LIBORA: Building an unbreakable chain of trust

In the world of software distribution, integrity is non-negotiable. A single modified byte can transform a trusted component into an attack vector. LIBORA's distributed integrity verification system offers a solution by creating a cryptographically-secured dependency chain that detects tampering at a granular level. Palo Alto Networks + 8 This research provides a comprehensive implementation guide for this robust security architecture.

How LIBORA creates tamper-resistant distributed systems

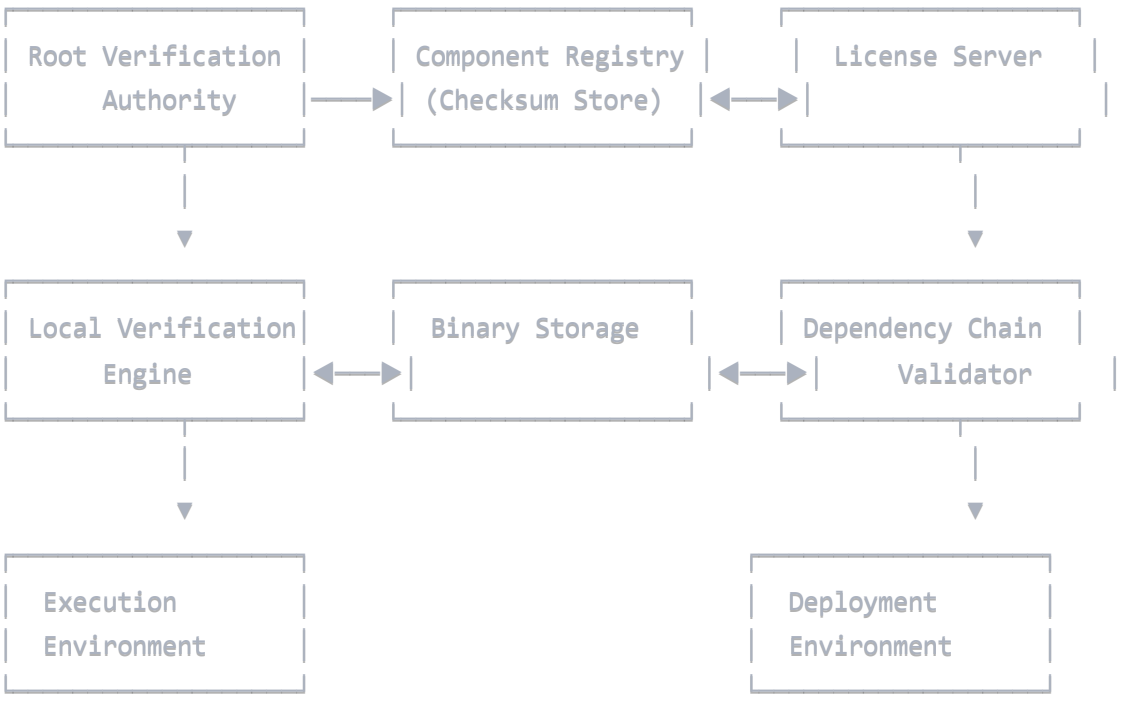
LIBORA (Library Oriented Robust Architecture) implements a multi-layered verification system that maintains integrity across distributed components through cryptographic dependency chains. Acm Whatls The system verifies not just individual components but also their relationships, detecting modification at both the component and system levels. Google + 3

Each binary component (like 01.bin, 02.bin) carries a unique cryptographic checksum (e.g., #1ad2, #23d3) that serves as its fingerprint. Wikipedia + 2 These checksums form dependency chains where each component's verification data incorporates references to its dependencies, creating an unbreakable chain of trust. Palo Alto Networks + 4

The architecture follows a defense-in-depth strategy, with multiple verification layers ensuring that compromising one component doesn't undermine the entire system. Verification can operate both online and offline, making it suitable for applications with intermittent connectivity. GeeksforGeeks + 4

System architecture: Dependency chains as cryptographic guarantees

The verification architecture consists of several key components working together:



Component structure

Components in LIBORA are organized as:

1. **Binary Components**: Modular software units (01.bin, 02.bin) that can be distributed independently but verified collectively
2. **Component Metadata**: Each component contains:
   - Unique identifier
   - Version information
   - Dependency specifications
   - Cryptographic signature (W3C)
   - Component checksum (#1ad2, #23d3) (TechTarget)
3. **Manifest Files**: Document expected checksums and dependency relationships

Dependency chain construction

Dependency chains are constructed as directed acyclic graphs (DAGs) where: (Wikipedia)

1. Each component maintains cryptographic links to its dependencies
2. Dependencies are expressed through manifests and embedded within components
3. Checksums of dependent components are incorporated into verification data (ScienceDirect + 3)

For example, a chain might look like:

```
Component A (#1ad2) → Component B (#23d3) → Component C (#45f6)
```

(ScienceDirect)

This creates a strong guarantee - modifying any component invalidates all components that depend on it, making the tamper immediately detectable. (Palo Alto Networks)

Byte-by-byte verification process for hash digests

The verification process examines each byte of a hash digest to ensure complete integrity of the component. (Microsoft + 2)

Hash digest calculation and verification

LIBORA uses 64-256 byte hash digests, typically truncated to hexadecimal representation for efficiency. (Cryptography + 4) The verification process includes:

1. **Constant-time comparison** to prevent timing attacks:

c

```c
int constant_time_compare(const unsigned char *digest1,
                          const unsigned char *digest2, size_t length) {
    unsigned char result = 0;
    for (size_t i = 0; i < length; i++) {
        result |= digest1[i] ^ digest2[i];  // Bitwise XOR and OR
    }
    return result == 0;
}
```

2. **Truncation to hexadecimal** representation:

c

```c
void binary_to_hex(const unsigned char *binary, size_t binary_len,
                   char *hex, size_t hex_len) {
    static const char hex_chars[] = "0123456789abcdef";

    for (size_t i = 0, j = 0; i < binary_len && j < hex_len - 2; i++) {
        hex[j++] = hex_chars[(binary[i] >> 4) & 0x0F];
        hex[j++] = hex_chars[binary[i] & 0x0F];
    }
    hex[hex_len - 1] = '\0';
}
```

Random distribution requirements

To ensure even random distribution in hash values, LIBORA implementations should:

1. Use hash functions with strong avalanche effect (SHA-256 or BLAKE2/3) Stack Exchange + 4
2. Add salt or personalization to prevent rainbow table attacks Stack Overflow UMA Technology
3. Implement domain separation for different component types
4. Verify distribution properties periodically through statistical tests Stack Exchange + 2

For modern implementations, **BLAKE3** provides the best combination of security and performance, processing data at rates up to 10 GB/s on multi-core systems while maintaining cryptographic strength. Cryptography + 4

Tamper detection and response mechanisms

LIBORA implements sophisticated methods to detect and respond to tampering attempts. ScienceDirect + 4

Detection algorithms

1. **Basic checksum verification**:

pseudocode

```
ALGORITHM: BasicChecksumVerification
INPUT: component, expected_checksum
OUTPUT: boolean indicating if component is tampered

function VerifyComponent(component, expected_checksum):
    actual_checksum = ComputeChecksum(component)
    return constant_time_compare(actual_checksum, expected_checksum)
```

2. **Merkle tree verification** (ResearchGate) for efficient verification of large sets: (Eclipse Forensics) (NCBI)

pseudocode

```
ALGORITHM: MerkleTreeVerification
INPUT: components[], root_hash
OUTPUT: boolean indicating if any component is tampered and set of tampered components

function VerifyWithMerkleTree(components[], root_hash):
    // Build Merkle tree from components
    leaf_nodes = []
    for each component in components:
        leaf_nodes.append(ComputeChecksum(component))

    // Calculate tree
    tree = BuildMerkleTree(leaf_nodes)

    // Compare calculated root with expected root
    if tree.root != root_hash:
        // Perform traversal to identify tampered components
        tampered_components = IdentifyTamperedComponents(tree, root_hash, components)
        return false, tampered_components

    return true, empty_set
```

3. **Runtime integrity verification** for ongoing protection:

c

```
int verify_runtime_integrity(void *code_region, size_t size,
                             const unsigned char *expected_hash) {
    unsigned char current_hash[32];
    SHA256(code_region, size, current_hash);

    return constant_time_compare(current_hash, expected_hash, 32);
}
```

Response mechanisms

When tampering is detected, LIBORA systems respond with:

1. **Isolation** of compromised components to prevent further damage Gradle
2. **Alerts** to administrators about the security breach Gradle
3. **Automated recovery** through component replacement or rollback Gradle
4. **Forensic logging** to record tampering details for investigation ScienceDirect + 4

For critical components, the system can enter a safe mode or shut down entirely:

c

```c
void handle_tamper_detection(const char *component_id,
                             TamperType type, int severity) {
    if (severity == CRITICAL) {
        issue_system_alert(component_id, type);
        isolate_component(component_id);

        if (is_core_component(component_id)) {
            initiate_system_safe_mode();
        }
    }

    log_tamper_event(component_id, type, get_forensic_data());
    attempt_component_recovery(component_id);
}
```

Updating components while maintaining integrity chains

LIBORA provides mechanisms to update components without breaking the verification chain. ScienceDirect

Atomic update protocol

Updates must be atomic to maintain system integrity: ResearchGate

c

```c
typedef struct {
    char component_id[64];
    unsigned char old_hash[32];
    unsigned char new_hash[32];
    time_t timestamp;
    char authorized_by[128];
    unsigned char signature[256];
} UpdateCertificate;

int update_component_atomically(const char *component_id,
                                const unsigned char *new_version,
                                size_t size,
                                UpdateCertificate *certificate) {
    // Verify update authorization
    if (!verify_signature(certificate)) {
        return UPDATE_UNAUTHORIZED;
    }

    // Prepare update
    unsigned char new_hash[32];
    SHA256(new_version, size, new_hash);

    // Verify hash matches certificate
    if (!constant_time_compare(new_hash, certificate->new_hash, 32)) {
        return UPDATE_HASH_MISMATCH;
    }

    // Store new component
    if (!store_component(component_id, new_version, size)) {
        return UPDATE_STORAGE_FAILED;
    }

    // Update integrity chain
    if (!update_integrity_chain(component_id, certificate)) {
        rollback_component_update(component_id);
        return UPDATE_CHAIN_FAILED;
    }

    return UPDATE_SUCCESS;
}
```

Rebalancing search space

When components are updated (e.g., updating #23d3), the verification system must rebalance its search space:

1. **Consistent hashing** ensures minimal disruption during updates Wikipedia Highscalability
2. **Virtual nodes** provide smoother distribution of verification responsibilities Ably Realtime Highscalability
3. **Jump hash algorithm** efficiently maps components to verification nodes Stack Overflow + 7

The Jump Hash algorithm is particularly efficient: Ably Realtime Toptal

c

```c
int jump_hash(uint64_t key, int num_buckets) {
    int64_t b = -1;
    int64_t j = 0;

    while (j < num_buckets) {
        b = j;
        key = key * 2862933555777941757ULL + 1;
        j = (b + 1) * ((double)(1LL << 31) /
                       ((double)((key >> 33) + 1)));
    }

    return (int)b;
}
```

Offline verification capabilities

LIBORA supports verification without internet connectivity, essential for applications that start without network access. Sunysb + 5

Hybrid verification strategy

The system implements a hybrid approach:

c

```c
typedef enum {
    VERIFICATION_ONLINE,
    VERIFICATION_OFFLINE,
    VERIFICATION_HYBRID
} VerificationMode;

int verify_with_fallback(VerificationState *state,
                         const unsigned char *component,
                         size_t size) {
    // Try online verification first if available
    if (state->mode != VERIFICATION_OFFLINE && is_online()) {
        if (verify_online(component, size)) {
            // Update cache for offline verification
            state->last_online_verification = time(NULL);
            generate_component_hash(component, size,
                                    state->cached_verification);
            state->verification_valid = 1;
            return 1;
        }
    }

    // Fall back to offline verification
    if (state->mode != VERIFICATION_ONLINE) {
        // Check if offline cache is still valid
        time_t current_time = time(NULL);
        if (state->verification_valid &&
            (current_time - state->last_online_verification <=
             state->offline_cache_expiry)) {

            // Verify against cached verification
            unsigned char current_hash[32];
            generate_component_hash(component, size, current_hash);

            return constant_time_compare(current_hash,
                                         state->cached_verification, 32);
        }
    }

    return 0; // Verification failed
}
```

Secure bootup verification

For offline verification during startup:

c

```c
typedef struct {
    const char *component_id;
    unsigned char expected_hash[32];
    int verification_status;
} BootVerificationEntry;

int verify_component_at_boot(BootVerificationTable *table,
                             const char *component_id,
                             const unsigned char *component,
                             size_t size) {
    // Find entry for this component
    BootVerificationEntry *entry = find_entry(table, component_id);

    if (!entry) {
        // Component not in verification table
        return 0;
    }

    // Calculate current hash
    unsigned char current_hash[32];
    SHA256(component, size, current_hash);

    // Verify against expected hash
    if (constant_time_compare(current_hash, entry->expected_hash, 32)) {
        entry->verification_status = VERIFIED;
        table->verified_count++;
        return 1;
    } else {
        entry->verification_status = FAILED;
        return 0;
    }
}
```

C implementation details

Component checksum generation

Efficient checksum generation in C using SHA-256: Wikipedia + 4

c

```c
#include <openssl/sha.h>

void generate_component_hash(const unsigned char *component, size_t size,
                             unsigned char output[SHA256_DIGEST_LENGTH]) {
    SHA256_CTX sha256;
    SHA256_Init(&sha256);

    // Process in chunks for large components
    const size_t chunk_size = 8192;
    for (size_t offset = 0; offset < size; offset += chunk_size) {
        size_t bytes = (offset + chunk_size > size) ?
                       (size - offset) : chunk_size;
        SHA256_Update(&sha256, component + offset, bytes);
    }

    SHA256_Final(output, &sha256);
}
```

Microsoft

Dependency tracking implementation

Efficient dependency tracking using a directed acyclic graph: DeepSource GeeksforGeeks

c

```c
typedef struct Node {
    char *component_id;
    struct Node **dependencies;
    int dependency_count;
    int max_dependencies;
    unsigned char checksum[32];
} Node;

typedef struct {
    Node **nodes;
    int node_count;
    int max_nodes;
} DependencyGraph;

Node* add_component(DependencyGraph *graph, const char *component_id) {
    if (graph->node_count >= graph->max_nodes) {
        // Resize graph if needed
        graph->max_nodes *= 2;
        graph->nodes = realloc(graph->nodes,
                               sizeof(Node*) * graph->max_nodes);
    }

    Node *node = malloc(sizeof(Node));
    node->component_id = strdup(component_id);
    node->dependencies = malloc(sizeof(Node*) * 5);
    node->dependency_count = 0;
    node->max_dependencies = 5;

    graph->nodes[graph->node_count++] = node;
    return node;
}

void add_dependency(Node *component, Node *dependency) {
    if (component->dependency_count >= component->max_dependencies) {
        component->max_dependencies *= 2;
        component->dependencies = realloc(component->dependencies,
                                          sizeof(Node*) * component->max_dependencies);
    }

    component->dependencies[component->dependency_count++] = dependency;
} :antCitation[]{citations="4eb6688e-5b47-47a8-b1c3-d1b1596f55a0"}
```

Tamper detection algorithms

Self-checksumming code implementation for runtime integrity: Guardsquare + 5

c

```c
#include <stdlib.h>
#include <stdint.h>
#include <openssl/sha.h>

// Reserve space for checksum in a specific section
#pragma section("integrity", read)
__declspec(allocate("integrity"))
static const unsigned char code_checksum[32] = {0};

int verify_code_integrity(void) {
    // Get function boundaries
    uintptr_t start_addr = (uintptr_t)&verify_code_integrity;
    uintptr_t end_addr = start_addr + 1024; // Approximate size

    // Calculate actual checksum
    unsigned char actual_checksum[32];
    SHA256_CTX sha256;
    SHA256_Init(&sha256);

    // Skip the checksum area itself during calculation
    uintptr_t checksum_start = (uintptr_t)&code_checksum;
    uintptr_t checksum_end = checksum_start + sizeof(code_checksum);

    // Hash code before checksum area
    if (start_addr < checksum_start) {
        SHA256_Update(&sha256, (void*)start_addr,
                      checksum_start - start_addr);
    }

    // Hash code after checksum area
    if (checksum_end < end_addr) {
        SHA256_Update(&sha256, (void*)checksum_end,
                      end_addr - checksum_end);
    }

    SHA256_Final(actual_checksum, &sha256);

    // Compare with embedded checksum
    return constant_time_compare(actual_checksum, code_checksum, 32);
} :antCitation[]{citations="a51e531f-2c62-4b68-b2c3-9036de3a6d16"}
```

Integrity chain updates

Transaction-based integrity updates ensure consistency: Wikipedia + 4

c

```c
typedef struct {
    unsigned char *old_data;
    unsigned char *new_data;
    size_t size;
    unsigned char old_checksum[32];
    unsigned char new_checksum[32];
    int is_committed;
} IntegrityTransaction;

IntegrityTransaction* begin_integrity_transaction(void *data, size_t size) {
    IntegrityTransaction *tx = malloc(sizeof(IntegrityTransaction));

    // Make a copy of the current data
    tx->old_data = malloc(size);
    memcpy(tx->old_data, data, size);

    // Allocate space for new data
    tx->new_data = malloc(size);
    memcpy(tx->new_data, data, size);

    tx->size = size;

    // Compute checksum of current data
    SHA256(tx->old_data, size, tx->old_checksum);

    tx->is_committed = 0;

    return tx;
}

int commit_integrity_transaction(IntegrityTransaction *tx, void *target_data) {
    // Calculate checksum of new data
    SHA256(tx->new_data, tx->size, tx->new_checksum);

    // Verify current data hasn't been modified
    unsigned char current_checksum[32];
    SHA256(target_data, tx->size, current_checksum);

    if (!constant_time_compare(current_checksum, tx->old_checksum, 32)) {
        return 0; // Current data changed, transaction can't be applied
    }

    // Apply changes
    memcpy(target_data, tx->new_data, tx->size);
    tx->is_committed = 1;
```

```
    return 1;
}
```

W3C

Network and offline verification modes

Implementation of both network and offline verification: Dpconline

c

```c
typedef enum {
    ONLINE,
    OFFLINE,
    HYBRID
} VerificationMode;

typedef struct {
    VerificationMode mode;
    time_t last_online;
    time_t cache_expiry;
    unsigned char cached_hashes[MAX_COMPONENTS][32];
    char component_ids[MAX_COMPONENTS][64];
    int component_count;
} VerificationCache;

int verify_integrity(VerificationCache *cache,
                     const char *component_id,
                     const unsigned char *data,
                     size_t size) {
    unsigned char hash[32];
    SHA256(data, size, hash);

    // Try online verification if available
    if ((cache->mode == ONLINE || cache->mode == HYBRID) &&
        is_network_available()) {

        if (verify_with_server(component_id, hash)) {
            // Update cache
            update_verification_cache(cache, component_id, hash);
            return 1;
        }
    }

    // Fallback to offline verification
    if (cache->mode == OFFLINE || cache->mode == HYBRID) {
        if (is_in_verification_cache(cache, component_id, hash)) {
            return 1;
        }
    }

    return 0; // Verification failed
}
```

Real-world applications of LIBORA principles

LIBORA's principles apply to several critical security domains:

1. **License verification systems** that prevent tampering with licensing code Keygen + 4
2. **Software supply chain security** to ensure all components are authentic Palo Alto Networks + 3
3. **Secure software updates** that maintain integrity during deployment Owasp + 5
4. **IoT device security** to verify firmware integrity on resource-constrained devices Stack Exchange + 3
5. **Financial transaction verification** where integrity is paramount Critical Chains + 5

Conclusion

The LIBORA cryptographic framework provides a robust foundation for distributed integrity verification. By implementing component-level verification with dependency chains, the system can detect tampering with high precision ScienceDirect while supporting offline verification scenarios. Palo Alto Networks + 6

This implementation approach creates a powerful security architecture for real-world applications requiring strong integrity guarantees. The combination of cryptographic verification, dependency tracking, and tamper response mechanisms forms a comprehensive defense against software tampering and supply chain attacks. Owasp + 9