: calculating the **Euclidean Distance** on encrypted data.

By the end of this chapter, you'll have a clear conceptual understanding of how to perform this fundamental AI operation privately using HEIR. You'll grasp not only how the process works but also why HEIR is a vital tool for balancing privacy and utility in AI.

Core Concepts: Euclidean Distance in Private AI

Euclidean Distance is a fundamental metric in countless machine learning algorithms. It quantifies the "straight-line" distance between two points in a multi-dimensional space, essentially telling us how similar or dissimilar two data points are.

What is Euclidean Distance and Why is it Challenging for Privacy?

For two vectors (or data points) `A = (a1, a2, ..., an)` and `B = (b1, b2, ..., bn)`, the Euclidean Distance `d(A, B)` is calculated using the formula:

`d(A, B) = sqrt( (a1 - b1)^2 + (a2 - b2)^2 + ... + (an - bn)^2 )`

This calculation is critical for:

- **Clustering:** Grouping similar data points together (e.g., in K-means).
- **Recommendation Systems:** Identifying items or users that are "close" to a given query.

- **Classification:** Determining which class a new data point belongs to based on its proximity to existing class centroids.

The inherent privacy challenge arises because computing this distance typically requires access to the raw, unencrypted values of `A` and `B`. If these vectors contain sensitive personal information, performing this computation on a remote server means exposing that data. This is a non-starter for privacy-preserving applications.

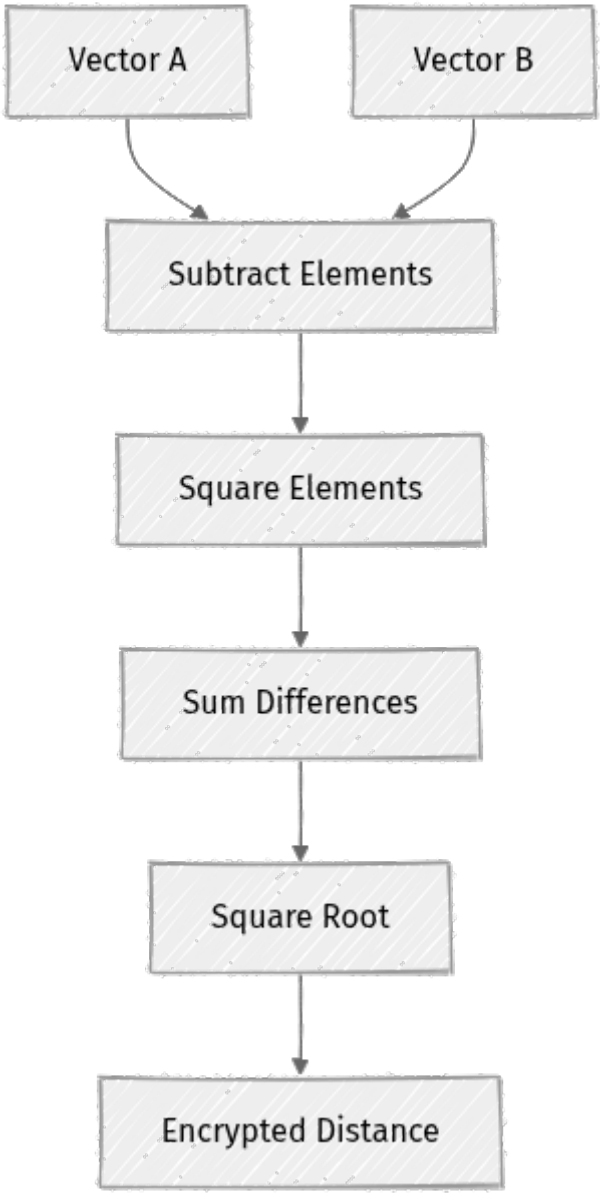
📌 **Key Idea:** FHE allows us to perform arithmetic operations directly on encrypted data, keeping sensitive information private throughout the computation. HEIR helps translate high-level operations into FHE-compatible instructions.

HEIR's Role in Private Euclidean Distance

HEIR (Homomorphic Encryption Intermediate Representation) acts as a specialized compiler infrastructure. Instead of developers needing to write intricate, low-level cryptographic code to handle operations on ciphertexts, HEIR allows you to describe your desired computation in a more accessible, higher-level manner.

HEIR then takes this description and translates it into an efficient, FHE-compatible program that can be run using various FHE backends (like SEAL or HElib). This abstraction is critical for making FHE practical for a broader range of developers.

Let's visualize the high-level flow of computing Euclidean Distance within an FHE context using HEIR:



This diagram illustrates the logical steps. HEIR's job is to ensure each of these steps can be executed securely on encrypted data.

Understanding FHE Operations for Euclidean Distance

To compute Euclidean Distance securely with FHE, we need to consider how each mathematical step translates into operations that FHE schemes can perform.

1. **Element-wise Subtraction (`a_i - b_i`):**
    - **What it is:** Subtracting corresponding elements of two vectors.
    - **Why it's important:** This is the first step in finding the difference between points.
    - **How FHE handles it:** FHE schemes natively support addition and subtraction on ciphertexts. If `Enc(a_i)` and `Enc(b_i)` are encrypted values, FHE can compute `Enc(a_i - b_i)` directly.
2. **Element-wise Squaring (`(a_i - b_i)^2`):**
    - **What it is:** Multiplying each difference by itself.
    - **Why it's important:** Squaring ensures all differences are positive and emphasizes larger differences.
    - **How FHE handles it:** FHE schemes also natively support multiplication on ciphertexts. If `Enc(diff_i)` is an encrypted difference, FHE can compute `Enc(diff_i * diff_i)`.
3. **Summation of Squared Differences (`sum(...)`):**
    - **What it is:** Adding all the squared differences together.
    - **Why it's important:** This aggregates the total "distance" before the final square root.
    - **How FHE handles it:** This is another series of addition operations, which FHE handles efficiently.

4. **Square Root (`sqrt(...)`):**
    - **What it is:** The final step to get the true Euclidean Distance.
    - **Why it's important:** It normalizes the squared sum back to a linear scale.
    - **How FHE handles it:** This is the trickiest part. FHE schemes primarily operate on polynomials. A true square root function is not a polynomial. Therefore, in FHE, square roots are typically handled in one of these ways:
        - **Polynomial Approximation:** Using a polynomial (e.g., Taylor series expansion) to approximate the square root over a specific, predefined range. This introduces some precision loss, which must be acceptable for the application.
        - **Avoiding the Square Root:** If the application only needs to compare distances (e.g., "is point A closer to B than to C?"), then comparing the squared Euclidean distances `d(A, B)^2` and `d(A, C)^2` is equivalent. This allows you to avoid the complex square root entirely.
        - **Post-Decryption:** The final result (the sum of squares) is decrypted, and the square root is taken on the plaintext. This requires trusting the party performing the decryption, which might not always align with the privacy goals.

⚡ **Real-world insight:** Many private AI scenarios, especially those involving clustering or nearest-neighbor searches, only require relative distances. By comparing squared distances, you can often bypass the computationally expensive and precision-losing square root operation in FHE.

🧠 **Important:** When working with FHE, always consider if an exact computation is strictly necessary or if an approximation or an alternative metric (like squared Euclidean distance) can fulfill your application's requirements. This often leads to significant performance and precision benefits.

Step-by-Step Implementation (Conceptual)

Since HEIR is a compiler, the "implementation" involves defining the computation graph in a way that HEIR can parse, understand, and translate into FHE-specific operations. As of 2026-08-18, HEIR is in active development, and its end-to-end integration for generating executables is still evolving. The `format_assistant/h` tool is highlighted for packaging compiled FHE programs.

This means we'll focus on the logic and structure you would provide to HEIR, rather than writing raw C++ FHE library calls. Assume you have HEIR built and configured (as discussed in earlier chapters, typically involving CMake).

1. Defining the High-Level Operation

Conceptually, you would define a function or program that takes two encrypted vectors and returns their encrypted Euclidean distance. Let's use a simplified, HEIR-friendly pseudo-language or an IR-like definition to illustrate the logical steps.

```
// Conceptual definition for HEIR
// This is not runnable code, but illustrates the logical steps HEIR would
compile.
// This describes the computation graph, not low-level FHE library calls.

function computeEuclideanDistance(encrypted_vector_A, encrypted_vector_B)
returns encrypted_scalar:
    // First, we'll prepare a container for our intermediate results:
    // an encrypted vector to store the squared differences for each element
pair.
    encrypted_squared_diffs =
new_encrypted_vector_of_same_size_as(encrypted_vector_A)

    // Next, we iterate through each corresponding element of the input
vectors.
    // For each pair (a_i, b_i), we'll compute (a_i - b_i)^2.
    for i from 0 to length(encrypted_vector_A) - 1:
        // Retrieve the i-th encrypted element from each input vector.
        // These are still ciphertexts, so their plaintext values remain
hidden.
        encrypted_a_i = get_element(encrypted_vector_A, i)
        encrypted_b_i = get_element(encrypted_vector_B, i)

        // Perform element-wise subtraction on the encrypted values.
        // The result, encrypted_diff_i, is also a ciphertext.
        encrypted_diff_i = subtract(encrypted_a_i, encrypted_b_i)

        // Square the encrypted difference. This is an encrypted
multiplication
        // of encrypted_diff_i by itself. The result is still encrypted.
        encrypted_squared_diff_i = multiply(encrypted_diff_i,
encrypted_diff_i)

        // Store this encrypted squared difference in our intermediate result
vector.
        set_element(encrypted_squared_diffs, i, encrypted_squared_diff_i)

    // After iterating through all elements, we have an encrypted vector
    // containing all the squared differences. Now, we sum these up.
    // This sum operation is also performed on ciphertexts, yielding an
encrypted scalar.
    encrypted_sum_of_squares = sum_elements(encrypted_squared_diffs)

    // Finally, we need to compute the square root. As discussed, this is
often
    // an approximation in FHE using polynomials. HEIR would select the
```

```
appropriate
    // polynomial approximation based on its configuration and the target FHE
backend.
    // We might conceptually call a function like 'approx_sqrt' here.
    encrypted_distance = approx_sqrt(encrypted_sum_of_squares)

    return encrypted_distance
end function
```

In a real HEIR workflow, you might express this computation using a domain-specific language (DSL) that HEIR can parse, or by directly constructing an Intermediate Representation (IR) program using HEIR's C++ API. The crucial point is that you're describing the computation graph or program logic, not writing the intricate low-level FHE operations yourself.

2. Compiling with HEIR (Conceptual Workflow)

Once your computation is defined in an HEIR-compatible format, the next step is to use the HEIR compiler.

1. **Input Program Preparation:** You would provide your high-level description (like the pseudo-code above, translated into a formal HEIR-compatible IR format, perhaps an MLIR dialect) to the HEIR compiler. Let's imagine this is in a file named `euclidean_distance.mlir`.
2. **Compilation to FHE-specific IR:** HEIR processes this program, performs optimizations suitable for homomorphic encryption, and translates it into an FHE-specific intermediate representation. This FHE-specific IR contains instructions tailored for a particular FHE backend (e.g., SEAL, HElib).
3. **Executable Generation:** As noted in the HEIR `README.md`, for an executable, you would use a tool like `format_assistant/h`. This tool helps in packaging the compiled FHE program into a runnable format that can take encrypted inputs and produce encrypted outputs.

Here's how a conceptual command-line interaction might look (actual syntax may vary as HEIR evolves):

```
# Conceptual command-line interaction with HEIR (example, actual syntax may
vary).
# These commands illustrate the pipeline, not exact current API.

# Step 1: Assume 'euclidean_distance.mlir' contains your HEIR IR definition.
#         This file conceptually holds the logic described in the pseudo-code.

# Step 2: Compile the HEIR program to an FHE-specific Intermediate
Representation (IR).
#         This step translates the generic HEIR IR into instructions optimized
for an FHE backend.
#         The '--target-backend SEAL' specifies which FHE library HEIR should
```

```
target.
heir-compiler --input euclidean_distance.mlir --output-fhe-ir compiled_euclide
an.fheir --target-backend SEAL

# Step 3: Generate an executable application from the FHE-specific IR.
#         As per HEIR documentation, 'format_assistant/h' is used for this
purpose.
#         This produces a standalone program that can process encrypted data.
format_assistant/h --fhe-ir compiled_euclidean.fheir --output-executable priva
te_euclidean_app
```

The resulting `private_euclidean_app` would be an application that, when executed, can accept ciphertexts (encrypted vectors `A` and `B`), securely perform the Euclidean Distance calculation on them, and output a ciphertext representing the distance. The client can then decrypt this final ciphertext to get the plaintext distance.

🧠 **Important: Current HEIR Status (Checked 2026-08-18)** The HEIR GitHub repository explicitly states: "integration between Middle-End and Back-End is not yet well-implemented." This means while the compiler infrastructure is robust, the end-to-end user experience for generating fully functional FHE executables might still be evolving and require careful attention to the latest documentation. Always refer to the official HEIR GitHub repository for the most current information and usage guidelines.

Mini-Challenge: Private Manhattan Distance

You've now walked through the conceptual flow for Euclidean Distance. Let's test your understanding with a related challenge!

Challenge: Adapt the conceptual HEIR program to calculate the **Manhattan Distance** (also known as L1 norm) between two encrypted vectors `A` and `B`.

The Manhattan Distance is defined as:

`d(A, B) = |a1 - b1| + |a2 - b2| + ... + |an - bn|`

Hint:

- How does the core operation sequence differ from Euclidean Distance? Specifically, which mathematical operation is replaced?

- The absolute value function `|x|` is not a simple polynomial. How might you conceptually handle `|x|` in an FHE-friendly way? Consider:
    - Polynomial approximations for `|x|` over a specific, known range.
    - If your data guarantees that `(a_i - b_i)` will always be positive within your FHE context, could you simplify? (This is a strong assumption, be careful!)
    - The `max(x, -x)` trick, and how `max` itself might be approximated or handled in FHE.

What to observe/learn:

- How different distance metrics directly translate into different FHE operation sequences.
- The challenges of implementing non-polynomial functions like `abs` or `max` in FHE, and why approximations or careful data design are often necessary.

Common Pitfalls & Troubleshooting

Working with FHE compilers like HEIR, especially during their active development phases, presents unique challenges. Being aware of these can save you significant debugging time.

1. ⚠️ **What can go wrong: High Performance Overheads and Resource Consumption.** FHE computations are inherently much slower and require significantly more memory than plaintext operations. Even a seemingly simple Euclidean Distance can take seconds or minutes on encrypted data, depending on chosen FHE parameters, vector dimensions, and the complexity of the FHE scheme.
    - **Troubleshooting:**
        - **Start small:** Begin with very small datasets and vector sizes to confirm correctness before scaling up.
        - **Optimize FHE Parameters:** If you have control over the FHE backend parameters (e.g., polynomial degree, number of slots, security level), experiment with values that meet your security requirements but are optimized for performance.
        - **Hybrid Approaches:** Consider if FHE is truly needed for all parts of your pipeline. Sometimes, a hybrid approach (e.g., FHE for sensitive parts, trusted execution environments for others) might be more practical.

2. ⚠️ **What can go wrong: Precision Loss with Approximations.** Operations like square root and absolute value often require polynomial approximations in FHE. This inherently introduces some loss of precision, which can be critical for applications that demand exact results.
    - **Troubleshooting:**
        - **Understand accuracy needs:** Clearly define the acceptable error margin for your application.
        - **Test rigorously:** Evaluate approximation errors with known inputs and compare against plaintext results.
        - **Alternative metrics:** Explore if your algorithm can function correctly with squared distances or other FHE-friendly alternatives that avoid problematic operations.
3. ⚠️ **What can go wrong: Data Scaling and Range Issues.** FHE schemes operate on integers or fixed-point numbers within a specific, finite range. Input data and all intermediate computation results must be carefully scaled to fit these constraints to avoid overflow, underflow, or incorrect results.
    - **Troubleshooting:**
        - **Normalize data:** Always normalize or scale your data before encryption.
        - **Track value ranges:** Be meticulous about estimating the maximum possible value any intermediate computation (e.g., `(a_i - b_i)^2`) can reach and ensure it stays within the FHE scheme's limits.
4. ⚠️ **What can go wrong: Difficulty in Debugging Encrypted Computations.** When issues arise, debugging FHE programs is extremely challenging because you cannot inspect intermediate encrypted values. All you see are ciphertexts.
    - **Troubleshooting:**
        - **Extensive plaintext testing:** Thoroughly test your logic with plaintext data before attempting FHE compilation.
        - **Small, controlled examples:** Use minimal examples with known inputs and expected outputs to isolate issues.
        - **Leverage compiler messages:** Pay close attention to HEIR's output and any error or warning messages, as these are your primary clues.

5. ⚠️ **What can go wrong: HEIR's Evolving Development State.** As highlighted, HEIR is under active development. Features, APIs, and documentation might change rapidly, and some functionalities might not be fully mature.
    - **Troubleshooting:**
        - **Stay updated:** Regularly refer to the latest HEIR GitHub repository for the most current information, examples, and best practices.
        - **Be prepared for experimentation:** Expect that you might need to experiment or adapt your approach as the project evolves.

Summary

You've taken a significant step forward in building practical privacy-preserving AI applications! This chapter guided you through the conceptual implementation of Euclidean Distance on encrypted data using HEIR.

Here are the key takeaways from our journey:

- **Euclidean Distance** is vital for many AI tasks, but poses a direct **privacy challenge** with sensitive data.
- **HEIR** acts as a powerful compiler, abstracting away the low-level FHE complexities, allowing developers to define computations at a higher level.
- We broke down the Euclidean Distance formula into **FHE-compatible operations**: element-wise subtraction, squaring, and summation.
- The **square root** operation in FHE often requires polynomial approximations or can be avoided if only relative distances are needed.
- We walked through a **conceptual HEIR workflow**, demonstrating how a high-level description of the computation can be translated into an FHE-ready program.
- You tackled a **mini-challenge** to consider Manhattan Distance, highlighting the translation of different mathematical functions into FHE and the specific challenges of non-polynomial operations.
- We discussed crucial **common pitfalls** in FHE development, including performance overheads, precision loss, data scaling, debugging difficulties, and HEIR's evolving nature, along with practical troubleshooting strategies.

This chapter has equipped you with a deeper understanding of how to bridge the gap between AI algorithms and privacy preservation. In the next chapter, we'll continue to explore more advanced use cases and dive into other essential FHE operations, building on the strong foundation you've established here.

References

- heir-compiler/HEIR GitHub Repository
- A Guide For HEIR Experiment Evaluation (Original Version)
- CMake Official Documentation

This page is AI-assisted and reviewed. It references official documentation and recognized resources where relevant.

CHAPTER 08

Evaluating HEIR Applications: Performance, Limitations, and Best Practices

Welcome back, privacy pioneers! You've journeyed through the intricate world of Homomorphic Encryption (HE) and Fully Homomorphic Encryption (FHE), grasped HEIR's role as a groundbreaking compiler, and even dabbled in its setup. You're now equipped with the foundational knowledge to appreciate one of the most critical aspects of private AI: its performance.

In this chapter, we're shifting gears from how HEIR works to how effectively it works in practice. Building privacy-preserving AI is incredibly powerful, but it comes with unique performance considerations. We'll dive deep into evaluating HEIR applications, understanding their performance implications, acknowledging current limitations (as of 2026-08-18), and discussing emerging best practices for this nascent field.

Why Performance Evaluation is Your Privacy Shield

Imagine you've built an AI model that can accurately predict medical conditions. Running inferences on patient data with FHE ensures ultimate privacy. But what if each inference takes several minutes, or consumes gigabytes of memory? Your groundbreaking privacy solution might become unusable in a real-world clinic.

Understanding these tradeoffs is paramount. It allows you to design FHE applications that are not just cryptographically secure, but also practically viable. This chapter will empower you to make informed engineering decisions, setting realistic expectations and guiding you toward deployable, privacy-preserving AI systems.

The Inherent Cost of FHE: Performance Deep Dive

Homomorphic encryption is a marvel, allowing computation on encrypted data without ever decrypting it. However, this cryptographic magic isn't free. It introduces significant overhead, primarily affecting:

1. **Latency (Computation Time):** Every operation on encrypted data (addition, multiplication, etc.) is orders of magnitude slower than its plaintext counterpart. This is due to the complex polynomial arithmetic and intricate noise management involved.
    - **Why it matters:** High latency can make real-time or interactive FHE applications impractical.
2. **Memory Usage:** Ciphertexts (encrypted data) are significantly larger than plaintexts. A single encrypted 64-bit integer might expand to thousands of bytes. This bloats memory consumption during computation and for storing results.
    - **Why it matters:** Increased memory requirements can lead to higher infrastructure costs and limit the scale of data you can process.
3. **Bandwidth:** Transferring these larger ciphertexts across networks naturally consumes more bandwidth, adding to network latency and costs.
    - **Why it matters:** For distributed FHE applications, bandwidth can become a major bottleneck.

Understanding Noise and the Bootstrapping Challenge

A fundamental concept in FHE is "noise." Every homomorphic operation, especially multiplication, adds a small amount of cryptographic noise to the ciphertext. If this noise accumulates too much, the ciphertext becomes undecipherable, and the computation fails.

Fully Homomorphic Encryption schemes address this through a process called **bootstrapping**. This operation essentially "refreshes" a noisy ciphertext, reducing its noise level without decrypting it.

`📌 Key Idea:` Bootstrapping is the most computationally intensive operation in FHE, often taking seconds or even minutes for a single refresh. Minimizing or avoiding bootstrapping is a critical strategy for improving FHE performance.

HEIR's Role in Optimizing FHE Workloads

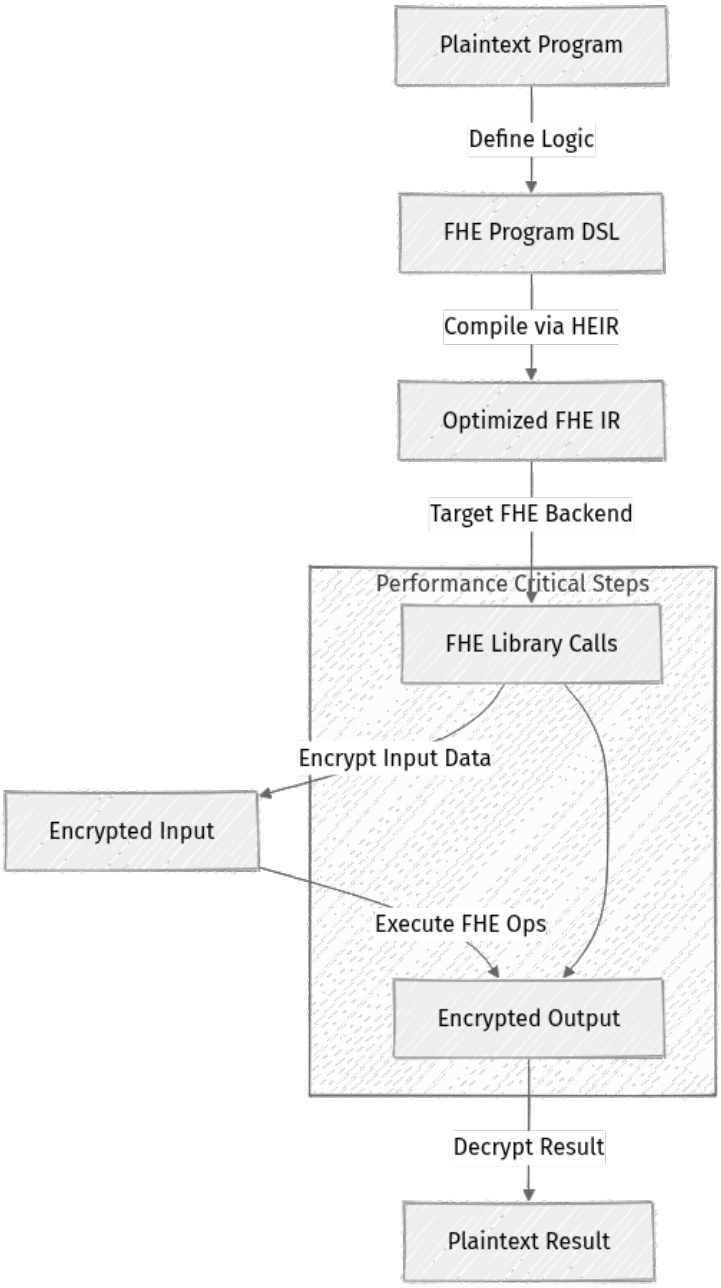
HEIR, as an FHE compiler, is designed to mitigate some of these inherent performance challenges by translating high-level FHE programs into optimized low-level operations. It achieves this through:

- **Compiler-driven Optimization:** HEIR can analyze the FHE program's structure, identify opportunities to reorder operations, choose optimal cryptographic parameters, or fuse operations to reduce noise growth and computational cost.
- **Targeting Specific FHE Backends:** Different FHE libraries (e.g., SEAL, TFHE, OpenFHE) have distinct performance characteristics for various operations and schemes. HEIR aims to abstract this complexity, potentially allowing developers to target the most efficient backend for their specific workload without rewriting their FHE logic.

🧠 `Important:` While HEIR strives for optimal performance, it cannot eliminate the fundamental overhead of FHE. It helps you achieve the best possible performance within the constraints of homomorphic encryption.

Visualizing the FHE Compilation and Execution Flow

Let's visualize how HEIR fits into the FHE execution pipeline and where performance bottlenecks typically emerge.



The "Performance Critical Steps" subgraph highlights the core FHE operations where the most significant overhead occurs. HEIR (step C) directly influences the efficiency of these FHE operations.

HEIR's Current State: Practical Implications (as of 2026-08-18)

It's crucial to acknowledge that HEIR is an actively developing open-source project. This has important practical implications for developers:

- **Early Stage Development:** The HEIR GitHub repository notes that "integration between Middle-End and Back-End is not yet well-implemented." This indicates that the full end-to-end optimization capabilities might still be maturing, and the compiler's full potential is still being realized.
- **Experimental Nature:** While incredibly powerful and promising, HEIR is still experimental. Performance benchmarks and API stability should be expected to evolve rapidly.
- **Tooling and Documentation:** Comprehensive performance guides, extensive benchmarking suites, and stable APIs are likely still under active development. Developers should anticipate engaging directly with the codebase and potentially contributing to its evolution.
- **Executable Limitations:** The documentation explicitly states: "If you require an executable, please use `format_assistant/h`." This implies that direct, general-purpose executable generation from HEIR might not be straightforward for all use cases, and interaction might be through specific tools or integration points.

`⚡ Real-world insight:` When evaluating HEIR, it's vital to factor in its developmental stage. Be prepared to conduct your own benchmarking for your specific use cases and stay vigilant for project updates. Your findings might contribute valuable feedback to the HEIR community!

Step-by-Step Approach: Designing an FHE Performance Evaluation

Given HEIR's current developmental stage and the `format_assistant/h` executable limitation, a "step-by-step implementation" of writing and running complex HEIR code for benchmarking might not be universally feasible without significant integration work.

Instead, let's focus on the methodology of performance evaluation. This section guides you through the process of designing a robust benchmark for an FHE operation, even if the direct execution relies on conceptual understanding or specific HEIR utility tools.

Goal: Measure Latency and Memory for Homomorphic Addition

We'll outline how to approach benchmarking a fundamental FHE operation: homomorphic addition (`A + B`).

1. **Identify the Target Operation:** Clearly define the specific FHE operation you want to measure. For example, `cipher_result = HEIR.add(cipher_A, cipher_B)`.
2. **Select Cryptographic Parameters:** FHE schemes require a set of parameters (e.g., polynomial degree, coefficient moduli chain, security level). These parameters directly influence both security and performance.
    - **Action:** Choose parameters that meet your desired security level (e.g., 128-bit security) and are suitable for the complexity of your computation. These will be passed to the FHE context initialization.
3. **Prepare Input Data:**
    - **Action:** Generate a sufficient number of random plaintext inputs (e.g., `N` pairs of integers for addition).
    - **Action:** Encrypt these inputs once before starting your timed benchmark loop. Encryption itself is part of the overall FHE process, but for benchmarking the homomorphic operation, you want to exclude the initial encryption time.
4. **Design the Measurement Loop:**
    - **Action:** Start a high-resolution timer (e.g., `time.perf_counter()` in Python, `std::chrono` in C++).
    - **Action:** Execute the target homomorphic operation (`N` times) within this timed loop.
    - **Action:** Monitor peak memory usage during the loop. Tools like `psutil` (Python) or `getrusage` (Linux C++) can help.
    - **Action:** Stop the timer after `N` iterations.
5. **Process and Analyze Results:**
    - **Action:** Calculate the total elapsed time.
    - **Action:** Compute the average latency per operation by dividing total time by `N`.
    - **Action:** Record the peak memory usage observed.

6. **Verify Correctness (Post-Benchmarking):**
    - **Action:** Decrypt a sample of the `cipher_result` outputs and compare them against the plaintext sum of the original inputs to ensure the computation was correct. This step is crucial for validating your benchmark.
7. **Iterate and Vary Parameters:**
    - **Action:** Repeat the entire process with different FHE parameters (e.g., larger polynomial degrees, different coefficient moduli) or varying input sizes (`N`). This helps you understand how these choices impact performance.

Mini-Challenge: Benchmarking a Conceptual FHE Operation

Now, let's put on our engineering hats.

Challenge: Based on the methodology above, outline the conceptual steps you would take to measure the performance (latency and memory) of a basic homomorphic multiplication operation using HEIR, assuming you have a way to compile and run it via HEIR's `format_assistant/h` or a similar test harness.

Hint: Homomorphic multiplication has a different impact on noise than addition. How might this influence your measurement or the parameters you choose?

What to Observe/Learn: This challenge encourages you to think critically about designing a scientific experiment for performance, considering the unique characteristics of FHE operations.

Conceptual Solution Approach (Do not run this code, it's illustrative):

Here's how you might structure the conceptual code, building on the ideas from our step-by-step guide. This pseudocode is for conceptual understanding of the benchmarking process, not runnable HEIR code itself (as of 2026-08-18).

```
# Conceptual Python-like pseudocode for FHE multiplication benchmarking

import time
import os
import psutil # For conceptual memory monitoring
import random

def benchmark_fhe_multiplication(num_iterations, fhe_params):
    """
    Conceptual function to benchmark homomorphic multiplication.
    This simulates the steps you'd take, assuming an HEIR-compiled executable
exists.
    """
    print(f"--- Starting FHE Multiplication Benchmark ---")
    print(f"Initializing FHE context with parameters: {fhe_params}...")
```

```
    # In a real scenario, this would involve calling HEIR's runtime or a
specific FHE backend
    # context = HEIR.initialize(fhe_params) # Conceptual API call

    # 1. Generate and encrypt inputs (outside timed loop)
    # FHE multiplication often requires inputs of specific ranges or types.
    # For this conceptual example, let's assume integers.
    plain_a_values = [random.randint(1, 100) for _ in range(num_iterations)]
    plain_b_values = [random.randint(1, 100) for _ in range(num_iterations)]

    # Conceptual encryption step
    # cipher_a = [context.encrypt(x) for x in plain_a_values]
    # cipher_b = [context.encrypt(x) for x in plain_b_values]
    print(f"Generated and conceptually encrypted {num_iterations} pairs of
inputs for multiplication.")

    # 2. Start timing homomorphic operations
    start_time = time.perf_counter()
    peak_memory = 0

    print("Executing homomorphic multiplication benchmark...")
    for i in range(num_iterations):
        # Conceptual call to HEIR-compiled homomorphic multiplication.
        # This would be the direct interaction with the HEIR output.
        # result_cipher = HEIR.multiply(cipher_a[i], cipher_b[i])

        # Simulate memory usage monitoring (conceptual)
        process = psutil.Process(os.getpid())
        current_memory_mb = process.memory_info().rss / (1024 * 1024)
        if current_memory_mb > peak_memory:
            peak_memory = current_memory_mb

    end_time = time.perf_counter()
    elapsed_time = end_time - start_time

    print(f"Finished {num_iterations} homomorphic multiplications.")
    print(f"Total time: {elapsed_time:.4f} seconds")
    print(f"Average time per multiplication: {(elapsed_time / num_iterations
* 1000):.4f} ms")
    print(f"Peak memory usage during operations: {peak_memory:.2f} MB")

    # Optional: Decrypt and verify correctness (conceptual)
    # decrypted_result = context.decrypt(result_cipher)
    # assert decrypted_result == plain_a_values[-1] * plain_b_values[-1] #
Verify last result

# Example conceptual usage:
# A realistic FHE parameter set would be much more complex.
# This is a placeholder for illustration.
# benchmark_fhe_multiplication(100, {"poly_mod_degree": 8192,
"coeff_mod_bits": [40, 30, 40]})
```

This conceptual code emphasizes the systematic approach: setup, isolation of the core operation, precise timing, and resource measurement.

Common Pitfalls & Troubleshooting in HEIR Development

Working with HEIR and FHE, especially in its active development phase, can present unique challenges. Here are some common issues and how to approach them:

1. **Cryptographic Parameter Mismatch:**
    - ⚠️ `What can go wrong:` Using inconsistent FHE parameters (e.g., polynomial degree, coefficient moduli chain) during encryption, computation, or decryption. This will lead to errors, undecipherable results, or security vulnerabilities.
    - **Troubleshooting:** Always ensure the exact same `FHEContext` or equivalent parameters are used consistently throughout the entire FHE lifecycle. Refer to the HEIR documentation (or the specific FHE backend library's docs) for recommended parameter sets tailored to your security level and computational depth. Small parameter changes can have drastic performance and security impacts.
2. **Noise Growth and Unbootstrapped Ciphertexts:**
    - ⚠️ `What can go wrong:` Performing too many homomorphic multiplications without adequate bootstrapping. Each multiplication significantly increases noise. If the noise exceeds the scheme's capacity, the ciphertext becomes corrupted and cannot be correctly decrypted. Alternatively, attempting to bootstrap when the FHE scheme or HEIR's current integration doesn't fully support it.
    - **Troubleshooting:**
        - **Algorithm Design:** Carefully design your FHE computation to minimize the "multiplication depth" (the longest chain of sequential multiplications).
        - **Bootstrapping Strategy:** If your computation requires deep circuits, ensure your chosen FHE scheme and HEIR's current capabilities support bootstrapping. Plan when and where to apply bootstrapping to manage noise.
        - **Debug Mode:** If available, utilize debug modes or tools within FHE libraries to inspect intermediate ciphertexts and monitor their noise levels.

3. **High Resource Consumption:**
    - ⚠️ `What can go wrong:` Your FHE application consumes excessive CPU, memory, or time, rendering it impractical for deployment.
    - **Troubleshooting:**
        - **Algorithm Optimization:** Can you reformulate the computation to reduce the number of homomorphic operations (especially multiplications) or use fewer encrypted inputs?
        - **Parameter Tuning:** Experiment with the smallest possible FHE parameters that still meet your security requirements. Smaller parameters generally mean faster, less memory-intensive operations.
        - **Batching/SIMD:** If your FHE scheme supports it (e.g., BFV/CKKS), pack multiple plaintext values into a single ciphertext. This allows you to perform a single homomorphic operation on many data points simultaneously (Single Instruction, Multiple Data - SIMD), dramatically improving throughput.
        - **Hardware Acceleration:** For extreme performance needs, specialized hardware (e.g., FPGAs, ASICs, GPUs) is an active area of research for accelerating FHE operations.

Best Practices for Developing with HEIR

Given HEIR's evolving nature and the inherent complexities of FHE, here are some best practices to guide your development:

- **Start Simple and Iterate:** Begin with the absolute smallest FHE program that demonstrates your core logic. Get it working correctly and then gradually increase complexity. This allows you to isolate issues and understand performance impacts incrementally.
- **Profile Aggressively from Day One:** Implement robust benchmarking and profiling from the very beginning. Measure latency, throughput, and memory usage for all critical operations and overall workflows. This data is invaluable for informed decision-making.
- **Understand Your FHE Backend:** Even as HEIR abstracts the underlying FHE libraries, having a fundamental understanding of the core FHE schemes (BFV, BGV, CKKS for approximate numbers, TFHE for arbitrary functions) will help you design more efficient FHE programs and make better choices about parameters.

- **Mind the Noise Budget:** Always be acutely aware of the noise introduced by homomorphic operations. Design your computations to stay within the noise budget of your chosen parameters, or strategically plan for bootstrapping.
- **Security First, Always:** Never compromise on cryptographic security parameters for the sake of performance. Always use parameters recommended by cryptographers for your desired security level (e.g., 128-bit security is a common minimum).
- **Stay Updated with HEIR:** HEIR is an actively evolving project. Regularly check the official GitHub repository for updates, new features, performance improvements, and changes in recommended usage. (Information checked on 2026-08-18).
- **Leverage Batching (if applicable):** For FHE schemes like BFV and CKKS, packing multiple data points into a single ciphertext (SIMD operations) can dramatically improve throughput. Design your data structures and computations to exploit this parallelism wherever possible.

Summary: Your Toolkit for Practical Private AI

In this chapter, you've gained crucial insights into the practical aspects of evaluating HEIR applications for private AI. We've covered:

- The inherent performance costs of Homomorphic Encryption, including increased latency, memory, and bandwidth.
- The critical role of noise management and the computational expense of bootstrapping in FHE.
- How HEIR aims to optimize FHE programs, while acknowledging that it doesn't eliminate fundamental FHE overhead.
- The current developmental stage of HEIR (as of 2026-08-18) and its implications for developers.
- A step-by-step methodology for designing effective FHE performance benchmarks.
- Common pitfalls like parameter mismatches, noise issues, and resource consumption, along with troubleshooting strategies.
- Essential best practices for developing robust and efficient privacy-preserving AI solutions with HEIR.

You now have a clearer, more realistic picture of the challenges and opportunities in building and evaluating privacy-preserving AI with HEIR. The journey into practical private AI is complex but incredibly rewarding, and you're now better equipped to navigate it.

What's Next?

With a solid understanding of HEIR's capabilities, limitations, and evaluation strategies, you're well-equipped to dive deeper into building more complex private AI applications. The next steps could involve exploring specific real-world use cases, integrating HEIR into larger system architectures, or even contributing to the open-source HEIR community as it continues to mature. Keep experimenting, keep learning, and keep building the future of private AI!

References

1. HEIR Compiler GitHub Repository
2. HEIR README.md (Original Version - A Guide For HEIR Experiment Evaluation)
3. OpenFHE Library Documentation
4. Microsoft SEAL Documentation
5. PALISADE Homomorphic Encryption Library

This page is AI-assisted and reviewed. It references official documentation and recognized resources where relevant.

CHAPTER 09

The Road Ahead: Real-World Use Cases and the Future of Private AI with FHE

Welcome to the final chapter of our comprehensive guide on Homomorphic Encryption (HE) and the HEIR compiler! Throughout this series, we've dissected the cryptographic marvels behind Fully Homomorphic Encryption (FHE), understood its profound implications for data privacy, and explored HEIR's role as a groundbreaking compiler for private AI inference.

In this concluding chapter, our journey takes a crucial turn from theoretical understanding to practical foresight. We'll delve into the most compelling real-world applications where FHE, powered by tools like HEIR, can truly redefine data utility and privacy. We'll then examine the significant challenges that still lie ahead and cast our gaze towards the exciting future of privacy-preserving AI.

To gain the most from this exploration, ensure you're familiar with:

- The fundamental differences between Homomorphic Encryption (HE) and Fully Homomorphic Encryption (FHE).
- The concept of private AI inference and its necessity.
- HEIR's architecture and its function as an FHE compiler, particularly its use of intermediate representations.

Let's envision a future where privacy is an inherent feature of every AI interaction, not an afterthought.

Real-World Impact: Where FHE and HEIR Transform Industries

Imagine a world where organizations can extract invaluable insights from vast datasets without ever needing to decrypt or expose the underlying sensitive information. This isn't a distant dream; it's the core promise of FHE, and open-source compilers like HEIR are designed to bridge the gap between this promise and practical, deployable AI systems. Let's explore some of the most impactful scenarios.

Privacy-Preserving Data Analytics

Many industries rely on analyzing massive, often highly sensitive datasets. Regulations like GDPR and HIPAA, coupled with ethical considerations, frequently restrict or complicate such analyses. FHE, by enabling computation directly on encrypted data, opens unprecedented avenues for analytics that were previously impossible.

Healthcare: Securing Genomic Research and Personalized Medicine

Healthcare stands out as a prime candidate for FHE adoption. Genomic sequences, individual patient records, and confidential drug trial results are among the most sensitive data points imaginable. ⚡ Real-world insight: Hospitals and research consortia frequently face immense hurdles in collaborating on large-scale studies due to the legal and ethical complexities of sharing raw patient data.

With FHE, researchers could achieve breakthroughs while maintaining stringent privacy:

- **Secure Collaborative Research:** Multiple medical institutions could pool their encrypted genomic data to conduct large-scale statistical analyses for identifying disease biomarkers or accelerating drug discovery. Crucially, no single party would ever gain access to the unencrypted genetic information of individuals.
- **Personalized Treatment Plans:** AI models could predict an individual patient's response to specific treatments based on their encrypted genetic markers and health history. The AI service provider would only ever process encrypted inputs and return an encrypted prediction, maintaining patient confidentiality.

Financial Services: Enhancing Fraud Detection and Credit Risk Assessment

Financial transactions, customer spending patterns, and credit histories are high-value targets for both legitimate analysis and malicious actors. FHE offers a robust solution for privacy-preserving analysis in this sector. 📌 Key Idea: FHE allows financial institutions to detect complex patterns and make predictions from sensitive data without ever directly viewing raw transaction details or personal financial records.

Consider these critical applications:

- **Advanced Fraud Detection:** Banks could securely share encrypted transaction patterns with a centralized AI service to identify sophisticated fraud rings operating across multiple institutions. Individual customer transactions would remain encrypted and private throughout the process.
- **Private Credit Scoring:** Lenders could evaluate a user's creditworthiness by running a credit scoring model on encrypted financial data. The model operates entirely on encrypted inputs, returning only an encrypted score. Neither the lender nor the model provider would ever see the raw financial history.

Private AI Inference as a Service (AIaaS)

This is where the HEIR compiler's role becomes truly transformative. Cloud providers offer powerful, pre-trained AI models as a service, but feeding sensitive user data into these models often requires a trade-off with privacy. FHE, orchestrated by compilers like HEIR, eliminates this dilemma.

Imagine a typical AIaaS interaction with an FHE-enabled twist:

1. **User Data:** A user possesses sensitive data (e.g., medical images for diagnosis, personal text for sentiment analysis, financial statements for predictive modeling).
2. **AI Model Host:** A cloud service hosts a sophisticated AI model that can provide valuable insights from this data.
3. **Privacy Goal:** The user wants to obtain a prediction from the AI model without revealing their raw data to the cloud service provider.

With an FHE compiler like HEIR, this becomes a practical reality:

- **Encryption by User:** The user encrypts their private input data using a chosen FHE library (e.g., OpenFHE, SEAL, HElib). This generates ciphertext.
- **Encrypted Data Transmission:** The encrypted data is securely transmitted to the cloud service.
- **Model Compilation (HEIR's Role):** HEIR, or a system leveraging HEIR, takes the original AI model (often expressed in an intermediate representation like MLIR) and compiles it into an FHE-compatible "circuit" of homomorphic operations. This circuit is a sequence of FHE-friendly mathematical operations.
- **Homomorphic Execution:** The cloud service executes this FHE-compiled model directly on the encrypted input data.

- **Encrypted Result:** An encrypted prediction is generated and sent back to the user.
- **Decryption by User:** Only the original user, possessing the secret key, can decrypt the final prediction.

This end-to-end process guarantees that the sensitive data remains encrypted throughout the entire inference pipeline, offering robust privacy. It unlocks services such as:

- **Private Image Diagnosis:** Upload an encrypted medical image to a cloud AI for anomaly detection without the cloud provider ever viewing the image content.
- **Secure Text Analysis:** An enterprise could use an FHE-enabled sentiment analyzer on encrypted customer feedback, ensuring privacy while gaining business intelligence.

Synergies with Other Privacy-Enhancing Technologies (PETs)

FHE is a potent privacy tool, but its power is often amplified when combined with other Privacy-Enhancing Technologies (PETs). ⚡ Quick Note: Other prominent PETs include Differential Privacy (DP) and Secure Multi-Party Computation (MPC).

- **FHE + Differential Privacy:** FHE ensures that data remains encrypted during computation, preventing direct exposure. Differential Privacy then adds carefully calibrated noise to the outputs of the computation, providing a mathematical guarantee against re-identification attacks, even if the decrypted results are aggregated or released. This combination offers a formidable, layered privacy defense.
- **FHE + Secure Multi-Party Computation (MPC):** MPC allows multiple parties to jointly compute a function over their private inputs without revealing those inputs to each other. FHE can serve as an efficient building block within MPC protocols for specific types of computations, offering different performance and security trade-offs depending on the application.

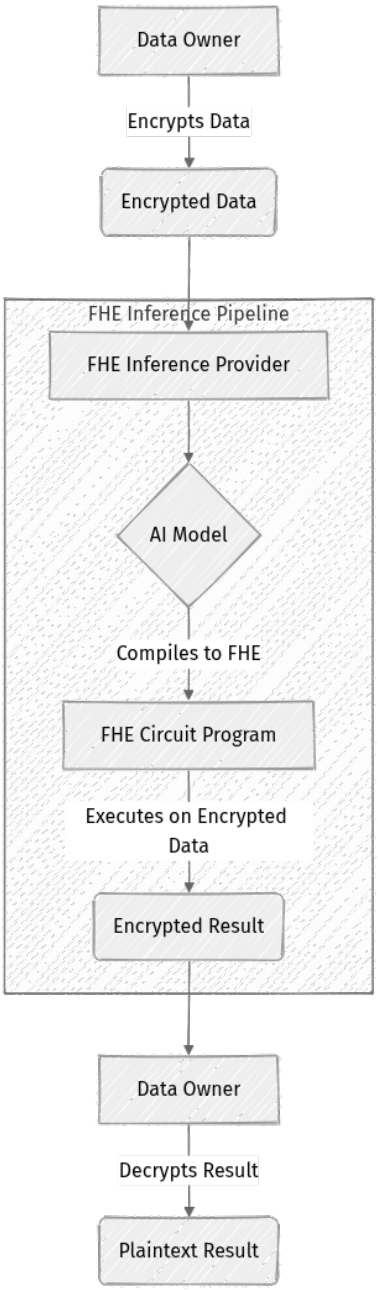
Enhancing Decentralized and Federated Learning

Federated learning (FL) inherently offers privacy by training a global AI model on decentralized datasets, where raw data never leaves the user's device. FHE can further fortify this privacy model.

- **Secure Aggregation of Updates:** When local model updates (e.g., gradients) are sent from individual devices to a central server for aggregation, FHE can encrypt these updates. This ensures that the server aggregates encrypted values, preventing it from inspecting individual contributions and adding an extra layer of privacy and robustness against malicious servers.

Conceptual Workflow: Private AI Inference with HEIR

To solidify your understanding, let's visualize the high-level steps involved in performing private AI inference using an FHE compiler like HEIR. This diagram illustrates the system's conceptual flow, rather than specific code.



Understanding the Flow:

- **A[Data Owner]:** This represents an individual user or an organization possessing sensitive data that wishes to utilize an AI model without exposing their raw inputs.
- **B(Encrypted Data):** The data owner encrypts their sensitive data using an FHE library. The output is a ciphertext, which is a scrambled version of the original data, unintelligible without the correct key.
- **C[FHE Inference Provider]:** This is typically a cloud service or a third-party entity that hosts the AI model and is equipped to perform FHE computations. They receive the encrypted data.

- **D{AI Model}:** This node represents the original AI model (e.g., a neural network, a decision tree, a linear regressor) that the FHE Inference Provider offers.
- **E[FHE Circuit Program]:** This is where HEIR's magic happens. The AI model is transformed by HEIR into an FHE-compatible program. This program is essentially a sequence of homomorphic operations that can be performed directly on encrypted data.
- **F(Encrypted Result):** The FHE circuit program executes on the encrypted input data, producing an encrypted output. At no point during this computation does the FHE Inference Provider see the plaintext inputs or outputs.
- **G[Data Owner]:** The encrypted result is securely transmitted back to the original data owner.
- **H(Plaintext Result):** The data owner, and only the data owner, uses their secret key to decrypt the result, obtaining the plaintext prediction from the AI model.

This entire conceptual pipeline ensures that sensitive data remains cryptographically protected throughout the AI inference process, providing a robust privacy guarantee.

Mini-Challenge: Designing a Privacy-First Smart City System

Let's apply your understanding of FHE's potential to a practical system design problem.

Challenge: You are tasked with designing a privacy-preserving system for a smart city initiative. The city wants to analyze real-time traffic sensor data (e.g., vehicle counts, average speeds, traffic density) to dynamically optimize traffic light timings, predict congestion, and improve urban planning. Residents are highly concerned about privacy, specifically that their individual travel patterns or routes could be tracked or inferred from this data.

Your Task: Describe how you would design a system using FHE and an FHE compiler like HEIR to address these privacy concerns. Outline the key components, the flow of encrypted data, and identify the points where encryption and decryption would occur. Focus on the system architecture and data privacy boundaries rather than specific code.

Hint: Consider the roles of different entities (sensors, data aggregators, AI processing units, city planners). What data needs to be encrypted at the source? Who holds the keys? What is the final output, and does it need to be decrypted?

What to observe/learn: This exercise challenges you to bridge the gap between FHE theory and real-world system architecture. It encourages you to think critically about data lifecycle, trust models, and how FHE can be integrated to achieve meaningful privacy guarantees in complex applications.

Common Pitfalls & Future Challenges

While the promise of FHE and compilers like HEIR is immense, it's crucial to acknowledge that the technology is still in active development and faces significant challenges before widespread adoption.

⚠️ What can go wrong: Performance Overhead

The most significant hurdle for FHE remains its inherent performance overhead. Homomorphic operations are fundamentally more computationally intensive than their plaintext equivalents.

- **Latency:** FHE computations can be orders of magnitude slower than traditional operations. This makes FHE challenging for real-time or low-latency applications.
- **Throughput:** Processing large volumes of encrypted data can lead to extremely slow batch processing, impacting the scalability of FHE-enabled systems.
- **Memory Footprint:** Ciphertexts are typically much larger than plaintexts, often by factors of hundreds or thousands, leading to increased memory consumption and bandwidth requirements.

While compilers like HEIR aim to optimize the translation and execution of FHE programs, these fundamental performance bottlenecks are rooted in the cryptographic primitives themselves and are a subject of intense ongoing research.

Complexity of FHE Development

FHE is a highly specialized and mathematically complex field. Developing FHE-enabled applications from scratch still requires a deep understanding of:

- **Cryptographic Schemes:** Different FHE schemes (e.g., BGV, BFV for exact arithmetic; CKKS for approximate arithmetic) have distinct properties, security assumptions, and suitability for various computation types.
- **Parameter Selection:** Choosing the correct FHE parameters (e.g., polynomial degree, number of prime moduli, scaling factors) is critically important for both the security and performance of the system. Incorrect parameter choices can lead to insecure systems or computations that are practically infeasible.
- **Arithmetic Limitations:** FHE schemes often impose limitations on the types of operations they can perform efficiently (e.g., restricted integer ranges, challenges with non-linear functions, approximate nature of floating-point operations in CKKS).

Compilers like HEIR are designed to abstract away much of this low-level cryptographic complexity by allowing developers to express computations in a higher-level language. However, understanding the underlying FHE capabilities and constraints remains important for effective application design.

Bootstrapping Costs

Fully Homomorphic Encryption schemes require a process called "bootstrapping" to refresh noisy ciphertexts, which is essential to allow an arbitrary number of homomorphic operations without noise accumulation rendering the ciphertext undecryptable. Bootstrapping is an extremely computationally intensive operation and remains a major performance bottleneck. While significant research is dedicated to improving its efficiency, it still represents a substantial cost in most FHE applications.

HEIR's Current Development Status (Checked 2026-08-18)

As an open-source project, HEIR is in active development and, like many cutting-edge research compilers, should be considered experimental rather than production-ready.

- **Integration Gaps:** The official HEIR GitHub repository explicitly states that "integration between Middle-End and Back-End is not yet well-implemented." This indicates that the full end-to-end compilation pipeline might still require manual steps or be less streamlined than a mature production compiler.

- **Executable Generation:** For practical testing and execution, developers might currently need to rely on specific utilities like `format_assistant/h`, as mentioned in the HEIR documentation. This suggests that direct, simplified executable generation from the main compiler might still be a work in progress.
- **Evolving Documentation:** As with any rapidly evolving project in a complex domain, documentation, best practices, and API stability are subject to change. Developers should consistently refer to the latest official repository and community discussions.

Lack of Standardized Practices

The FHE ecosystem is still relatively nascent and maturing. There isn't yet a widely adopted set of industry standards or best practices for deploying FHE in production environments, particularly for AI inference. This can introduce challenges in system design, interoperability, auditing, and regulatory compliance.

The Future of Private AI with FHE

Despite the current challenges, the trajectory of FHE and private AI is incredibly promising. The pace of innovation and investment in this field is rapidly accelerating.

Democratizing FHE Through Compilers

Projects like HEIR are instrumental in making FHE accessible to a broader audience of developers. By compiling high-level AI models into FHE-compatible operations, HEIR abstracts away much of the underlying cryptographic complexity. This "compiler approach" is critical for FHE to transition from academic research into mainstream software engineering, allowing AI engineers to focus on model development rather than intricate cryptographic details.

Hardware Acceleration for FHE

The development of dedicated hardware accelerators for FHE is a major area of research and investment. Specialized chips, similar to how GPUs revolutionized deep learning, could drastically reduce the performance overhead of homomorphic operations. This could make real-time private AI inference a practical reality for a wider range of applications.

Standardization and Interoperability

Efforts are underway within academic, industry, and standards bodies to standardize FHE schemes, APIs, and best practices. This will foster greater interoperability between different FHE libraries, compilers, and tools, making it easier to build robust, secure, and maintainable FHE-enabled applications across various platforms.

Broader Integration with Privacy-Enhancing Technologies

The future will likely see FHE increasingly integrated with other PETs, such as differential privacy, trusted execution environments (TEEs), and secure multi-party computation. These layered privacy solutions will address diverse threat models and provide more comprehensive privacy guarantees.

Expanding AI Model Compatibility

As FHE schemes and compilers continue to evolve, they will gradually support a wider array of AI model architectures, non-linear operations, and data types. This expansion will enable private inference for increasingly complex and sophisticated machine learning tasks, pushing the boundaries of what's possible with privacy-preserving AI.

Summary

Congratulations on completing this in-depth exploration of Homomorphic Encryption and the HEIR compiler! We've journeyed from the foundational concepts to the cutting edge of privacy-preserving AI.

Here are the key takeaways from this final chapter:

- **FHE Empowers Privacy-Preserving Analytics:** FHE is revolutionizing data analysis in sensitive sectors like healthcare and finance by enabling computations directly on encrypted data.
- **Private AI Inference is a Game-Changer:** HEIR is crucial for compiling AI models to perform inference on encrypted data, allowing cloud-based AI services to operate without compromising user privacy.
- **Synergy Enhances Privacy:** FHE achieves even greater privacy guarantees when combined with other PETs, such as Differential Privacy and Secure Multi-Party Computation.
- **Significant Challenges Remain:** Performance overhead, the inherent complexity of FHE development, and the computational cost of bootstrapping are key hurdles that require ongoing research and innovation.

- **HEIR is a Catalyst:** As an FHE compiler, HEIR is central to democratizing FHE, abstracting cryptographic complexity for developers, though it is currently an actively developing project (as of 2026-08-18).
- **A Bright Future Ahead:** Hardware acceleration, standardization efforts, and deeper integration with other PETs promise to make private AI with FHE a widespread and practical reality in the coming years.

The vision of leveraging powerful artificial intelligence without sacrificing fundamental privacy is within our grasp. Tools like HEIR are paving the way, and your understanding of these concepts positions you at the forefront of this exciting technological frontier. Keep exploring, keep building, and continue to champion privacy in all your innovations!

References

- HEIR GitHub Repository
- A Guide For HEIR Experiment Evaluation (Original Version)
- Microsoft Research Blog: Homomorphic Encryption Explained
- IBM Cloud Learn: What is Federated Learning?
- NIST: Secure Multi-Party Computation (MPC)

This page is AI-assisted and reviewed. It references official documentation and recognized resources where relevant.