DIRAM (Directed Instruction RAM)

DIRAM (Directed Instruction RAM)

DIRAM is software. But it dreams of hardware.

DIRAM is not your everyday memory manager. It's a software emulator for a future hardware memory system—one that doesn't just hold data, but anticipates, governs, and introspects it. This is **predictive, cryptographically-aware, zero-trust RAM** that understands its own state, its future allocation paths, and the AI it serves.

🚨 Mission Statement

We built DIRAM as a placeholder for a real, physical Directed RAM architecture—a memory standard for intelligent systems. Not RAM as we know it. Not dumb, passive, addressable storage. But memory that:

- **Looks ahead**: Using predictive allocation strategies to prepare memory for algorithms before they're even called
- **Governs itself**: Enforcing cryptographic constraints and zero-trust boundaries at the allocation level
- **Thinks about thinking**: Providing introspective capabilities for AI systems to understand their own memory patterns

Use this software to simulate the behavior of DRAM. But know this: the real goal is to build it in silicon.

🧠 What Makes DIRAM Different?

DIRAM (Directed Instruction RAM) is not another LRU cache with trust issues. It introduces:

- 🎯 **Predictive Allocation**
  Anticipates future memory needs using asynchronous promises and lookahead strategies

- 🧬 **Governed Eviction**
  Enforces runtime memory constraints (`ε(x) ≤ 0.6`)—allocations are audited and evicted based on behavioral rules, not just age or usage

- 🧠 **Traceable Memory Dynamics**
  Tracks every allocation with SHA-256 receipts and enforces zero-trust boundaries, providing full cryptographic traceability

- 🥷 **Fork-Safe Detached Execution**
  Supports background operation with audit logging (`alloc_trace.log`), interactive REPL, and telemetry for real-time introspection

DIRAM transforms memory management into a predictive, auditable, and cryptographically secure process—ideal for systems demanding intelligent, zero-trust allocation.

Project Status

DIRAM is currently in **emulation mode**:

- ☑ **Software emulator complete** (CLI, REPL, traceable alloc/free, detached processes)
- **Hardware prototype NOT built** (awaiting silicon partners)
- **Suitable for algorithm testing, AI model memory shaping, and simulation**
- **Active research** into hardware memory cell design for predictive allocation

Table of Contents

Features

Current Software Emulation

- **Cryptographic Memory Tracing**: SHA-256 receipts for all allocations
- **Heap Constraint Enforcement**: Sinphasé governance with ε(x) ≤ 0.6 constraint
- **Zero-Trust Memory Boundaries**: Cryptographically enforced isolation
- **Detached Daemon Mode**: Background operation with comprehensive logging
- **Enhanced Error Indexing**: Telemetry-driven error tracking and recovery
- **Memory Space Isolation**: Named memory spaces with configurable limits
- **REPL Interface**: Interactive memory allocation and inspection

Future Hardware Features (Specification)

- **Hardware-accelerated predictive allocation**
- **On-chip cryptographic receipt generation**
- **Physical memory cell audit trails**
- **AI-optimized access patterns**
- **Zero-copy predictive caching**

Architecture

DIRAM implements a multi-layer memory management system that models future hardware behavior:

```
┌─────────────────────────────────────┐
│     AI Application Layer            │ ← Future: Direct hardware API
├─────────────────────────────────────┤
│   Predictive Allocation Engine      │ ← Future: Hardware accelerated
│  (Promises & Lookahead Cache)       │
├─────────────────────────────────────┤
│    Enhanced Feature Allocation      │
│  (Error Indexing & Governance)      │
├─────────────────────────────────────┤
│      Core Traced Allocation         │ ← Future: On-chip SHA engine
│    (SHA-256 Receipt Generation)     │
├─────────────────────────────────────┤
│      Heap Event Constraints         │ ← Future: Hardware enforced
│        (ε(x) ≤ 0.6 Enforcement)     │
└─────────────────────────────────────┘
          ↓
    [Future Hardware Layer]
```

Installation

Prerequisites

- GCC or compatible C compiler
- POSIX-compliant system (Linux, macOS, BSD)
- GNU Make
- pthread support

Building from Source

```
git clone https://github.com/obinexus/diram.git
cd diram
make clean
make
```

Installation

```
# System-wide installation (requires privileges)
sudo make install

# Custom prefix installation
make install PREFIX=$HOME/.local
```

Usage

Basic Commands

```
# Initialize DIRAM with standard governance
diram --init

# Run with tracing enabled
diram --trace

# Start interactive REPL
diram --repl

# Run in detached daemon mode
diram --detach -c /path/to/config.drc
```

Detached Mode Example

```
# Start DIRAM daemon with custom configuration
diram --detach --config production.drc --trace

# Logs will be written to:
# logs/diram.out.log - Standard output
# logs/diram.err.log - Error output
# logs/alloc_trace.log - Allocation traces (if enabled)
```

Configuration

DIRAM uses a hierarchical configuration system with `.dramrc` files that supports both simple key-value pairs and structured sections for advanced features:

Configuration File Format

```
# ~/.dramrc or project-local .dramrc
# Basic Configuration
memory_limit=2048       # Memory limit in MB
memory_space=production # Named memory space
trace=true              # Enable allocation tracing
log_dir=logs           # Log directory path

# Heap constraint configuration
max_heap_events=3      # Maximum allocations per epoch

# Process isolation
detach_timeout=30      # Daemon timeout in seconds
pid_binding=strict     # Fork safety enforcement

# Memory protection
guard_pages=true       # Enable guard pages
canary_values=true     # Enable canary values
aslr_enabled=true      # Address space randomization
```

```
# Zero-trust configuration
zero_trust=true        # Enable zero-trust boundaries
memory_audit=true      # Enable audit trail

# Telemetry settings
telemetry_level=2      # 0=disabled, 1=system, 2=opcode-bound
telemetry_endpoint=/var/run/diram/telemetry.sock

# Advanced sections for async and resilience features
[async]
enable_promises=true
default_timeout_ms=10000
max_pending_promises=100
lookahead_cache_size=1024

[detach]
enable_detach_mode=true
log_async_operations=true
persist_promise_receipts=true

[resilience]
retry_on_transient_failure=true
max_retry_attempts=3
exponential_backoff=true
```

Example: diram.drc Configuration File

The project includes a comprehensive example configuration file (`diram.drc`) that demonstrates all available options:

```
# DIRAM Configuration File
# OBINexus Project - Directed Instruction RAM

# Memory Configuration
memory_limit=6144      # 6GB in MB
memory_space=userspace # Named memory space identifier

# Tracing Configuration
trace=true             # Enable SHA-256 receipt generation

# Logging Configuration
log_dir=logs           # Directory for detached mode logs

# Heap Constraint Configuration (Sinphasé Governance)
# ε(x) ≤ 0.6 constraint enforced at runtime
max_heap_events=3      # Maximum allocations per command epoch

# Process Isolation Settings
detach_timeout=30      # Seconds before detached process self-terminates
pid_binding=strict     # Enforce strict PID binding for fork safety
```

```
# Memory Protection Flags
guard_pages=true      # Enable guard pages for boundary protection
canary_values=true    # Enable canary values for overflow detection
aslr_enabled=true     # Address Space Layout Randomization

# Telemetry Configuration
telemetry_level=2     # 0=disabled, 1=system, 2=opcode-bound
telemetry_endpoint=/var/run/diram/telemetry.sock

# Zero-Trust Memory Policy
zero_trust=true       # Enable zero-trust memory boundaries
memory_audit=true     # Enable memory audit trail

[async]
enable_promises=true
default_timeout_ms=10000
max_pending_promises=100
lookahead_cache_size=1024

[detach]
enable_detach_mode=true
log_async_operations=true
persist_promise_receipts=true

[resilience]
retry_on_transient_failure=true
max_retry_attempts=3
exponential_backoff=true
```

Configuration Hierarchy

DIRAM loads configuration in the following order, with later sources overriding earlier ones:

1. System-wide: `/etc/diram/config.dram`
2. User home: `~/.dramrc`
3. Project local: `./.dramrc`
4. Command line: `-c <file>`
5. Environment: `DIRAM_CONFIG=<file>`

Runtime Configuration

The REPL provides a `config` command to inspect and modify configuration at runtime:

```
diram> config
DIRAM Configuration:
  Memory Configuration:
    memory_limit: 6144 MB
    memory_space: userspace
  Tracing:
    trace_enabled: yes
    log_dir: logs
```

```
  Heap Constraints:
    max_heap_events: 3
    epsilon: 1.0 (ε = events/max)
  Process Isolation:
    detach_timeout: 30 seconds
    pid_binding: strict
  Memory Protection:
    guard_pages: enabled
    canary_values: enabled
    aslr_enabled: enabled
  Telemetry:
    telemetry_level: 2
    telemetry_endpoint: /var/run/diram/telemetry.sock
  Zero-Trust Policy:
    zero_trust: enabled
    memory_audit: enabled
```

Configuration API

For programmatic access, DIRAM provides a comprehensive configuration API:

```c
// Initialize configuration with defaults
diram_config_init();

// Load configuration from file
diram_config_load_file("custom.dramrc", CONFIG_SOURCE_LOCAL);

// Set individual values
diram_config_set_value("memory_limit", "8192");
diram_config_set_value("trace", "true");

// Get configuration values
const char* space = diram_config_get_value("memory_space");

// Validate configuration
if (!diram_config_validate()) {
    fprintf(stderr, "Config error: %s\n", diram_config_get_errors());
}

// Save current configuration
diram_config_save("backup.dramrc");
```

CLI Reference

REPL Commands

When running in REPL mode (`diram --repl`):

```
Commands:
  alloc <size> <tag>    Allocate traced memory
  free <addr>           Free allocated memory
  trace                 Show allocation trace
  config                Show current configuration
  exit/quit             Exit REPL
```

Example REPL Session

```
$ diram --repl --trace
DIRAM REPL v1.0.0
Type 'help' for commands, 'exit' to quit

diram> alloc 1024 user_buffer
Allocated 1024 bytes at 0x7f8a2c001000 (SHA: 3d4f2c8a9b6e1f...)

diram> trace
Active allocations:
  0x7f8a2c001000: 1024 bytes, tag=user_buffer, SHA=3d4f2c8a9b6e1f...

diram> config
Current configuration:
  Memory limit: 2048 MB
  Memory space: default
  Trace enabled: yes
```

Future REPL Enhancements

The REPL will soon support advanced memory operations:

```
diram> set left_operand 0x560d2a496f10
diram> set right_operand 0x560d2a497f90
diram> multiply left_operand right_operand result
diram> get result
Value at 0x560d2a499010: <computed value>
```

These features will enable real-time memory experiments and cryptographic memory workflows.

Memory Governance

Heap Event Constraints

DIRAM enforces the Sinphasé governance constraint $\varepsilon(x) \leq 0.6$:

- Maximum 3 heap events per command epoch
- Automatic epoch detection and counter reset
- Constraint violations result in allocation deferral

Zero-Trust Enforcement

Memory boundaries are cryptographically enforced:

```
// Each allocation generates a cryptographic receipt
typedef struct {
    void* base_addr;
    size_t size;
    uint64_t timestamp;
    char sha256_receipt[65];
    uint8_t heap_events;
    pid_t binding_pid;
} diram_allocation_t;
```

Error Index Categories

DIRAM tracks and categorizes errors for governance:

- `0x1001`: Heap constraint violation (ε(x) > 0.6)
- `0x1002`: Memory exhausted condition
- `0x1003`: PID mismatch (fork safety)
- `0x1004`: Zero-trust boundary breach
- `0x1005`: SHA-256 verification failure

Development

Project Structure

```
diram/
├── include/
│   └── diram/
│       └── core/
│           └── feature-alloc/
│               ├── alloc.h
│               └── feature_alloc.h
├── src/
│   ├── cli/
│   │   └── main.c
│   └── core/
│       └── feature-alloc/
│           ├── alloc.c
│           └── feature_alloc.c
├── tests/
├── examples/
├── Makefile
├── diram.drc
└── README.md
```

Building Debug Version

```
make clean
make DEBUG=1
```

Running Tests

```
make test
```

Static Analysis

```
make analyze
```

Future Hardware Vision

We envision a DRAM chip that:

- **Performs predictive allocation at the hardware level**: Using AI-driven access pattern analysis
- **Embeds audit trails in physical memory cells**: Each cell contains its own cryptographic history
- **Implements zero-trust at the transistor level**: Hardware-enforced process isolation
- **Optimizes for AI workflows**: Heap-like operations with O(1) access for neural network memory patterns
- **Provides hardware introspection**: Memory that reports its own state and health

Hardware Specification Goals

- **Memory Cell Design**: 8nm process with embedded SHA-256 engine per memory bank
- **Predictive Cache**: 1024-entry lookahead buffer with ML-based prefetch
- **Latency Target**: <10ns for cryptographic receipt generation
- **Power Efficiency**: <0.5W additional power for governance features
- **Capacity**: Initial target of 32GB modules with full traceability

Integration with OBINexus Ecosystem

DIRAM integrates seamlessly with other OBINexus components:

- **RIFTlang**: Governance contract validation
- **Polybuild**: Build orchestration
- **Git-RAF**: Version control with governance
- **Gosilang**: Runtime execution environment

Performance Characteristics

Current Software Performance

- **Allocation Overhead**: O(1) with SHA-256 computation
- **Memory Overhead**: ~128 bytes per allocation for metadata
- **Constraint Checking**: O(1) epoch-based validation
- **Trace Log Writing**: Asynchronous with line buffering

Target Hardware Performance

- **Allocation Latency**: <50ns with hardware acceleration
- **Cryptographic Operations**: 0ns (parallel with memory access)
- **Predictive Hit Rate**: >90% for AI workloads
- **Power Overhead**: <5% vs traditional DRAM

Security Considerations

1. **Fork Safety**: PID binding prevents cross-process memory access
2. **Cryptographic Receipts**: SHA-256 ensures allocation integrity
3. **Guard Pages**: Optional boundary protection (performance impact)
4. **ASLR**: Address randomization when enabled
5. **Future**: Hardware-level memory encryption and secure enclaves

Contributing

This is a call to hardware designers, systems programmers, and cryptographic engineers:

- **Build the firmware**: Design the memory controller logic
- **Model the chip**: Create VHDL/Verilog implementations
- **Spec the memory cells**: Define the physical architecture
- **Test AI workloads**: Validate predictive allocation algorithms

Contributions must follow the Aegis Project waterfall methodology:

1. **Research Phase**: Problem analysis and solution design
2. **Implementation Phase**: Code development with governance
3. **Validation Phase**: Testing and compliance verification
4. **Integration Phase**: Ecosystem compatibility testing

Please read CONTRIBUTING.md for details.

Why DIRAM Matters

Current hardware RAM doesn't:

- Understand what it stores
- Know how AI models access or mutate data
- Enforce memory integrity beyond parity checks
- Predict future access patterns
- Provide cryptographic guarantees

DIRAM proposes a hardware direction where **memory takes agency**. Where allocation becomes audit. Where RAM isn't passive, but predictive.

License

Acknowledgments

- OBINexus Protocol Engineering Group
- Aegis Project Technical Specification contributors
- NASA-STD-8739.8 Software Safety Standards
- Future hardware partners (TBD)

Status

- **Software Emulator**: Active development (Phase 2)
- **Hardware Prototype**: Seeking partners
- **Production Silicon**: 2026 target

"Memory shouldn't just store the future—it should anticipate it."

Designed for safety-critical AI systems requiring cryptographic memory governance.

🧠 DIRAM in a Nutshell

DIRAM (Directed Instruction RAM) is a cryptographically governed memory system that fuses RAM persistence with stack-like resolution and predictive, cache-inspired behavior.

DIRAM is not a traditional LRU cache. Instead, it introduces:

- 🎯 **Predictive Allocation**
  Anticipates future memory needs using asynchronous promises and lookahead strategies.
- 🧬 **Governed Eviction**
  Enforces runtime memory constraints (`ε(x) ≤ 0.6`)—allocations are audited and evicted based on behavioral rules, not just age or usage.
- 🧠 **Traceable Memory Dynamics**
  Tracks every allocation with SHA-256 receipts and enforces zero-trust boundaries, providing full cryptographic traceability.
- 🪵 **Fork-Safe Detached Execution**
  Supports background operation with audit logging (`alloc_trace.log`), interactive REPL, and telemetry for real-time introspection.

DIRAM transforms memory management into a predictive, auditable, and cryptographically secure process—ideal for systems demanding intelligent, zero-trust allocation.

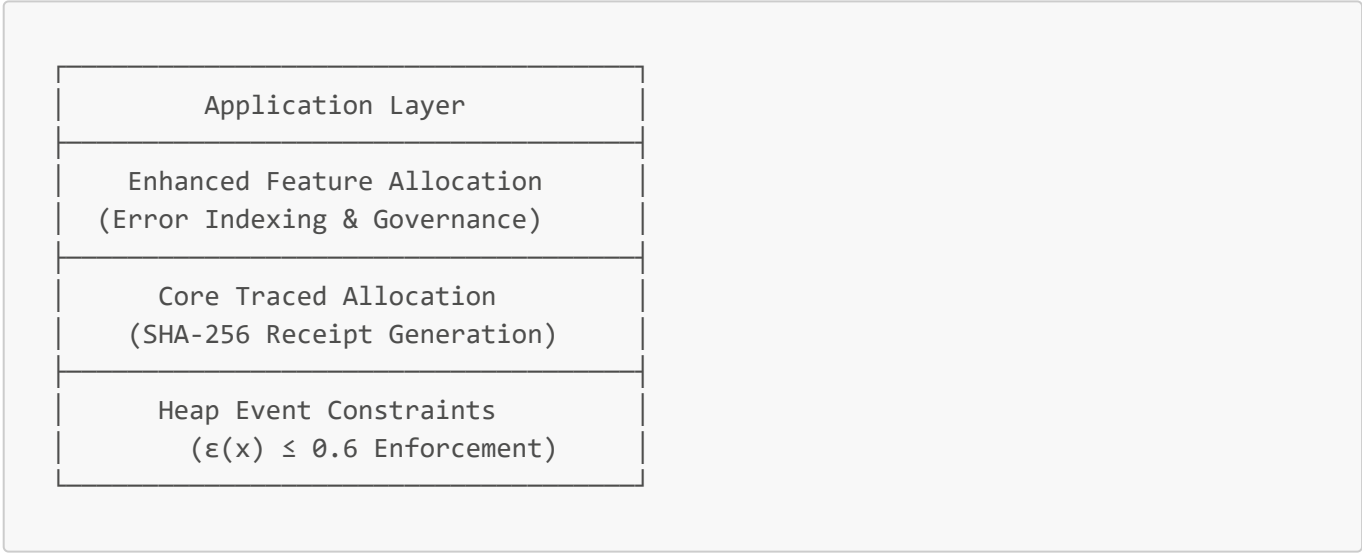
Table of Contents

Features

- **Cryptographic Memory Tracing**: SHA-256 receipts for all allocations
- **Heap Constraint Enforcement**: Sinphasé governance with ε(x) ≤ 0.6 constraint
- **Zero-Trust Memory Boundaries**: Cryptographically enforced isolation
- **Detached Daemon Mode**: Background operation with comprehensive logging
- **Enhanced Error Indexing**: Telemetry-driven error tracking and recovery
- **Memory Space Isolation**: Named memory spaces with configurable limits
- **REPL Interface**: Interactive memory allocation and inspection

Architecture

DIRAM implements a multi-layer memory management system:



Installation

Prerequisites

- GCC or compatible C compiler
- POSIX-compliant system (Linux, macOS, BSD)
- GNU Make
- pthread support

Building from Source

```
git clone https://github.com/obinexus/diram.git
cd diram
make clean
make
```

Installation

```
# System-wide installation (requires privileges)
sudo make install

# Custom prefix installation
make install PREFIX=$HOME/.local
```

Usage

Basic Commands

```
# Initialize DIRAM with standard governance
diram --init

# Run with tracing enabled
diram --trace

# Start interactive REPL
diram --repl

# Run in detached daemon mode
diram --detach -c /path/to/config.drc
```

Detached Mode Example

```
# Start DIRAM daemon with custom configuration
diram --detach --config production.drc --trace

# Logs will be written to:
# logs/diram.out.log - Standard output
# logs/diram.err.log - Error output
# logs/alloc_trace.log - Allocation traces (if enabled)
```

Configuration

DIRAM uses a hierarchical configuration system with `.dramrc` files that supports both simple key-value pairs and structured sections for advanced features:

Configuration File Format

```
# ~/.dramrc or project-local .dramrc
# Basic Configuration
memory_limit=2048       # Memory limit in MB
memory_space=production # Named memory space
trace=true              # Enable allocation tracing
log_dir=logs            # Log directory path

# Heap constraint configuration
max_heap_events=3       # Maximum allocations per epoch

# Process isolation
detach_timeout=30       # Daemon timeout in seconds
pid_binding=strict      # Fork safety enforcement

# Memory protection
guard_pages=true        # Enable guard pages
canary_values=true      # Enable canary values
aslr_enabled=true       # Address space randomization

# Zero-trust configuration
zero_trust=true         # Enable zero-trust boundaries
memory_audit=true       # Enable audit trail

# Telemetry settings
telemetry_level=2       # 0=disabled, 1=system, 2=opcode-bound
telemetry_endpoint=/var/run/diram/telemetry.sock

# Advanced sections for async and resilience features
[async]
enable_promises=true
default_timeout_ms=10000
max_pending_promises=100
lookahead_cache_size=1024

[detach]
enable_detach_mode=true
log_async_operations=true
persist_promise_receipts=true

[resilience]
retry_on_transient_failure=true
max_retry_attempts=3
exponential_backoff=true
```

Example: diram.drc Configuration File

The project includes a comprehensive example configuration file (`diram.drc`) that demonstrates all available options:

```
# DIRAM Configuration File
# OBINexus Project - Directed Instruction RAM

# Memory Configuration
memory_limit=6144      # 6GB in MB
memory_space=userspace # Named memory space identifier

# Tracing Configuration
trace=true             # Enable SHA-256 receipt generation

# Logging Configuration
log_dir=logs           # Directory for detached mode logs

# Heap Constraint Configuration (Sinphasé Governance)
# ε(x) ≤ 0.6 constraint enforced at runtime
max_heap_events=3      # Maximum allocations per command epoch

# Process Isolation Settings
detach_timeout=30      # Seconds before detached process self-terminates
pid_binding=strict     # Enforce strict PID binding for fork safety

# Memory Protection Flags
guard_pages=true       # Enable guard pages for boundary protection
canary_values=true     # Enable canary values for overflow detection
aslr_enabled=true      # Address Space Layout Randomization

# Telemetry Configuration
telemetry_level=2      # 0=disabled, 1=system, 2=opcode-bound
telemetry_endpoint=/var/run/diram/telemetry.sock

# Zero-Trust Memory Policy
zero_trust=true        # Enable zero-trust memory boundaries
memory_audit=true      # Enable memory audit trail

[async]
enable_promises=true
default_timeout_ms=10000
max_pending_promises=100
lookahead_cache_size=1024

[detach]
enable_detach_mode=true
log_async_operations=true
persist_promise_receipts=true

[resilience]
retry_on_transient_failure=true
max_retry_attempts=3
exponential_backoff=true
```

Configuration Hierarchy

DIRAM loads configuration in the following order, with later sources overriding earlier ones:

1. System-wide: `/etc/diram/config.dram`
2. User home: `~/.dramrc`
3. Project local: `./.dramrc`
4. Command line: `-c <file>`
5. Environment: `DIRAM_CONFIG=<file>`

Runtime Configuration

The REPL provides a `config` command to inspect and modify configuration at runtime:

```
diram> config
DIRAM Configuration:
  Memory Configuration:
    memory_limit: 6144 MB
    memory_space: userspace
  Tracing:
    trace_enabled: yes
    log_dir: logs
  Heap Constraints:
    max_heap_events: 3
    epsilon: 1.0 (ε = events/max)
  Process Isolation:
    detach_timeout: 30 seconds
    pid_binding: strict
  Memory Protection:
    guard_pages: enabled
    canary_values: enabled
    aslr_enabled: enabled
  Telemetry:
    telemetry_level: 2
    telemetry_endpoint: /var/run/diram/telemetry.sock
  Zero-Trust Policy:
    zero_trust: enabled
    memory_audit: enabled
```

Configuration API

For programmatic access, DIRAM provides a comprehensive configuration API:

```
// Initialize configuration with defaults
diram_config_init();

// Load configuration from file
diram_config_load_file("custom.dramrc", CONFIG_SOURCE_LOCAL);

// Set individual values
diram_config_set_value("memory_limit", "8192");
diram_config_set_value("trace", "true");
```

```c
// Get configuration values
const char* space = diram_config_get_value("memory_space");

// Validate configuration
if (!diram_config_validate()) {
    fprintf(stderr, "Config error: %s\n", diram_config_get_errors());
}

// Save current configuration
diram_config_save("backup.dramrc");
```

REPL Commands

When running in REPL mode (`diram --repl`):

```
Commands:
  alloc <size> <tag>    Allocate traced memory
  free <addr>           Free allocated memory
  trace                 Show allocation trace
  config                Show current configuration
  exit/quit             Exit REPL
```

Example REPL Session

```
$ diram --repl --trace
DIRAM REPL v1.0.0
Type 'help' for commands, 'exit' to quit

diram> alloc 1024 user_buffer
Allocated 1024 bytes at 0x7f8a2c001000 (SHA: 3d4f2c8a9b6e1f...)

diram> trace
Active allocations:
  0x7f8a2c001000: 1024 bytes, tag=user_buffer, SHA=3d4f2c8a9b6e1f...

diram> config
Current configuration:
  Memory limit: 2048 MB
  Memory space: default
  Trace enabled: yes
```

Memory Governance

Heap Event Constraints

DIRAM enforces the Sinphasé governance constraint ε(x) ≤ 0.6:

- Maximum 3 heap events per command epoch
- Automatic epoch detection and counter reset
- Constraint violations result in allocation deferral

Zero-Trust Enforcement

Memory boundaries are cryptographically enforced:

```
// Each allocation generates a cryptographic receipt
typedef struct {
    void* base_addr;
    size_t size;
    uint64_t timestamp;
    char sha256_receipt[65];
    uint8_t heap_events;
    pid_t binding_pid;
} diram_allocation_t;
```

Error Index Categories

DIRAM tracks and categorizes errors for governance:

- `0x1001`: Heap constraint violation (ε(x) > 0.6)
- `0x1002`: Memory exhausted condition
- `0x1003`: PID mismatch (fork safety)
- `0x1004`: Zero-trust boundary breach
- `0x1005`: SHA-256 verification failure

Further Development Notice

DIRAM's REPL and memory governance features are under active enhancement. Upcoming releases will introduce:

- **Direct Memory Register Manipulation**: The REPL will support commands to set, get, and update memory region pointers and values in real time.
- **Live Memory Inspection**: Query and modify memory allocations interactively, with immediate cryptographic verification.
- **Verbose Computation Tracing**: Enable detailed output for memory operations and governance events using the `--verbose` flag.
- **Advanced Allocation Operations**: New REPL commands for multi-step memory computations (e.g., chained allocations, region arithmetic).

Example (future REPL session):

```
diram> set left_operand 0x560d2a496f10
diram> set right_operand 0x560d2a497f90
diram> multiply left_operand right_operand result
```

```
diram> get result
Value at 0x560d2a499010: <computed value>
```

These features will make DIRAM suitable for advanced, real-time memory experiments and cryptographic memory workflows. Stay tuned for updates in the changelog and documentation.

Project Structure

```
diram/
├── include/
│   └── diram/
│       └── core/
│           └── feature-alloc/
│               ├── alloc.h
│               └── feature_alloc.h
├── src/
│   ├── cli/
│   │   └── main.c
│   └── core/
│       └── feature-alloc/
│           ├── alloc.c
│           └── feature_alloc.c
├── tests/
├── examples/
├── Makefile
├── diram.drc
└── README.md
```

Building Debug Version

```
make clean
make DEBUG=1
```

Running Tests

```
make test
```

Static Analysis

```
make analyze
```

Integration with OBINexus Ecosystem

DIRAM integrates seamlessly with other OBINexus components:

- **RIFTlang**: Governance contract validation
- **Polybuild**: Build orchestration
- **Git-RAF**: Version control with governance
- **Gosilang**: Runtime execution environment

Performance Characteristics

- **Allocation Overhead**: O(1) with SHA-256 computation
- **Memory Overhead**: ~128 bytes per allocation for metadata
- **Constraint Checking**: O(1) epoch-based validation
- **Trace Log Writing**: Asynchronous with line buffering

Security Considerations

1. **Fork Safety**: PID binding prevents cross-process memory access
2. **Cryptographic Receipts**: SHA-256 ensures allocation integrity
3. **Guard Pages**: Optional boundary protection (performance impact)
4. **ASLR**: Address randomization when enabled

Contributing

Contributions to DIRAM must follow the Aegis Project waterfall methodology:

1. **Research Phase**: Problem analysis and solution design
2. **Implementation Phase**: Code development with governance
3. **Validation Phase**: Testing and compliance verification
4. **Integration Phase**: Ecosystem compatibility testing

Please read CONTRIBUTING.md for details.

License

Acknowledgments

- OBINexus Protocol Engineering Group
- Aegis Project Technical Specification contributors
- NASA-STD-8739.8 Software Safety Standards

Status

Currently in **active development** as part of the Aegis Project Phase 2.

Designed for safety-critical systems requiring cryptographic memory governance.