REDIS SECURITY

CODE EXAMPLES INCLUDED

OKAN YILDIZ

JUNE 2025

Introduction to Redis Security

Redis has evolved from a simple caching solution to a critical component powering modern enterprise applications. As an in-memory data store, Redis handles sensitive user sessions, application data, and real-time analytics. This central role makes Redis security absolutely critical for protecting business data and maintaining customer trust.

The challenge with Redis security lies in its performance-first design philosophy. Historically, Redis prioritized speed over security, shipping with minimal protection enabled by default. This approach worked well for isolated development environments but creates significant risks in production deployments where Redis often handles sensitive data like user sessions, payment information, and business intelligence.

Modern enterprise Redis deployments require comprehensive security measures that protect data without compromising the performance benefits that make Redis valuable. This guide provides practical security strategies that maintain Redis's high-performance characteristics while implementing enterprise-grade protection.

Why Redis Security Matters:

Redis typically stores highly sensitive data including user authentication tokens, session data, cached database queries, and real-time application state. A security breach can expose this information directly from memory, making the impact immediate and severe. Unlike traditional databases that primarily store data on encrypted disks, Redis keeps active datasets in memory where they're more vulnerable to various attack vectors.

Additionally, Redis's network accessibility means security failures can be exploited remotely. Default configurations often bind to all network interfaces without authentication, creating obvious attack vectors that malicious actors can easily discover and exploit.

Redis Security Fundamentals

Understanding Redis Attack Vectors

Before implementing security measures, it's important to understand how Redis can be compromised. The most common attack vectors include:

Unauthenticated Access represents the most basic but devastating attack vector. Default Redis installations accept connections without any authentication, allowing anyone with network access to read, modify, or delete all data. This vulnerability has been responsible for numerous high-profile data breaches.

Weak Authentication occurs when Redis uses simple or default passwords that can be easily guessed or brute-forced. Many organizations set weak passwords like "redis" or "password" that provide minimal actual protection.

Network Exposure happens when Redis instances are accessible from the internet or untrusted networks. Attackers regularly scan for exposed Redis instances and can compromise them within minutes of discovery.

Command Injection exploits Redis's powerful command set to execute dangerous operations. Commands like FLUSHALL can delete all data, while CONFIG commands can modify server settings and potentially execute arbitrary code.

Data Interception occurs when Redis communications travel over unencrypted network connections, allowing attackers to intercept sensitive data including authentication credentials and application data.

Core Security Principles

Effective Redis security follows several fundamental principles that work together to create comprehensive protection:

Defense in Depth implements multiple security layers so that if one fails, others continue providing protection. This includes network security, authentication, authorization, encryption, and monitoring working together.

Principle of Least Privilege ensures users and applications have only the minimum access necessary to perform their functions. This limits the potential damage from compromised credentials or applications.

Secure by Default means configuring Redis with security-first settings rather than convenience-first defaults. This involves enabling authentication, restricting network access, and disabling dangerous commands.

Continuous Monitoring provides real-time visibility into Redis access patterns and security events, enabling rapid detection and response to potential threats.

Authentication and Access Control

Modern Redis Authentication

Redis 6.0 introduced Access Control Lists (ACLs) that provide granular user management far superior to the traditional single-password authentication. ACLs allow creating users with specific command permissions and key access patterns.

Setting Up Basic Authentication:

```
# Traditional password authentication
requirepass "your-secure-password-here"

# Client connection with password
```

```
redis-cli -a "your-secure-password-here"
```

Implementing ACL-Based Authentication:

```
# Create application-specific users
ACL SETUSER app_cache on >cache_password +@read +@write ~cache:* ~temp:*
ACL SETUSER session_store on >session_password +@read +@write +expire
~session:*
ACL SETUSER readonly_monitor on >monitor_password +@read ~*

# Administrative user with restricted dangerous commands
ACL SETUSER admin on >admin_password +@all ~* -flushall -flushdb

# List all users and their permissions
ACL LIST
```

This approach creates specialized users for different application functions. The `app_cache` user can only read and write to keys matching `cache:*` and `temp:*` patterns, while the `readonly_monitor` user can only execute read commands.

Role-Based Access Control Implementation

For larger organizations, implementing role-based access control simplifies user management while maintaining security:

```
class RedisACLManager:
    def __init__(self, redis_client):
        self.redis = redis_client
        self.predefined_roles = {
            'cache_user': {
                'commands': ['+@read', '+@write', '-@dangerous'],
                'keys': ['cache:*', 'temp:*']
            },
            'session_manager': {
                'commands': ['+@read', '+@write', '+expire', '+ttl'],
                'keys': ['session:*', 'user:*']
            },
            'readonly_analyst': {
                'commands': ['+@read', '+info', '+ping'],
                'keys': ['analytics:*', 'metrics:*']
            }
        }
```

```
    def create_user_with_role(self, username, password, role):
        if role not in self.predefined_roles:
            raise ValueError(f"Unknown role: {role}")

        role_config = self.predefined_roles[role]

        # Build ACL command
        acl_parts = [f"ACL SETUSER {username} on >{password}"]
        acl_parts.extend(role_config['commands'])

        for key_pattern in role_config['keys']:
            acl_parts.append(f"~{key_pattern}")

        acl_command = ' '.join(acl_parts)
        return self.redis.execute_command(acl_command)

# Usage
acl_manager = RedisACLManager(redis_client)
acl_manager.create_user_with_role('app1_cache', 'secure_password',
'cache_user')
```

Password Security and Management

Strong password policies are essential for Redis security:

Password Generation and Storage:

```
# Generate cryptographically secure passwords
openssl rand -base64 32

# Store passwords securely (example with environment variables)
export REDIS_APP_PASSWORD=$(openssl rand -base64 32)
export REDIS_ADMIN_PASSWORD=$(openssl rand -base64 32)

# Use password files for applications
echo "$(openssl rand -base64 32)" > /secure/redis-app-password.txt
chmod 600 /secure/redis-app-password.txt
chown app-user:app-group /secure/redis-app-password.txt
```

Password Rotation Procedures:

```
#!/bin/bash
# Redis password rotation script
OLD_PASSWORD="$1"
```

```
NEW_PASSWORD="$(openssl rand -base64 32)"

# Update password in Redis
redis-cli -a "$OLD_PASSWORD" CONFIG SET requirepass "$NEW_PASSWORD"

# Update application configurations
echo "New password: $NEW_PASSWORD"
echo "Update your applications with this new password"
```

Network Security Configuration

TLS Encryption Setup

TLS encryption protects Redis communications from eavesdropping and tampering. Modern Redis deployments should use TLS for all connections:

Basic TLS Configuration:

```
# Redis configuration for TLS
port 0                    # Disable plain TCP
tls-port 6380            # Enable TLS port
tls-cert-file /etc/redis/tls/server.crt
tls-key-file /etc/redis/tls/server.key
tls-ca-cert-file /etc/redis/tls/ca.crt
tls-protocols "TLSv1.2 TLSv1.3"
```

Generate TLS Certificates:

```
# Create Certificate Authority
openssl genrsa -out ca-key.pem 4096
openssl req -new -x509 -days 365 -key ca-key.pem -out ca-cert.pem

# Create server certificate
openssl genrsa -out server-key.pem 2048
openssl req -new -key server-key.pem -out server-req.pem
openssl x509 -req -in server-req.pem -CA ca-cert.pem -CAkey ca-key.pem -CAcreateserial
-out server-cert.pem -days 365
```

Client TLS Connection:

```
import redis
```

```
import ssl

# Create TLS context
context = ssl.create_default_context(ssl.Purpose.SERVER_AUTH)
context.load_verify_locations('/path/to/ca-cert.pem')

# Connect with TLS
redis_client = redis.Redis(
    host='redis.company.com',
    port=6380,
    password='your-password',
    ssl=True,
    ssl_context=context
)
```

Network Access Control

Proper network configuration limits Redis access to authorized systems only:

Binding Configuration:

```
# Bind to specific interfaces only (never use 0.0.0.0 in production)
bind 127.0.0.1 10.0.1.50

# Enable protected mode
protected-mode yes

# Set connection limits
maxclients 1000
```

Firewall Configuration:

```
# Allow Redis access only from application servers
iptables -A INPUT -p tcp --dport 6380 -s 10.0.1.0/24 -j ACCEPT
iptables -A INPUT -p tcp --dport 6380 -j DROP

# Rate limiting
iptables -A INPUT -p tcp --dport 6380 -m limit --limit 100/sec -j ACCEPT
```

Connection Security

Implementing connection security prevents various network-based attacks:

```
# Connection timeout settings
timeout 300              # Client idle timeout
tcp-keepalive 300       # TCP keepalive interval

# Rename dangerous commands
rename-command FLUSHDB ""
rename-command FLUSHALL ""
rename-command CONFIG "CONFIG_a1b2c3d4"
rename-command SHUTDOWN ""
```

Data Protection and Encryption

Application-Level Encryption

For sensitive data, implement encryption at the application level to ensure data remains protected even if Redis is compromised:

```
from cryptography.fernet import Fernet
import json

class EncryptedRedisClient:
    def __init__(self, redis_client, encryption_key):
        self.redis = redis_client
        self.fernet = Fernet(encryption_key)

    def set_encrypted(self, key, value, ex=None):
        """Set encrypted value in Redis"""
        # Serialize and encrypt
        serialized = json.dumps(value) if isinstance(value, (dict,
list)) else str(value)
        encrypted = self.fernet.encrypt(serialized.encode())

        # Store in Redis
        return self.redis.set(key, encrypted, ex=ex)

    def get_encrypted(self, key):
        """Get and decrypt value from Redis"""
        encrypted_data = self.redis.get(key)
        if not encrypted_data:
            return None

        # Decrypt and deserialize
        decrypted = self.fernet.decrypt(encrypted_data)
        try:
```

```
            return json.loads(decrypted.decode())
        except json.JSONDecodeError:
            return decrypted.decode()

# Usage
encryption_key = Fernet.generate_key()
encrypted_client = EncryptedRedisClient(redis_client, encryption_key)

# Store sensitive user data
user_data = {'email': 'user@example.com', 'phone': '+1234567890'}
encrypted_client.set_encrypted('user:123', user_data, ex=3600)

# Retrieve and decrypt
retrieved_data = encrypted_client.get_encrypted('user:123')
```

Data Classification and Protection

Implement data classification to apply appropriate protection levels:

```
import re

class DataClassifier:
    def __init__(self):
        self.sensitive_patterns = {
            'email':
r'\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,}\b',
            'ssn': r'\b\d{3}-\d{2}-\d{4}\b',
            'phone': r'\b\d{3}-\d{3}-\d{4}\b',
            'credit_card': r'\b\d{4}[-\s]?\d{4}[-\s]?\d{4}[-\s]?\d{4}\b'
        }

    def contains_sensitive_data(self, data):
        """Check if data contains sensitive information"""
        text = json.dumps(data) if isinstance(data, (dict, list)) else
str(data)

        for pattern_name, pattern in self.sensitive_patterns.items():
            if re.search(pattern, text, re.IGNORECASE):
                return True, pattern_name

        return False, None

    def should_encrypt(self, key, data):
        """Determine if data should be encrypted based on key and
content"""
```

```
        # Always encrypt based on key patterns
        sensitive_key_patterns = ['user:', 'session:', 'payment:',
'auth:']
        if any(pattern in key for pattern in sensitive_key_patterns):
            return True

        # Check data content
        is_sensitive, _ = self.contains_sensitive_data(data)
        return is_sensitive

# Integration with encrypted client
classifier = DataClassifier()

def smart_set(key, value, ex=None):
    """Automatically encrypt sensitive data"""
    if classifier.should_encrypt(key, value):
        return encrypted_client.set_encrypted(key, value, ex)
    else:
        return redis_client.set(key, value, ex)
```

Redis Cluster Security

Cluster Authentication

Redis Cluster requires special consideration for inter-node communication security:

```
# Cluster configuration with authentication
cluster-enabled yes
masterauth "cluster-node-password"
requirepass "cluster-node-password"

# Cluster node communication timeout
cluster-node-timeout 15000
```

Cluster TLS Configuration

Enable TLS for cluster communication:

```
# Enable TLS for cluster bus
tls-cluster yes
tls-cert-file /etc/redis/cluster-cert.pem
tls-key-file /etc/redis/cluster-key.pem
```

```
tls-ca-cert-file /etc/redis/cluster-ca.pem
```

Cluster Security Monitoring

Monitor cluster security across all nodes:

```
class ClusterSecurityMonitor:
    def __init__(self, cluster_nodes, password):
        self.nodes = cluster_nodes
        self.password = password

    def check_cluster_security(self):
        """Check security status across all cluster nodes"""
        results = {}

        for node in self.nodes:
            try:
                client = redis.Redis(
                    host=node['host'],
                    port=node['port'],
                    password=self.password
                )

                # Check authentication
                client.ping()

                # Check configuration
                config = client.config_get('*')

                results[f"{node['host']}:{node['port']}"] = {
                    'auth_enabled': 'requirepass' in config,
                    'cluster_enabled': config.get('cluster-enabled') ==
'yes',
                    'protected_mode': config.get('protected-mode') ==
'yes'
                }

            except Exception as e:
                results[f"{node['host']}:{node['port']}"] = {
                    'error': str(e)
                }

        return results
```

Monitoring and Auditing

Security Event Monitoring

Implement comprehensive monitoring to detect security incidents:

```
import time
import json
from collections import defaultdict

class RedisSecurityMonitor:
    def __init__(self, redis_client):
        self.redis = redis_client
        self.failed_attempts = defaultdict(list)
        self.alert_threshold = 5
        self.time_window = 300  # 5 minutes

    def log_auth_attempt(self, client_ip, username, success):
        """Log authentication attempts"""
        event = {
            'timestamp': time.time(),
            'client_ip': client_ip,
            'username': username,
            'success': success,
            'event_type': 'auth_attempt'
        }

        # Store in Redis for persistence
        self.redis.lpush('security_events', json.dumps(event))
        self.redis.ltrim('security_events', 0, 10000)  # Keep last 10k
events

        # Check for brute force
        if not success:
            self.check_brute_force(client_ip)

    def check_brute_force(self, client_ip):
        """Detect brute force attempts"""
        current_time = time.time()
        window_start = current_time - self.time_window

        # Add this failed attempt
        self.failed_attempts[client_ip].append(current_time)

        # Remove old attempts
        self.failed_attempts[client_ip] = [
```

```
            attempt for attempt in self.failed_attempts[client_ip]
            if attempt > window_start
        ]

        # Check threshold
        if len(self.failed_attempts[client_ip]) >= self.alert_threshold:
            self.trigger_security_alert('brute_force', {
                'client_ip': client_ip,
                'attempts': len(self.failed_attempts[client_ip])
            })

    def trigger_security_alert(self, alert_type, details):
        """Trigger security alert"""
        alert = {
            'alert_type': alert_type,
            'details': details,
            'timestamp': time.time()
        }

        print(f"SECURITY ALERT: {alert_type} - {details}")

        # Store alert
        self.redis.lpush('security_alerts', json.dumps(alert))

        # Implement your notification logic here
        # (email, Slack, webhook, etc.)
```

Audit Logging

Implement comprehensive audit logging for compliance:

```
import logging
import json
from datetime import datetime

class RedisAuditLogger:
    def __init__(self):
        # Configure logger
        self.logger = logging.getLogger('redis_audit')
        self.logger.setLevel(logging.INFO)

        # File handler
        handler = logging.FileHandler('/var/log/redis/audit.log')
        formatter = logging.Formatter(
            '%(asctime)s - %(levelname)s - %(message)s'
```

```
        )
        handler.setFormatter(formatter)
        self.logger.addHandler(handler)

    def log_access(self, user, action, resource, result):
        """Log access events"""
        audit_entry = {
            'timestamp': datetime.utcnow().isoformat(),
            'user': user,
            'action': action,
            'resource': resource,
            'result': result,
            'event_type': 'access'
        }

        self.logger.info(json.dumps(audit_entry))

    def log_admin_action(self, admin_user, action, details):
        """Log administrative actions"""
        audit_entry = {
            'timestamp': datetime.utcnow().isoformat(),
            'admin_user': admin_user,
            'action': action,
            'details': details,
            'event_type': 'admin'
        }

        self.logger.info(json.dumps(audit_entry))
```

Security Best Practices

Essential Security Configuration

Never Use Default Settings in Production:

```
# Secure Redis configuration template
# Authentication
requirepass "generated-secure-password"

# Network security
bind 127.0.0.1 10.0.1.50
protected-mode yes
port 0
tls-port 6380
```

```
# Connection limits
maxclients 1000
timeout 300

# Disable dangerous commands
rename-command FLUSHDB ""
rename-command FLUSHALL ""
rename-command CONFIG "CONFIG_hidden_name"
rename-command SHUTDOWN ""
rename-command DEBUG ""
rename-command EVAL ""

# Logging
logfile /var/log/redis/redis-server.log
loglevel notice
```

Security Hardening Checklist

Pre-Production Security Verification:

- Authentication enabled (ACL or password)
- TLS encryption configured
- Network access restricted (bind, firewall)
- Dangerous commands disabled/renamed
- Connection limits configured
- Logging enabled
- Monitoring systems operational
- Backup security verified
- Security tests passed

Environment-Specific Considerations

Development Environment:

- Use separate credentials from production
- Enable basic authentication
- Restrict network access to development team
- Regular security reviews

Staging Environment:

- Mirror production security settings
- Use production-like data protection
- Test security configurations
- Validate monitoring systems

Production Environment:

- Maximum security configuration
- Comprehensive monitoring
- Regular security audits
- Incident response procedures

Common Security Pitfalls

Dangerous Default Configurations

Mistake: Leaving Redis Open to the Internet

Many Redis instances are accidentally exposed to the internet with default configurations. Always check your network configuration:

```
# Check if Redis is exposed
nmap -p 6379 your-server-ip

# Proper binding (never use 0.0.0.0)
bind 127.0.0.1 10.0.1.50
```

Mistake: Using Weak or Default Passwords

Default or weak passwords are easily compromised:

```
# Bad examples
requirepass "password"
requirepass "redis"
requirepass "123456"

# Good example
requirepass "$(openssl rand -base64 32)"
```

Mistake: Not Disabling Dangerous Commands

Commands like FLUSHALL can destroy all data:

```
# Disable in production
rename-command FLUSHALL ""
rename-command FLUSHDB ""
rename-command CONFIG ""
```

Access Control Mistakes

Mistake: Overprivileged Applications

Applications should have minimal necessary permissions:

```
# Bad: Full access
ACL SETUSER app on >password +@all ~*

# Good: Limited access
ACL SETUSER app on >password +@read +@write ~app:* ~cache:*
```

Mistake: Shared Credentials

Each application should have unique credentials:

```
# Create separate users for each application
ACL SETUSER app1_cache on >unique_password1 +@read +@write ~app1:*
ACL SETUSER app2_session on >unique_password2 +@read +@write ~session:*
```

Monitoring Oversights

Mistake: No Security Monitoring

Implement basic security monitoring:

```
# Monitor authentication failures
def check_failed_auths():
    events = redis_client.lrange('auth_failures', 0, -1)
    recent_failures = [
        event for event in events
        if time.time() - float(event) < 3600  # Last hour
    ]

    if len(recent_failures) > 10:
        send_alert("Multiple authentication failures detected")
```

Frequently Asked Questions

What are the most critical Redis security measures?

The three most critical Redis security measures are **authentication**, **network security**, and **command restrictions**. Every production Redis instance must have authentication enabled - either through passwords or ACLs. Network security involves binding Redis to specific interfaces and using firewalls to restrict access to authorized systems only. Command restrictions disable or rename dangerous commands that could delete data or compromise the server.

These three measures address the most common attack vectors. Authentication prevents unauthorized access, network security limits who can attempt connections, and command restrictions prevent data destruction even if someone gains access. Without any one of these measures, Redis deployments remain highly vulnerable to basic attacks.

How do I secure Redis without impacting performance?

Redis security can be implemented with minimal performance impact through careful configuration choices. **TLS encryption** adds some overhead but is essential for protecting data in transit - use TLS 1.3 and modern cipher suites for best performance. **Authentication caching** reduces the overhead of ACL checks. **Connection pooling** minimizes the cost of TLS handshakes by reusing secure connections.

For data encryption, implement **selective encryption** - only encrypt truly sensitive data rather than all cached content. Use **efficient encryption algorithms** like AES-GCM that can leverage CPU acceleration. **Monitoring systems** should use sampling rather than logging every operation to reduce overhead.

The key is balancing security with performance based on your specific threat model and compliance requirements.

Should I encrypt all data stored in Redis?

Data encryption in Redis should be based on **data sensitivity** and **compliance requirements** rather than applied universally. Encrypt data containing PII, authentication tokens, payment information, or any data subject to regulatory requirements. Less sensitive cache data like computed results or temporary data may not require encryption.

Application-level encryption provides the strongest protection but adds complexity and performance overhead. **Transport encryption (TLS)** protects data in transit and should be considered mandatory. **Disk encryption** protects Redis persistence files but doesn't protect data in memory.

Consider your threat model - if Redis servers could be physically compromised or if you need to meet specific compliance requirements, more comprehensive encryption may be necessary.

How do I monitor Redis for security incidents?

Effective Redis security monitoring focuses on **authentication events**, **command patterns**, and **performance anomalies**. Monitor failed authentication attempts to detect brute force attacks. Track execution of administrative commands like CONFIG or FLUSHALL. Watch for unusual connection patterns, command rates, or memory usage that might indicate compromise.

Implement **real-time alerting** for critical events like multiple authentication failures from the same IP, execution of dangerous commands, or sudden spikes in activity. **Centralized logging** helps correlate Redis events with other security data. **Baseline establishment** helps identify deviations that might indicate security issues.

Key metrics include authentication success/failure rates, connection counts, command execution patterns, and memory usage trends. Integrate Redis monitoring with your broader security monitoring infrastructure.

What's the difference between Redis authentication and authorization?

Authentication verifies identity - proving that a user is who they claim to be. In Redis, this typically involves passwords or certificates. **Authorization** determines what authenticated users can do - which commands they can execute and which keys they can access.

Traditional Redis used simple password authentication with no authorization granularity. **Redis ACLs** provide both authentication (user creation with passwords) and authorization (command and key permissions). Modern deployments should use ACLs to implement **role-based access control** where different applications have different permission levels.

For example, a caching application might be authorized to read and write cache keys but not execute administrative commands, while a monitoring system might only be authorized to execute read-only commands.

How do I handle Redis security in containerized environments?

Container security for Redis requires additional considerations beyond traditional deployments. **Never expose Redis ports** outside the container network unless absolutely necessary. Use **container-specific networks** to isolate Redis containers from other services. **Secrets management** should use container orchestration features rather than environment variables.

Image security involves using official Redis images, regularly updating base images, and scanning for vulnerabilities. **Runtime security** includes read-only root filesystems where possible and running containers with non-root users. **Network policies** in Kubernetes can restrict which pods can connect to Redis.

Volume security protects persistent data through encrypted storage and proper access controls. Consider using **service mesh** technologies for automatic TLS and authentication between services.

What are the compliance considerations for Redis?

Compliance requirements vary by regulation but common considerations include **data encryption** (both in transit and at rest), **access controls** with audit trails, **data retention** policies, and **incident response** procedures. GDPR requires data protection by design and the ability to delete personal data. HIPAA demands encryption and access logging for protected health information.

Audit logging must capture sufficient detail for compliance reporting while protecting sensitive information from exposure in logs. **Data residency** requirements may affect where Redis instances can be deployed. **Access reviews** ensure permissions remain appropriate over time.

Documentation should cover security procedures, risk assessments, and compliance measures. Regular **security assessments** verify ongoing compliance and identify areas for improvement.

How do I secure Redis Cluster deployments?

Redis Cluster security requires protecting both client communications and inter-node communications. **Cluster authentication** uses shared passwords for node membership. **TLS encryption** should be enabled for both client connections and cluster bus communications. **Network segmentation** isolates cluster nodes in protected subnets.

Node security involves securing each cluster member individually with proper authentication, command restrictions, and monitoring. **Split-brain protection** prevents cluster partitions from compromising data consistency. **Backup security** protects cluster snapshots and ensures recovery procedures maintain security configurations.

Monitoring cluster topology helps detect unauthorized nodes attempting to join the cluster. **Firewall rules** should allow cluster communication on necessary ports while blocking unauthorized access.

Related Articles

- Redis Official Security Documentation
- Redis ACL Documentation
- Redis TLS Configuration Guide
- OWASP Database Security Guidelines
- NIST Cybersecurity Framework
- Redis Cluster Security Best Practices
- AWS ElastiCache Security Features
- Google Cloud Memorystore Security
- Azure Cache for Redis Security
- Container Security Best Practices
- Database Encryption Standards
- Redis Enterprise Security Features
- In-Memory Computing Security Research