Apertures: Coordinated Partial Evaluation for Distributed Computation

Alexander Towell
Department of Computer Science
Southern Illinois University Edwardsville
atowell@siue.edu

Abstract

We present *apertures*, a coordination mechanism for distributed computation based on partial evaluation with explicit holes. Apertures (denoted `?variable`) represent unknown expressions that enable pausable and resumable evaluation across multiple parties. Unlike security-focused approaches, we make no claims about privacy preservation—apertures leak information through program structure and evaluation results. Instead, we position apertures as a lightweight coordination primitive for scenarios where parties can share computation structure but need to control when and where data is introduced. We demonstrate practical coordination patterns including multi-party progressive refinement and local computation workflows, where untrusted servers can optimize expressions without seeing private inputs or outputs. Our C++ implementation shows apertures add minimal overhead (2-5%) compared to direct evaluation while enabling novel distributed computation patterns. We are explicit about limitations: apertures are unsuitable for adversarial settings and should not be used when information leakage is unacceptable.

1 Introduction

Distributed computation often requires coordination between parties with different capabilities and trust levels. A common pattern involves untrusted servers with computational resources and clients with private data. Current solutions typically fall into two extremes: full encryption with massive overhead, or complete data sharing with no privacy.

We introduce *apertures*, a coordination mechanism that enables a middle ground through partial evaluation. An aperture, written `?variable`, is a hole in an expression representing an unknown value or subexpression. Programs with apertures can be partially evaluated, optimized, and transformed by parties that don't know the aperture values. When apertures are eventually filled, evaluation resumes from the partially evaluated state.

Critical disclaimer: Apertures are *not* a security mechanism. They leak information through program structure, evaluation patterns, and algebraic relationships. We make no privacy claims. Instead, apertures provide a coordination primitive for distributed systems where parties need to control the timing and location of data introduction while sharing computational structure.

Consider this local computation pattern:

```
;; Server optimizes without data or results
(let ((plan (server-optimize ?query)))
  (client-execute plan ?local-data))
;; Results stay local
```

The server can optimize the query structure without seeing the actual query parameters or the local data. The client benefits from server-side optimization while keeping both inputs and outputs private. However, the server learns the query structure and optimization patterns, which may reveal information about the workload.

Our contributions:

- A simple calculus for partial evaluation with holes
- Coordination patterns for distributed computation

- Explicit characterization of when apertures should *not* be used
- Performance evaluation showing 2-5% overhead for coordination

2 The Aperture Language

2.1 Core Syntax

We define a minimal Lisp-like language with apertures:

$$e ::= v \mid x \mid \texttt{?}h \mid (e\ e^*) \mid (\lambda\ (x^*)\ e) \tag{1}$$
$$\mid (\text{let}\ ((x\ e)^*)\ e) \mid (\text{if}\ e\ e\ e) \tag{2}$$
$$v ::= \text{nil} \mid n \mid s \mid \text{bool} \mid \text{list} \tag{3}$$

Where v ranges over values, x over variables, h over hole identifiers, and n, s over numbers and strings respectively. The key addition is $\texttt{?}h$, representing an aperture (hole).

Terminology clarification: While we write `?variable` for readability, apertures are not restricted to atomic values. An aperture can be filled with any expression, making them "unknown expressions" in the partial evaluation sense.

2.2 Evaluation Semantics

We define evaluation contexts and partial evaluation:

$$E ::= [\cdot] \mid (E\ e^*) \mid (v\ v^*\ E\ e^*) \tag{4}$$
$$\mid (\text{let}\ ((x\ v)^*\ (x\ E)\ b^*)\ e) \tag{5}$$
$$\mid (\text{if}\ E\ e\ e) \tag{6}$$

Standard evaluation rules apply, with apertures blocking evaluation:

$$E[\texttt{?}h] \to_p E[\texttt{?}h] \quad \text{(aperture blocks)} \tag{7}$$
$$E[(+\ n_1\ n_2)] \to n_1 + n_2 \quad \text{(arithmetic)} \tag{8}$$
$$E[(+\ \texttt{?}h\ n)] \to_p (+\ \texttt{?}h\ n) \quad \text{(partial)} \tag{9}$$

The key insight: evaluation proceeds until blocked by apertures, creating partially evaluated expressions that preserve computation structure.

2.3 Hole Filling

Apertures are filled via substitution:

$$\text{fill}(e, h, e') = e[e'/\texttt{?}h]$$

After filling, evaluation can resume:

$$\text{eval}(\text{fill}(e, h, v)) \to \text{eval}(e[v/\texttt{?}h])$$

Multiple apertures can be filled incrementally, enabling progressive refinement across multiple parties.

2.4 Algebraic Simplification

During partial evaluation, we apply algebraic simplifications that preserve apertures:

$$(* \ 0 \ ?x) \to_p 0 \tag{10}$$
$$(+ \ 0 \ ?x) \to_p ?x \tag{11}$$
$$(* \ 1 \ ?x) \to_p ?x \tag{12}$$
$$(\text{if true } e_1 \ e_2) \to_p e_1 \tag{13}$$

These simplifications reduce expression size while maintaining evaluation semantics. However, they also leak information—knowing that $(* \ ?x \ e) \to 0$ reveals that $e = 0$.

3 Coordination Patterns

Apertures enable several coordination patterns for distributed computation:

3.1 Progressive Refinement

Multiple parties can incrementally fill apertures:

```
;; Initial expression with multiple holes
(let ((query (select ?table
                (where (and (> ?col1 ?val1)
                            (< ?col2 ?val2))))))

  ;; Party A knows table and column names
  (fill query {table: "users",
               col1: "age",
               col2: "salary"})

  ;; Party B knows value constraints
  (fill query {val1: 18, val2: 100000})

  ;; Execute when all holes filled
  (execute query))
```

Each party contributes their knowledge without seeing others' contributions until execution.

3.2 Local Computation Pattern

The most defensible use case: untrusted optimization with local execution:

```
;; Client has private data
(define private-data (load-sensitive))

;; Server optimizes generic computation
(define (optimize-computation expr)
  ;; Algebraic simplification
  ;; Common subexpression elimination
  ;; Partial evaluation of known parts
  (simplify expr))

;; Client sends structure, not data
```

```
(define template
  (sum (map (lambda (x)
              (* ?weight (f x)))
            ?data)))

;; Server optimizes without seeing data
(define optimized
  (server-optimize template))
;; Returns: (let ((w ?weight))
;;            (sum (map (lambda (x)
;;                        (* w (f x)))
;;                      ?data)))

;; Client evaluates locally
(local-eval (fill optimized
                  {weight: 0.5,
                   data: private-data}))
```

Benefits:

- Server CPU used for optimization
- Data never leaves client
- Results stay local
- Reduced leakage channels

Limitations:

- Server learns computation structure
- Optimization patterns may leak workload information
- No protection against malicious servers

3.3 Speculative Compilation

Servers can compile multiple specializations:

```
;; Template with configuration hole
(define (process-data config data)
  (case ?config
    [(fast) (quick-process data)]
    [(accurate) (slow-process data)]
    [(balanced) (hybrid-process data)]))

;; Server compiles all branches
(define compiled
  (compile-all-paths template))

;; Client selects path at runtime
(execute compiled {config: 'accurate})
```

The server prepares optimized code paths without knowing which will be used.

4 Implementation Highlights

Our C++ implementation provides:

- S-expression parser with aperture syntax

- Partial evaluator with algebraic simplification
- Hole tracking and filling mechanism
- Fluent API for natural expression construction

Key design choices:

- Shared pointer memory management
- Visitor pattern for evaluation
- Variant-based value representation
- Stack-based evaluation with continuation frames

4.1 Performance Characteristics

We measured overhead compared to direct evaluation:

Operation	Direct (μs)	With Apertures (μs)	Overhead
Arithmetic (1000 ops)	127	131	3.1%
List processing	89	93	4.5%
Conditionals	43	44	2.3%
Lambda application	156	164	5.1%

Apertures add minimal overhead (2-5%) while enabling distributed coordination. The overhead comes from:

- Hole checking during evaluation
- Maintaining partial evaluation state
- Expression reconstruction after simplification

Partial evaluation can actually *improve* performance when expressions are reused with different aperture values, as the partially evaluated form serves as a template.

5 Case Study: Query Optimization

Consider a distributed database scenario where a client wants to run complex analytical queries on private data. Traditional approaches either:

1. Send data to server (privacy loss)
2. Send query to data (lose server optimization)
3. Use homomorphic encryption (1000× overhead)

With apertures, we enable a fourth option:

```
;; Client query template
(define query-template
  (select ?private-table
    (where (and (> profit ?threshold)
                (in region ?regions)))
    (group-by department)
```

```
    (having (> (count *) ?min-size))))

;; Server optimizes structure
(define optimized-plan
  (server-optimize query-template))
;; Server can:
;; - Reorder predicates
;; - Plan aggregation strategy
;; - Choose join algorithms
;; - Prepare indexes

;; Client executes locally
(define results
  (local-execute
    (fill optimized-plan
          {private-table: my-data,
           threshold: 1000000,
           regions: ["NA", "EU"],
           min-size: 10})))
```

The server provides optimization expertise without seeing:

- Actual table data
- Specific threshold values
- Selected regions
- Query results

However, the server learns:

- Query structure and complexity
- Types of operations performed
- Optimization applicability

This tradeoff is acceptable when:

- Query patterns are not sensitive
- Optimization benefits outweigh leakage
- Server is honest-but-curious, not malicious

6 Related Work

Partial evaluation [2]: Apertures extend partial evaluation with explicit holes for multi-party coordination. Unlike traditional partial evaluation which requires all static values upfront, apertures allow incremental filling.

Staged computation [4]: Multi-stage programming separates computation phases. Apertures provide a simpler mechanism focused on coordination rather than performance.

Homomorphic encryption [1]: FHE provides cryptographic privacy but with 1000-10000× overhead. Apertures trade privacy for practical performance.

Secure multi-party computation [5]: MPC protocols provide strong security guarantees through cryptographic protocols. Apertures offer a lightweight alternative for non-adversarial settings.

Information flow control [3]: Type systems can track information flow statically. Apertures provide dynamic coordination without static guarantees.

7 Limitations: When NOT to Use Apertures

We explicitly enumerate scenarios where apertures are inappropriate:

7.1 Adversarial Settings

Apertures provide **no protection** against malicious parties. An adversary can:

- Infer aperture values from program structure
- Use algebraic relationships to constrain possibilities
- Exploit side channels in evaluation patterns
- Inject malicious code during partial evaluation

7.2 High-Sensitivity Data

When data sensitivity is high, apertures leak too much:

- Medical records: Disease patterns visible in query structure
- Financial data: Trading strategies revealed by operations
- Personal data: Behavioral patterns in access patterns

7.3 Regulatory Compliance

Apertures cannot provide guarantees required by:

- GDPR: No verifiable data protection
- HIPAA: Insufficient privacy safeguards
- Financial regulations: No audit trail of data access

7.4 Information Leakage Examples

Consider this expression:

```
(if (> ?age 65)
    (discount 0.2)
    (if (< ?age 18)
        (discount 0.1)
        (no-discount)))
```

Even without knowing `?age`, observers learn:

- System has age-based discrimination
- Exact discount tiers
- Business logic for pricing

Or this query:

```
(select users
  (where (and (= status ?s)
              (> last-login ?date)
              (in department ?dept))))
```

Structure alone reveals:

- Organization has departments
- Tracks login times
- Has user statuses
- Performs temporal queries

7.5 When Apertures ARE Appropriate

Apertures work well when:

- Computation structure is not sensitive
- Parties are honest-but-curious, not adversarial
- Performance matters more than perfect privacy
- Coordination benefits outweigh leakage risks
- Local computation can limit result leakage

Examples of good use cases:

- Scientific computing with public algorithms
- Optimization of non-sensitive business logic
- Collaborative debugging and testing
- Educational and demonstration systems

8 Conclusion

Apertures provide a lightweight coordination mechanism for distributed computation through partial evaluation with holes. We make no security claims—apertures leak information through structure and evaluation patterns. Instead, they offer a practical tool for scenarios where parties need to coordinate computation while controlling when and where data is introduced.

The key insight is that many real-world scenarios don't require cryptographic privacy but do need coordination primitives. Apertures fill this gap with minimal overhead (2-5%) while enabling patterns like local computation with remote optimization.

Future work includes:

- Quantifying information leakage formally
- Combining apertures with oblivious algorithms
- Exploring apertures in specific domains (databases, ML)
- Building practical systems using aperture coordination

We hope apertures contribute to the discussion of practical coordination mechanisms that acknowledge rather than obscure their limitations. Not every distributed computation needs cryptographic security, but all need clear understanding of their tradeoffs.

References

[1] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of STOC*, pages 169–178. ACM, 2009.

[2] Neil D Jones, Carsten K Gomard, and Peter Sestoft. Partial evaluation and automatic program generation. Prentice Hall, 1993.

[3] Andrei Sabelfeld and Andrew C Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.

[4] Walid Taha and Tim Sheard. Multi-stage programming: Axiomatization and type safety. *Proceedings of ICFP*, 1997.

[5] Andrew C Yao. Protocols for secure computations. In *Proceedings of FOCS*, pages 160–164. IEEE, 1982.