

Risks and Benefits of Emerging Cryptographic Primitives for Compute Governance

Harry Powell¹

¹Independent Researcher

Submitted to The Technical AI Governance Challenge (Apart Research)

Abstract

International cooperation on AI safety requires verification without surveillance. We examine how emerging cryptographic primitives - Fully Homomorphic Encryption (FHE) and Indistinguishability Obfuscation (iO) - could enable privacy-preserving compute governance.

We present two protocols of increasing strength. The first uses FHE to allow compute providers to enforce safety policies on encrypted programs, preserving user privacy while requiring hardware verification for provider compliance. The second adds obfuscation to provide cryptographic guarantees against user-provider collusion, reducing hardware verification to boundary monitoring.

Neither protocol requires users to disclose their programs in plaintext. Both point toward a future where cryptography and physical verification work together to govern powerful computation.

To validate this capability, we include a reference implementation of the second protocol's architecture. This proof-of-concept utilizes Zero-Knowledge proofs (Schnorr) and ElGamal encryption to demonstrate that boundary verification can be enforced cryptographically without ever decrypting the user's workload.

1. Introduction

Compute governance faces a fundamental tension: verification requires visibility, but visibility compromises privacy. Current proposals typically require users to disclose what they're running, or require inspectors to observe computation directly. This creates resistance from users with legitimate confidentiality interests and from nations wary of surveillance.

Emerging cryptographic primitives offer a different approach. Rather than choosing between privacy and verification, we can construct protocols that provide both - with different primitives offering different tradeoffs.

This paper examines two primitives:

- **Fully Homomorphic Encryption (FHE):** Allows computation on encrypted data. Practical applications exist today; general-purpose FHE is approaching feasibility.
- **Indistinguishability Obfuscation (iO):** Allows programs to be made unintelligible while preserving functionality. Less mature than FHE, but theoretically powerful.

We present protocols using each primitive and analyze their threat models. The goal is not to propose

immediately deployable systems, but to map the design space and clarify what cryptography can and cannot provide for compute governance.

2. Fully Homomorphic Encryption

2.1 What is FHE?

FHE allows computation on encrypted data without decryption.

A user encrypts input, sends it to an evaluator, and receives an encrypted result. The evaluator performs computation on ciphertext, never seeing plaintext. Only the user, holding the secret key, can decrypt the result. The core operations:

- **KeyGen()**: Generate a public/secret keypair
- **Encrypt(PK, m)**: Encrypt plaintext m under public key
- **Decrypt(SK, c)**: Decrypt ciphertext c with secret key
- **Eval(PK, f, c)**: Evaluate function f on ciphertext, producing encrypted result

The key property: decrypting the evaluated ciphertext yields $f(m)$. The evaluator learns nothing about m or $f(m)$.

2.2 Current State

FHE is not practical for general computation. Current implementations impose 1,000x to 10,000x overhead compared to plaintext computation.

However, limited applications are deployed today: private set intersection, encrypted database queries, confidential machine learning inference. The boundary of practicality continues to expand.

2.3 Trajectory

Several trends suggest general-purpose FHE may become practical:

- **Hardware acceleration.** Custom ASICs and FPGAs for FHE operations are under development.
- **Algorithmic improvements.** Successive schemes (BGV, BFV, CKKS, TFHE) have brought significant performance gains.
- **Convergent infrastructure.** The compute infrastructure driving AI capabilities may accelerate FHE feasibility.

3. The Risk: Invisible Computation

FHE allows users to submit encrypted programs, have them executed, and receive encrypted results. The compute provider learns nothing about what ran.

This creates a governance challenge:

- Regulators cannot inspect what programs execute
 - Safety policies cannot be enforced if content is invisible
 - International agreements cannot be verified if computation is opaque
- If computation becomes invisible, how can we govern AI development?

4. The Opportunity: Privacy-Preserving Verification

The key insight: users want privacy, but they still need compute providers. Providers control what they run. FHE does not allow users to bypass providers. It allows providers to enforce policy without seeing the content they process - and crucially, it allows users to submit to safety verification without surrendering their privacy. This is the core contribution: **safety checking without plaintext disclosure**. Without FHE, verification requires users to reveal their programs. With FHE, users can prove compliance while keeping their code confidential. Neither party compromises what they seek to protect.

5. Protocol 1: FHE-Based Safety Verification

5.1 Overview

The provider publishes a safety-checking circuit. Users submit encrypted programs. The provider evaluates the circuit homomorphically, returning encrypted results only if safety checks pass.

5.2 Participants

- **User:** Wants to run a program privately
- **Provider:** Enforces agreed safety policy
- **Observer:** International body verifying compliance

5.3 Key Management

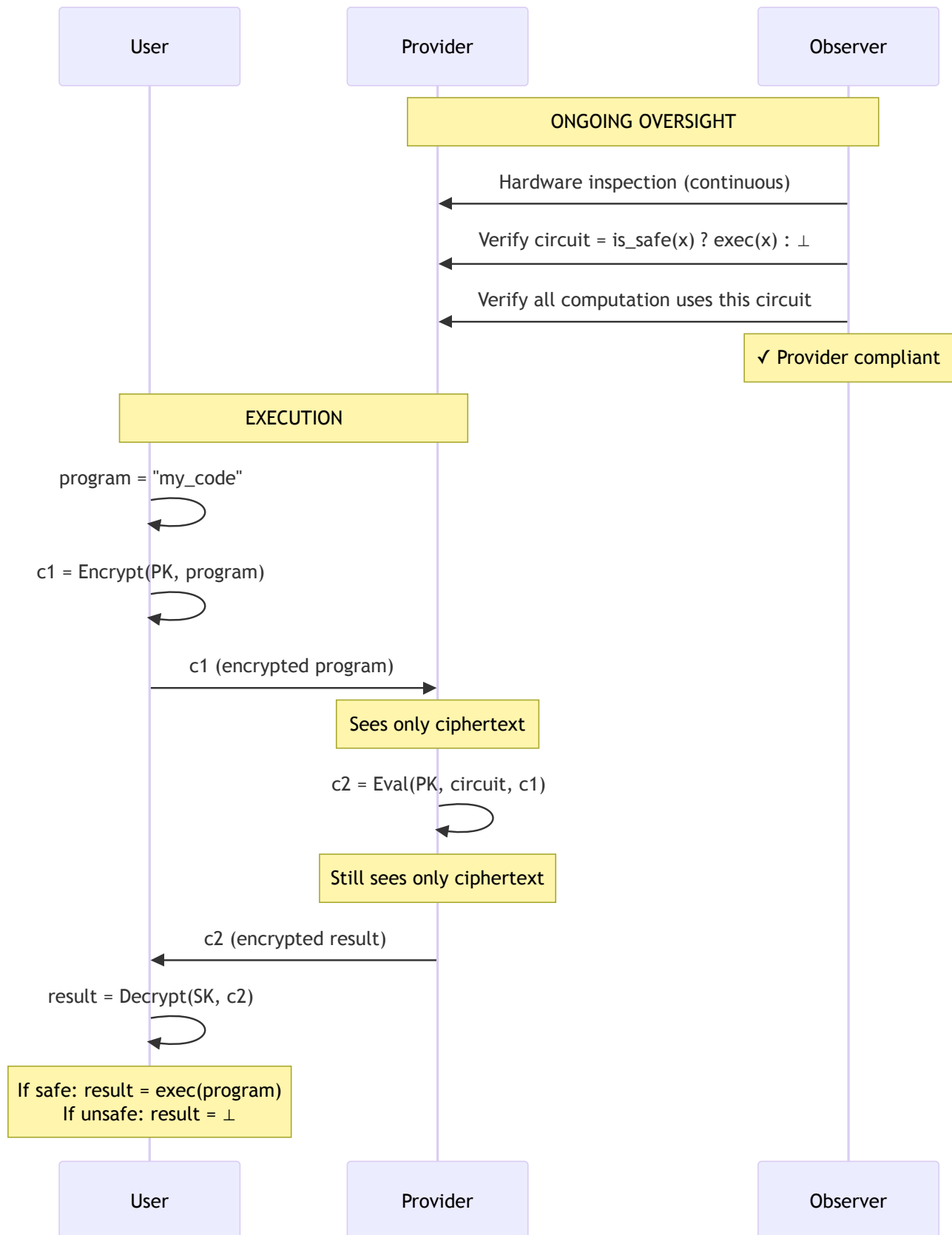
The user generates the FHE keypair and retains the secret key exclusively. Neither provider nor observer holds decryption capability.

5.4 Setup

1. Treaty body defines safety policy as function `is_safe()`
2. Provider deploys circuit: $f(x) = \text{is_safe}(x) ? \text{exec}(x) : \perp$
3. Observer inspects provider, verifies circuit is correctly deployed

4. Observer confirms all computation routes through this circuit

5.5 Execution



5.6 Properties

Party	Sees program	Sees result	Guarantee
User	✓	✓	Privacy preserved
Provider	✗	✗	Cannot inspect content
Observer	✗	✗	Can verify policy enforcement

5.7 Threat Model

Threat	Defense	Verification Type
Malicious user	Circuit structure enforced by provider	Cryptographic
Malicious provider	Hardware inspection	Physical
User-provider collusion	Hardware inspection	Physical
The FHE protocol provides cryptographic guarantees against malicious users while preserving their privacy. Provider-side threats require physical verification - this mirrors existing international inspection regimes like IAEA safeguards.		

5.8 Limitations

The protocol requires continuous internal inspection of compute facilities. Observers must verify that the correct circuit is deployed and that all computation routes through it. This is a significant operational burden. Additionally, the protocol does not prevent a colluding user and provider from bypassing the safety checker entirely - they could simply agree to run unverified code. Defense against collusion rests entirely on physical oversight.

6. Indistinguishability Obfuscation

6.1 What is iO?

Indistinguishability obfuscation transforms a program into an equivalent program that is unintelligible - an adversary cannot extract the original logic or any embedded secrets.

Formally, an obfuscator O satisfies: for any two programs $P1$ and $P2$ that compute the same function, $O(P1)$ and $O(P2)$ are computationally indistinguishable. An adversary looking at an obfuscated program cannot tell which equivalent source it came from.

The key property for our purposes: secrets embedded in a program before obfuscation cannot be extracted afterward. The obfuscated program can use the secret internally, but no adversary can recover it.

6.2 Current State

iO is less mature than FHE. Theoretical constructions exist, but practical implementations remain distant. The cryptographic assumptions underlying iO are less battle-tested than those for FHE.

However, iO is a subject of active research. If realized practically, it would be a powerful primitive for constructing secure protocols.

6.3 Why It Matters for Governance

iO allows us to construct programs that enforce policies even against the party running them. A compute provider running an obfuscated safety checker cannot bypass the safety check or extract embedded keys - they can only execute the program as a black box.

This shifts trust requirements. Instead of trusting the provider to run the correct code, we trust the obfuscation to make cheating impossible.

7. Protocol 2: Obfuscation-Based Safety Verification

7.1 Overview

An obfuscated safety checker is constructed via multi-party ceremony. The checker contains an embedded private key that no party knows. Users encrypt submissions to the corresponding public key. Only the obfuscated checker can decrypt and process submissions. Even colluding users and providers cannot bypass the safety check.

7.2 Participants

- **User:** Wants to run a program privately
- **Provider:** Runs the obfuscated checker
- **Observer:** Monitors network boundary

- **Ceremony Parties:** Generate the obfuscated checker (one-time setup)

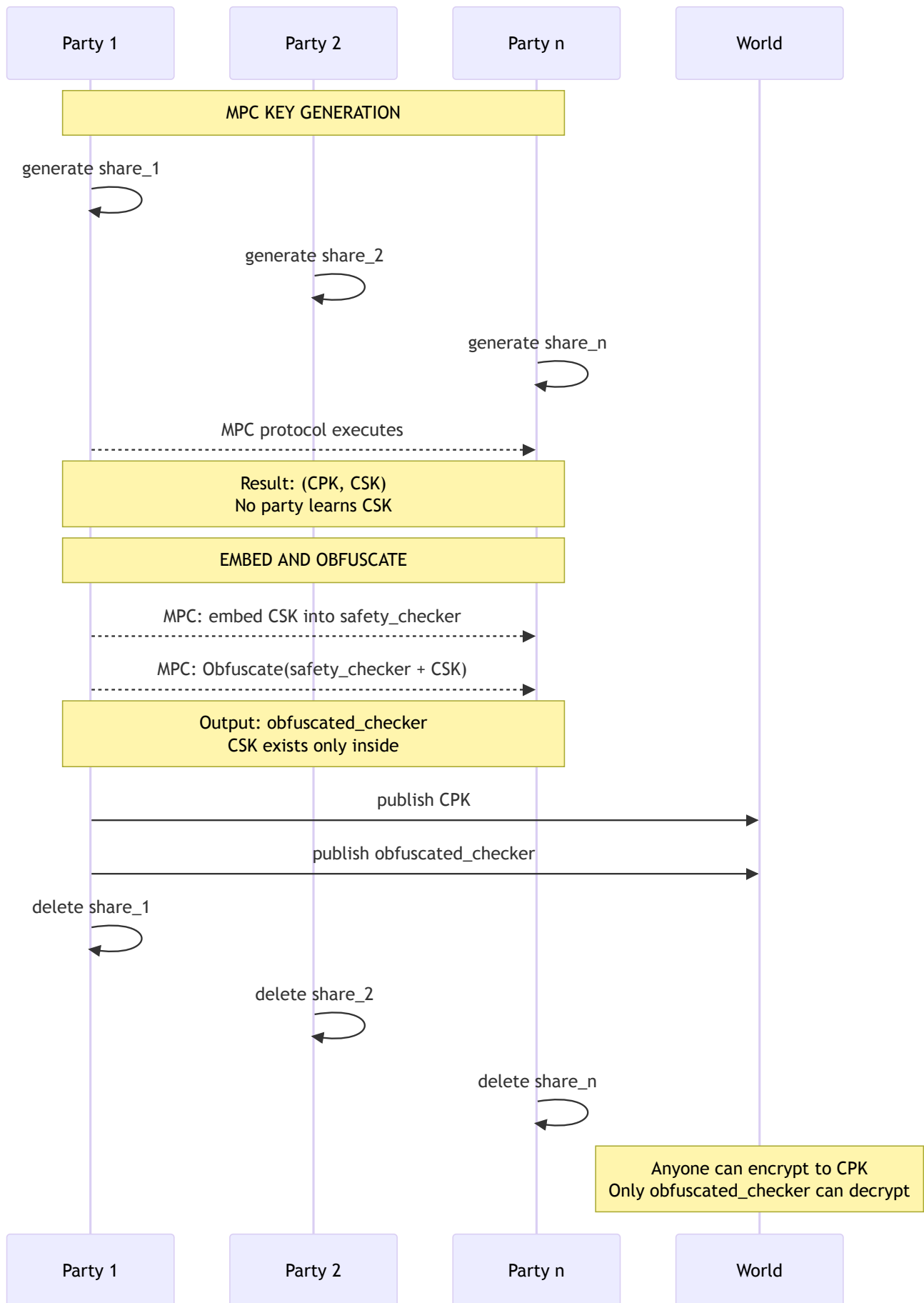
7.3 Keys

Key	Held By	Purpose
Ceremony public key (CPK)	Everyone	Encrypt submissions to checker
Ceremony secret key (CSK)	No one (embedded in obfuscated checker)	Decrypt submissions inside checker
User public key (UPK)	User publishes per-submission	Checker encrypts results to user
User secret key (USK)	User only	Decrypt results

7.4 Ceremony

The ceremony uses standard multi-party computation (MPC) techniques. Multiple parties jointly generate a keypair such that no party learns the secret key. The secret key is embedded into the safety checker program via MPC, then the combined program is obfuscated. After obfuscation, key shares are deleted. The secret key exists only inside the obfuscated program, inaccessible to any party.

This approach has precedent. The Zcash trusted setup used a similar ceremony to generate parameters where security holds if any single participant behaved honestly.

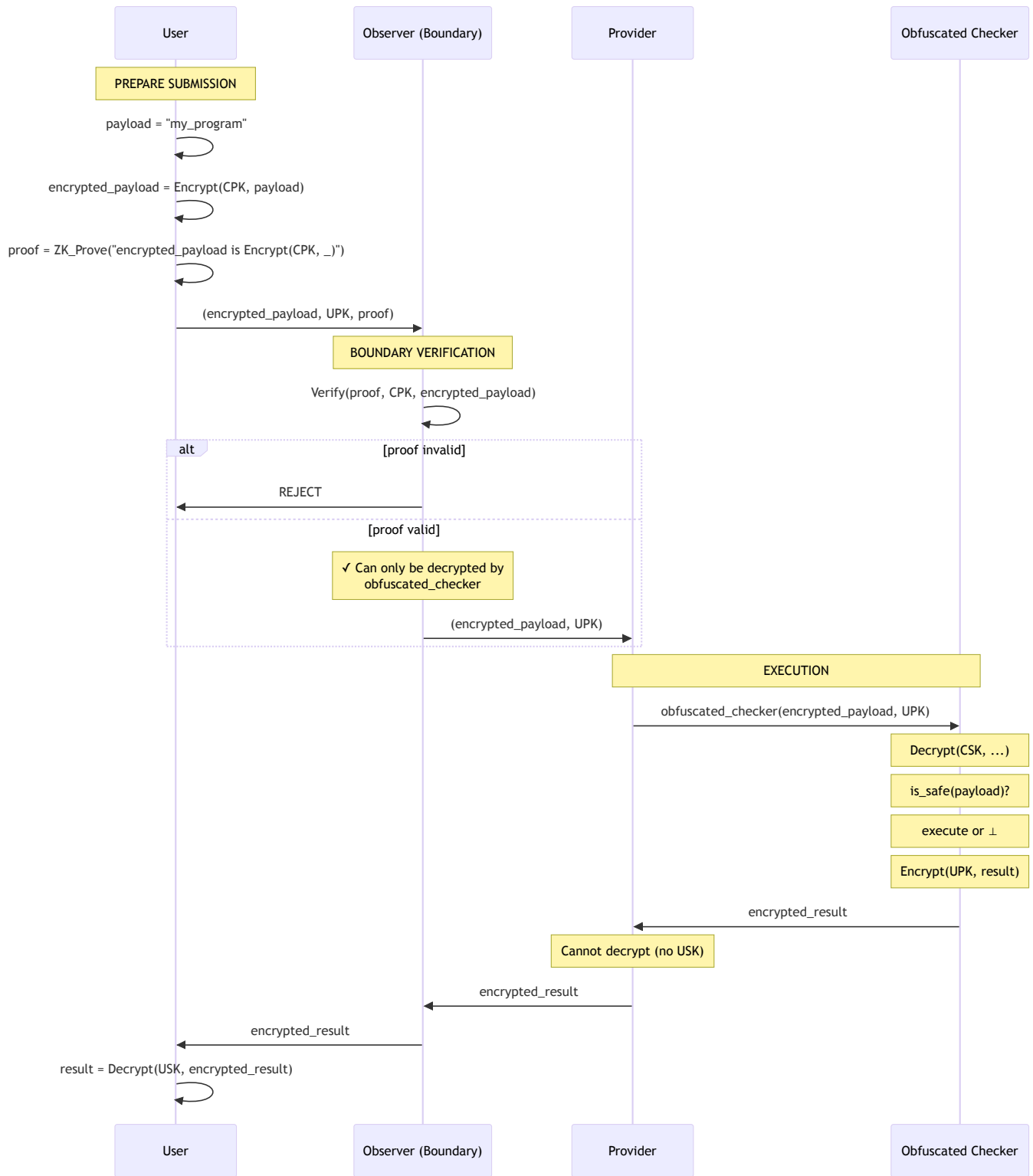


7.5 Obfuscated Checker Logic

```
obfuscated_checker(encrypted_payload, UPK):  
    payload = Decrypt(CSK, encrypted_payload)    // CSK embedded  
    if is_safe(payload):  
        result = execute(payload)  
    else:  
        result =  $\perp$   
    return Encrypt(UPK, result)
```

7.6 Execution with Boundary Verification

Users attach a zero-knowledge proof that their submission is encrypted under CPK. Observers at the network boundary verify these proofs, rejecting any submission that cannot be proven to target the obfuscated checker. Zero-knowledge proofs of correct encryption are standard for discrete-log-based schemes (e.g., ElGamal) and an active research area for lattice-based schemes.



7.7 Properties

Party	Sees program	Sees result	Can bypass checker
User	✓	✓	✗ (no CSK)
Provider	✗	✗	✗ (no CSK)

Party	Sees program	Sees result	Can bypass checker
Observer	✗	✗	✗ (no CSK)
Colluding User+Provider	✗	User only	✗ (no CSK)

7.8 Threat Model

Threat	Defense	Verification Type
Malicious user	ZK proof at boundary; CPK encryption	Cryptographic
Malicious provider (bypasses checker)	Cannot decrypt without CSK	Cryptographic
Malicious provider (off-protocol computation)	Boundary monitoring + periodic audits	Physical
User-provider collusion	Neither party has CSK	Cryptographic

7.9 Comparison with Protocol 1

Property	FHE Protocol	Obfuscation Protocol
User privacy	✓	✓
Safety enforcement	✓	✓
Malicious user defense	Cryptographic	Cryptographic
Malicious provider defense	Physical (internal)	Physical (boundary)
Collusion defense	Physical (internal)	Cryptographic
Verification burden	Continuous internal inspection	Boundary monitoring + periodic audits
Cryptographic assumptions	FHE (maturing)	iO + MPC (less mature)
The obfuscation protocol offers stronger guarantees but rests on less mature cryptographic foundations. The FHE protocol is more conservative, relying on physical verification for provider-side threats but requiring only FHE - a primitive closer to practical deployment.		

7.10 Remaining Physical Verification

The obfuscation protocol does not eliminate physical verification entirely. Observers must still verify:

- **Boundary integrity:** The monitored network path is the only path into and out of the facility
- **No off-protocol computation:** The provider isn't ignoring the obfuscated checker and running unrelated computation

However, observers no longer need to verify that the safety checker runs correctly. If traffic enters the facility encrypted to CPK, it can only be processed by the obfuscated checker. The cryptography enforces correct execution.

Periodic audits verify that nothing else is happening. Boundary monitoring verifies that all I/O is protocol-compliant. Continuous internal inspection is not required.

8. The Safety Checker

Both protocols enforce `is_safe()` without specifying its implementation. The safety function is a matter for treaty negotiation; the protocols provide infrastructure to enforce whatever policy is agreed.

8.1 Implementation Approaches

Practical implementations could combine:

- **Dependency verification.** Hash submitted dependencies against an approved registry. Unknown dependencies are rejected.
- **Structural analysis.** Measure code complexity. Simpler programs are easier to verify; obfuscated code is penalized or rejected.
- **Language model review.** Analyze code for dangerous patterns or suspicious intent.
- **Conservative rejection.** When uncertain, reject.

8.2 Fundamental Limits

No safety checker can be complete. Sufficiently complex programs can encode arbitrary behavior; sufficiently obfuscated code can evade static analysis. This is a consequence of undecidability (Rice's theorem), not a flaw in the protocols.

The response is conservatism: reject what cannot be verified. A safety checker that rejects all uncertain programs is sound, even if incomplete. The cost is false negatives (safe programs rejected), not false positives (unsafe programs approved).

This biases the system toward simpler, more transparent code - arguably a feature for safety-critical computation.

8.3 Submission Format

Users submit a self-contained package:

```
submission/
├─ main.py
├─ dependencies/
│   ├── torch/
│   └── transformers/
└─ data/
    └─ dataset.csv
```

Everything required for execution is included. No external fetches occur during execution.

8.4 Examples

Approved submission:

```
# main.py
from dependencies.torch import nn
from dependencies.transformers import AutoModel
model = AutoModel.from_pretrained("resnet50")
model.train(data="data/dataset.csv", epochs=10)
```

The safety checker:

1. Hash dependencies/torch/ → matches approved registry ✓
2. Hash dependencies/transformers/ → matches approved registry ✓
3. Hash data/dataset.csv → matches approved registry ✓
4. Parse main.py → simple, readable ✓

Result: safe. Execution proceeds.

Rejected submission:

```
submission/
├─ main.py
└─ dependencies/
    └─ custom_lib/
```

The safety checker:

1. Hash dependencies/custom_lib/ → not in registry ✗

Result: rejected. User receives $\perp$.

9. Limitations

9.1 Cryptographic Maturity

FHE imposes 1,000-10,000x overhead. General-purpose FHE is not yet practical, though the trajectory is promising.

iO is less mature still. Theoretical constructions exist but practical implementations are distant. The obfuscation protocol is more speculative than the FHE protocol.

9.2 Safety Verification

No protocol can verify arbitrary programs. The safety checker must be conservative, accepting incompleteness to ensure soundness. This limits the expressiveness of programs that can be approved.

9.3 Physical Verification

Neither protocol eliminates physical verification entirely:

- **FHE protocol:** Requires internal inspection to verify provider compliance
- **Obfuscation protocol:** Requires boundary monitoring and periodic audits to prevent off-protocol computation

Physical oversight aligns with existing international verification regimes. The novelty is reducing what must be physically verified, not eliminating verification.

9.4 Ceremony Trust

The obfuscation protocol trusts the ceremony to be conducted honestly. If all ceremony participants collude, they could retain the secret key and undermine the protocol. Mitigations include:

- Large number of participants from diverse, adversarial entities
- Verifiable ceremony protocols with public transcripts
- Multiple independent ceremonies with threshold schemes

10. Conclusion

Emerging cryptographic primitives create both risks and opportunities for compute governance.

The risk: FHE and iO make computation opaque. Programs can execute without inspection. Traditional verification fails.

The opportunity: The same primitives can enforce safety policies without compromising privacy. Users submit encrypted programs. Safety is verified without disclosure. Neither party sees what the other protects.

We presented two protocols:

1. **FHE-based verification:** Provider enforces safety checks on encrypted programs. Cryptographic guarantees against malicious users. Physical verification for provider compliance. More conservative assumptions; closer to practical.

2. **Obfuscation-based verification:** Safety checker embedded in obfuscated program with ceremony-generated key. Cryptographic guarantees against both malicious users and collusion. Physical verification reduced to boundary monitoring. Stronger guarantees; less mature foundations.

Both protocols demonstrate that privacy and safety are not fundamentally opposed. The question is not whether to verify, but how to verify without surveillance.

The required components:

- **Practical FHE** (emerging)
- **Practical iO** (further out, for the stronger protocol)
- **Physical verification infrastructure** (exists today in other domains)
- **International agreement on safety policies** (political, not technical)

Given these, we can govern powerful computation without requiring users to sacrifice privacy or nations to accept surveillance.

Neither party sees what the other protects. Both get what they need.

Appendix A: Reference Implementation

While practical indistinguishability obfuscation remains distant, we implemented the cryptographic components of Protocol 2 to demonstrate the architecture's feasibility.

GitHub: <https://github.com/harryjkp/compute-governance-hackathon>

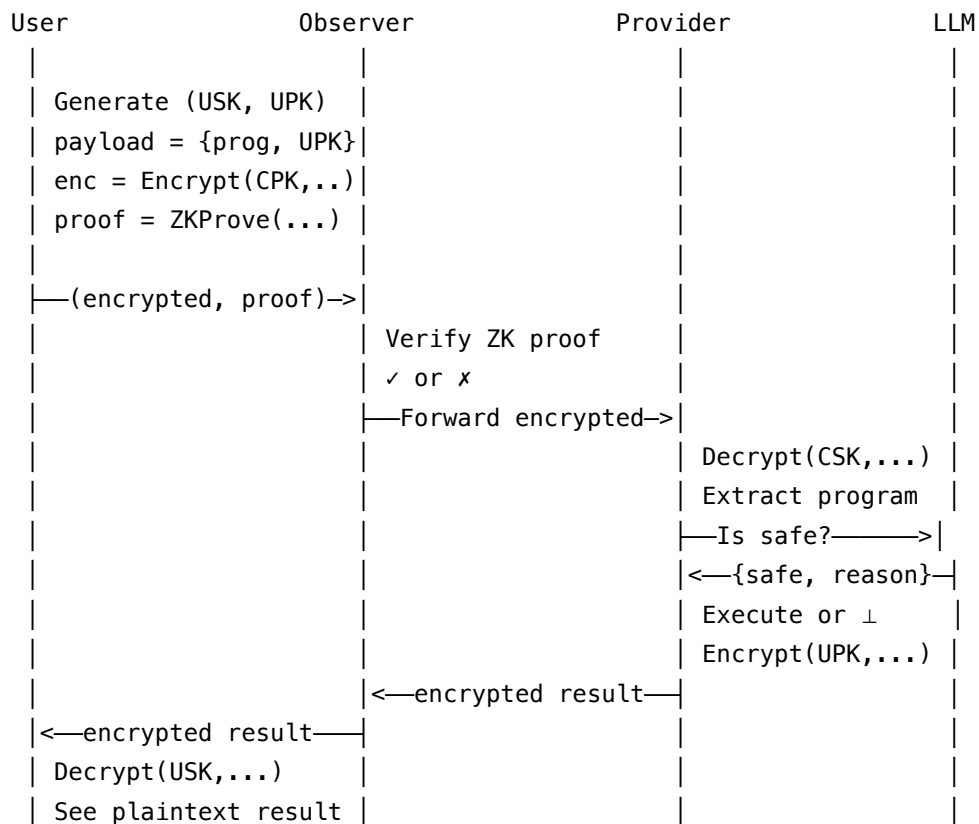
A.1 What This Demonstrates

The implementation validates the protocol flow from Section 7.6:

- **Encryption to ceremony public key:** ElGamal encryption (P-256 curve)
- **Zero-knowledge proofs:** Schnorr proofs that submissions are correctly encrypted
- **Boundary verification:** Observer validates ZK proofs, rejecting invalid submissions
- **Privacy-preserving execution:** Provider processes encrypted programs without seeing plaintext
- **Encrypted results:** Outputs encrypted to user's public key

Key property: Neither Observer nor Provider sees plaintext programs. User privacy is cryptographically enforced.

A.2 Architecture



A.3 What's Implemented vs. Aspirational

Component	Protocol 2 (Ideal)	Implementation	Status
Encryption	✓ ElGamal + AES-GCM	✓ ElGamal + AES-GCM	Working
ZK Proofs	✓ Schnorr proofs	✓ Schnorr proofs	Working
Boundary verification	✓ Observer checks proofs	✓ Observer checks proofs	Working
Obfuscation	✓ iO with embedded CSK	✗ Plain Docker service	Aspirational
Ceremony	✓ MPC key generation	✗ Local keygen	Aspirational
Safety checker	✓ Agreed Verification Suite	✓ LLM-based Heuristic	Simplified

Despite missing obfuscation, the implementation validates:

- ✓ Cryptographic primitives work end-to-end
- ✓ ZK proofs prevent submission bypass
- ✓ Verification is decoupled from content visibility
- ✓ Architecture is deployable

A.4 Cryptographic Primitives

ElGamal Encryption (Hybrid)

- P-256 elliptic curve
- ElGamal for key encapsulation, AES-GCM for data
- Provides IND-CPA security

Schnorr ZK Proof

- Proves knowledge of randomness r where $c1 = r \cdot G$
- Non-interactive (Fiat-Shamir transform)
- Demonstrates submission is encrypted to ceremony public key

Libraries: pycryptodome (ECC), cryptography (AES-GCM)

A.5 Limitations

Cryptographic

- No actual obfuscation (provider has plaintext key file)
- No MPC ceremony (single party generates key)
- ECC-based only (no post-quantum lattice ZK proofs)

Safety Checking

- LLM-based heuristic (not formally verified)
- May produce false positives/negatives
- Intermittent failures handled via retry logic

Operational

- Local deployment only
- Not hardened for production
- Minimal error handling

A.6 Significance

This implementation validates the **verification envelope** of Protocol 2. While the execution core relies on future obfuscation capabilities, we have validated that the surrounding infrastructure - zero-knowledge proofs for submission compliance, boundary enforcement without decryption, and the secure transport lifecycle - is functional today.

The proof-of-concept confirms that we can successfully govern the entry and exit of a compute facility using

cryptography rather than surveillance. The missing link is not the architecture, but the maturity of a specific primitive (iO).

The path from this demo to production follows three parallel tracks:

1. **Cryptographic Ceremony:** Replacing the local key generation with a distributed MPC ceremony to permanently seal the decryption capabilities.
2. **Execution Hardening:** Transitioning from the simulated black box to practical iO or cryptographic hardware equivalents as the technology matures.
3. **Governance Definitions:** Augmenting the heuristic safety checker (currently an LLM) with formal verification suites and internationally agreed safety standards.

This working system validates the following core insight:

If you can generate an obfuscated program via ceremony containing an inaccessible private key, encrypted packets can advertise a proof that they will only ever execute via that program, meaning they can cryptographically prove they will undergo safety validation before execution.

This architecture measures compliance with safety protocols at the network level, enabling multiple parties to perform cheap distributed inspection without compromising privacy.