Zenroom Technical Whitepaper

Architecture, Security, and Implementation

Denis Roio
Dyne.org

Puria Nafisi Azizi
Dyne.org

Andrea D'Intino
Forkbomb B.V.

November 15, 2025

Abstract

Zencode is a human-readable domain-specific language (DSL) for cryptographic operations, executed by the Zenroom Virtual Machine in deterministic, isolated environments. Designed for zero-trust architectures, Zencode enables CTOs and CISOs to implement privacy-by-design principles with verifiable, auditable smart contracts that non-technical stakeholders can comprehend. This whitepaper presents the evolution from the 2019 proof-of-concept to the current production-grade implementation, detailing architectural decisions, security guarantees, and real-world deployments in digital identity, verifiable credentials, and blockchain integration.

1 Executive Summary

Most organizations today face a critical contradiction in their security infrastructure. While threat models evolve toward zero-trust architectures and cryptographic agility becomes mandatory for compliance, the tools available to implement these principles remain locked in patterns inherited from product-centric development more than 20 years ago. Smart contracts in production systems are written in languages designed for compilers and virtual machines, not for human comprehension or audit. The gap between what security officers need to verify and what developers can demonstrate grows wider with each new regulation.

Zenroom addresses this contradiction through a different starting point. Rather than building yet another blockchain virtual machine or extending existing languages with cryptographic libraries, we designed a restricted execution environment where the contract itself can be read and understood by stakeholders who need to trust it. The Zencode language that runs inside Zenroom is not a simplified syntax layered on top of complexity, it is the actual executable specification.

This distinction matters in practice. When your organization needs to implement W3C Verifiable Credentials for digital identity, you can show the Zencode contract to legal counsel, privacy officers, and auditors. They read the same text that executes in production. When regulations require you to transition from ECDSA to post-quantum signatures, the migration path is visible in human-readable statements, not buried in dependency chains and compiler optimizations.

The technical implementation reflects eight years of production deployment, not laboratory research. Zenroom has zero external dependencies by design. The entire virtual machine, cryptographic primitives, and language parser compile to under 3MB across every major platform: iOS and Android mobile, WebAssembly browsers, server Linux and Windows, embedded ARM Cortex and ESP32 chips. This footprint is not an optimization target, it is a security requirement. Attack surface minimization starts with what you choose not to include.

Our cryptographic capabilities span the current transition period in the industry. Native support for NIST P-256, secp256k1, BLS12-381, Ed25519 covers existing infrastructure integration. Post-quantum implementations of ML-KEM (Kyber) and ML-DSA (Dilithium) prepare for the upcoming mandates. BBS+ signatures enable selective disclosure and zero-knowledge proofs for privacy-preserving credentials. This range is not feature creep, it reflects the reality that organizations must maintain multiple cryptographic generations simultaneously during transitions that span years, not quarters.

Current production deployments demonstrate scope beyond pilot projects. The CREDIMI implementation pro-

vides EUDI-ARF wallet certification infrastructure for the European Digital Identity framework. DIDROOM operates as production W3C DID and Verifiable Credential infrastructure. The Global Passport Project processes identity verification at scale. These are not use cases or prototypes, they are running systems that would cost significantly more to implement and audit using conventional approaches.

For technical decision-makers, three aspects distinguish Zenroom from alternatives:

① Deterministic execution is guaranteed by the virtual machine design, not promised through careful coding practices. The same Zencode contract produces identical cryptographic outputs across all platforms, every time. This is verifiable.

② Isolation is enforced at the VM level. Zenroom has no access to filesystem, network, or calling process memory. Malicious contracts cannot break sandbox boundaries because those boundaries are not implemented through runtime checks, they are structural.

③ Language-theoretic security principles prevent entire classes of parser differential attacks. The Zencode grammar is intentionally restricted to avoid Turing-complete ambiguity. Input validation happens before processing, not during, following formal separation that most languages ignore.

Organizations adopting Zenroom typically cite reduced audit costs and compressed integration timelines as immediate value. The longer-term value emerges in cryptographic agility. When algorithm transitions become mandatory, whether due to quantum computing threats or regulatory requirements, the migration is a contract update visible to all stakeholders, not a infrastructure replacement project requiring specialized consultants.

The decision to adopt Zenroom is ultimately a choice about where complexity lives in your security infrastructure. You can push it into increasingly sophisticated toolchains and frameworks that require specialists to audit, or you can contain it inside a small, auditable VM that executes contracts anyone can read. Both approaches have costs. The first compounds over time. The second does not.

2 Introduction: The Algorithmic Sovereignty Challenge

2.1 The Trust Problem in Modern Computing

We are operating systems that most participants cannot read. The code executing decisions about resource allocation, access control, and identity verification runs in environments designed to optimize machine efficiency, not human understanding. This creates an asymmetry that compounds with scale. As more infrastructure moves toward automated decision-making through smart contracts and algorithmic governance, the gap between those who can verify system behavior and those who must trust it widens.

This asymmetry is not primarily technical. Every organization employing developers has staff capable of reading code. The problem is structural: most production systems depend on toolchains, frameworks, and runtime environments complex enough that verification requires specialists. A security audit of a Solidity smart contract must account for the compiler version, the EVM implementation, gas optimization patterns, and the interpretation of ambiguous language specifications. The contract source code is only the starting point for understanding what actually executes.

The situation worsens under distributed architectures where multiple implementations must reach consensus on contract execution results. Parser differentials between implementations create exploitable inconsistencies. Different organizations run different versions of the same virtual machine, apply different optimizations, and interpret edge cases differently. The 2016 DAO exploit demonstrated this at considerable cost. Subsequent incidents continue to validate that complexity enables exploitation.

2.2 Why Human-Readable Smart Contracts Matter

The term smart contract implies two things that rarely coexist in practice: machine executability and human comprehensibility. Most blockchain languages optimize for the first and treat the second as a documentation problem. Developers write in high-level languages that compile to opcodes, then separately document what the contract purports to do. Auditors review both the source and the compiled output, but cannot guarantee equivalence without trusting the compiler.

This separation introduces risk that extends beyond technical audit. Legal frameworks for contract enforcement assume all parties can examine contract terms. Regulatory compliance requires demonstrating to non-technical authorities what data processing occurs and under what conditions. Privacy by design principles mandate that individuals understand how systems handle their information. None of these requirements can be satisfied by showing stakeholders assembly code or pointing to documentation that might not match implementation.

The response in much of the industry has been to build increasingly sophisticated tooling: formal verification frameworks, automated testing suites, static analysis engines. These tools are valuable but they address symptoms rather than causes. They attempt to verify complex systems rather than questioning whether the complexity is necessary. Each new tool adds dependencies, increases the specialist knowledge required, and expands the trusted computing base.

A different approach starts by constraining what smart contracts can express. If the language is restricted enough,

human readability and machine executability can converge. The contract text itself becomes the executable specification. Verification means reading and understanding the contract, not analyzing compilation artifacts. This requires accepting limitations on what the language can do, but those limitations are precisely what make verification tractable.

2.3 The Cost of Cryptographic Complexity

Organizations implementing cryptographic systems face a timeline problem. Current infrastructure relies on algorithms that function adequately today but carry known future risks. NIST has published timelines for transitioning to post-quantum cryptography. The European Union mandates cryptographic agility in its cybersecurity frameworks. Industry standards bodies recommend maintaining multiple algorithm generations simultaneously.

These mandates create concrete costs. Supporting multiple cryptographic implementations means multiple libraries, multiple sets of dependencies, multiple testing and audit cycles. Each algorithm requires specialists familiar with its particular security assumptions and implementation pitfalls. When vulnerabilities emerge in widely-used libraries, the response involves tracking dependency chains through multiple layers of abstraction.

The conventional approach treats each cryptographic capability as a separate component integrated through APIs. You have an ECDSA library for signatures, an AES library for encryption, a different library for key derivation, another for random number generation. Each comes from different maintainers, follows different conventions, gets updated on different schedules. The integration code that binds these components becomes its own maintenance burden and potential vulnerability surface.

Production systems must also handle transitions between algorithms. When SHA-1 became insecure for certificates, organizations could not simply flip a switch. They had to maintain dual infrastructure, coordinate with partners on transition schedules, verify that all components properly handled both algorithms during the migration period. Similar transitions are coming for asymmetric cryptography as quantum computing advances. Organizations that handled one algorithm migration know the resource requirements. Having to handle multiple such migrations simultaneously, which is the current trajectory, scales costs considerably.

2.4 Zenroom's Mission and Design Philosophy

Zenroom emerged from research into how distributed systems can make decisions that participants can verify without requiring specialized expertise. The DECODE project for the European Union investigated whether communities could maintain technological sovereignty while using sophisticated cryptographic tools. The question was not whether experts could implement secure systems, but whether non-experts could audit and trust what experts built.

This question led to three design principles that distinguish Zenroom from other approaches:

① The contract language must be readable by stakeholders with no programming experience. This is not about making syntax slightly more friendly or adding better comments. The contract text itself, as written, must communicate its purpose and operations to someone familiar with the domain but not with coding.

② The execution environment must contain no hidden complexity. What you see in the contract is what executes. No compilation steps that could introduce behavior not visible in source. No runtime optimizations that could change outcomes. No external dependencies that could behave differently across installations.

③ Security must be structural, not procedural. The environment should make entire classes of vulnerabilities impossible by design rather than depending on developers following best practices. If the VM has no file system access, contracts cannot have file system vulnerabilities. If the parser rejects ambiguous statements, contracts cannot contain parser differential exploits.

These principles impose constraints that some will find limiting. Zencode is deliberately not Turing complete. It cannot express arbitrary computations. The cryptographic primitives are fixed at compile time, not loadable at runtime. The execution model is strictly sequential with no concurrent operations. Each of these constraints closes attack surfaces and simplifies verification.

The eight years since the initial DECODE implementation have validated this approach in production environments that traditional smart contract platforms struggle to address. Digital identity systems require privacy-preserving credential verification. Supply chain tracking needs cryptographic guarantees without exposing business logic. Municipal governance demands transparency without requiring citizens to understand assembly code. These use cases share a requirement: the stakeholders who must trust the system cannot be expected to trust developers by reputation. They need to verify by inspection.

Zenroom's current deployment across European digital identity infrastructure, global passport verification, and blockchain integration demonstrates that constrained languages can address real-world requirements. The question for organizations evaluating Zenroom is not whether it can express every possible computation, but whether its constraints align with actual security requirements better than unconstrained alternatives.

3 Threat Model and Security Requirements

3.1 Attack Surface Analysis

The attack surface of a cryptographic system extends beyond the primitives themselves. Most exploits in production occur not in the core algorithms but in the layers surrounding them: parsers, memory management, state machines, input validation, compilation pipelines, and the integration between components. Each of these layers introduces opportunities for adversaries to inject unexpected state or trigger undefined behavior.

Zenroom's threat model starts from the assumption that contracts may be malicious. This is not paranoia but a requirement for any system executing untrusted code. In distributed architectures, you cannot vet who writes contracts or predict their intentions. The execution environment must therefore be hostile to exploitation by design. Traditional approaches attempt to sandbox malicious behavior through runtime checks and permission systems. Zenroom eliminates attack vectors structurally before execution begins.

Consider the attack surface dimensions:

Input processing: Every data structure entering Zenroom passes through schema validation before any contract logic touches it. The schemas are declarative specifications written in the same readable format as contracts. A malformed input that violates its schema never reaches the contract execution phase. This is recognition before processing, a core tenet of language-theoretic security that most systems violate.

Parser implementation: The Zencode parser is a finite state machine with explicit transitions between Given, When, and Then phases. Each phase has defined read and write permissions on memory regions. Invalid state transitions are structurally impossible because the state machine definition prevents them. This differs from runtime permission checks which depend on correct implementation of every check in every code path.

Memory access: Zenroom implements three isolated memory compartments: IN for input data, ACK for computation workspace, and OUT for results. Contracts in the Given phase can read IN but cannot write to it. The When phase operates on ACK with no direct access to raw input. The Then phase writes to OUT but cannot modify ACK retroactively. These boundaries are enforced by the VM memory model, not by application logic that could contain mistakes.

External dependencies: Zero external dependencies means the trusted computing base is exactly the Zenroom binary. No dynamic library loading that could introduce compromised code. No network access that could leak data or receive commands. No filesystem operations that could persist state or read configuration files. The absence of these capabilities is architectural, verified at compile time, not disabled at runtime.

Compilation and distribution: Zencode contracts are not compiled. What executes is what you read. There is no intermediate representation that could differ from source. No optimization phase that could introduce timing variations. No linking step that could pull in different versions of shared libraries. The contract text maps directly to function calls in the VM, eliminating an entire class of supply chain attacks.

This attack surface analysis reveals where Zenroom differs from conventional approaches. Rather than protecting against attacks through layered defenses that each could fail, Zenroom removes the surfaces that would enable attacks. The principle is simple: if a component cannot express a vulnerability, you do not need to defend against that vulnerability.

3.2 Zero-Trust Architecture Principles

Zero-trust architecture assumes that no component is inherently trustworthy. Network position, authentication credentials, and prior good behavior do not grant trust. Every transaction must be verified, every input validated, every execution isolated. These principles map directly to Zenroom's design.

Never trust input: Zenroom treats all input as hostile until proven otherwise. The schema validation phase forces explicit specification of what constitutes valid input. If your contract expects a public key, the schema defines the exact encoding, length constraints, and curve membership requirements. An input that merely looks like a public key but fails point-on-curve validation never enters the computation phase. This approach follows the zero-trust principle that nothing is assumed safe merely because it has the right shape.

Isolation between components: In distributed systems, components from different trust domains must interact. Zero-trust requires that no component can affect another's internal state except through defined interfaces. Zenroom implements this through process isolation. Each contract execution runs in a fresh VM instance with no shared memory, no inter-process communication, and no persistent state between executions. One malicious contract cannot compromise subsequent executions.

Least privilege: Zero-trust mandates that components have only the minimum capabilities required for their function. Zenroom's VM has no privileges beyond memory allocation and computation. It cannot open files, create network connections, spawn processes, or access system resources. The host application provides data through defined input channels and receives results through output channels. The VM cannot request additional privileges or escalate its capabilities.

Verification at every step: Zero-trust architecture requires continuous verification, not authentication at entry

followed by implicit trust. Zenroom enforces this through its phase-based execution model. Data validated in the Given phase is tagged with metadata about its validation status. The When phase can only operate on validated data. The Then phase outputs results with cryptographic proof of their provenance. Each phase verifies preconditions before proceeding.

Determinism as security property: Zero-trust in distributed systems requires that all participants can verify results independently. Zenroom's deterministic execution serves this requirement. The same contract with the same inputs produces identical outputs on every platform, every time. This is not a performance optimization but a security property. If execution were non-deterministic, participants could not verify each other's results without trusting their honesty.

These principles are not implemented through configuration or policy. They are structural properties of the VM design. An organization deploying Zenroom does not need to configure zero-trust behavior or train developers to follow zero-trust practices. The architecture enforces it.

3.3 Isolation and Sandboxing Requirements

Effective sandboxing requires more than preventing file system and network access. A truly isolated execution environment must protect the host from memory corruption, infinite loops, resource exhaustion, side-channel leaks, and covert channels. Conventional sandboxing approaches use operating system mechanisms like seccomp, SELinux, or containerization. These are valuable layers but they depend on correct OS implementation and configuration. Zenroom implements isolation at the VM level, independent of host platform capabilities.

Memory isolation: Zenroom allocates a fixed memory pool at initialization and never requests additional memory during contract execution. All allocations come from this pool managed by an embedded memory manager. The VM cannot access memory outside this pool. Buffer overflows cannot escape to host memory because the VM has no pointers to host memory. Use-after-free vulnerabilities cannot affect host state because the VM's memory space is completely separate.

Computational bounds: Contracts run with defined computational limits. Zenroom enforces a maximum iteration count and maximum memory usage specified at initialization. These bounds prevent resource exhaustion attacks. A malicious contract that attempts infinite loops or exponential memory consumption hits the limit and terminates cleanly. The limits are not advisory, they are enforced by the VM's execution engine.

No dynamic code execution: The VM does not support eval or similar dynamic code execution mechanisms. All operations are defined at contract parse time. This prevents code injection attacks entirely. A contract cannot construct code strings and execute them. All execution paths are visible in the contract text.

Side-channel resistance: Complete side-channel resistance is difficult to achieve in software running on general-purpose hardware. Zenroom's approach is to minimize timing variation in cryptographic operations through constant-time implementations where feasible and to provide clear documentation of what guarantees exist. The deterministic execution model helps here because timing differences between platforms are not hidden, they are measurable and documented.

State wiping: Zenroom clears sensitive data from memory on contract termination using explicit wiping functions. This reduces the window for memory scraping attacks. The fixed memory pool simplifies wiping because the VM knows exactly which memory regions to clear.

The isolation mechanisms work together to create defense in depth. Even if one mechanism failed, the others contain damage. More importantly, the isolation is verifiable. You can inspect the VM initialization code and confirm that it requests no OS privileges, opens no file descriptors, and creates no network sockets. The absence of capabilities is easier to verify than the correct implementation of permission checks.

3.4 Deterministic Execution as Security Feature

Determinism is typically discussed as a reliability feature for distributed systems. Nodes must produce identical results to reach consensus. This is accurate but incomplete. Determinism is also a security feature that prevents subtle attacks and enables verification.

Preventing parser differentials: In systems with multiple implementations, subtle differences in parsing can create security vulnerabilities. An attacker crafts input that one implementation interprets as benign while another interprets as malicious. Both implementations accept the input but process it differently. The attacker exploits this differential to bypass security checks. Zenroom eliminates this threat through deterministic parsing. There is one parser implementation. All platforms run identical code from the same source. Parser differentials cannot exist.

Enabling verification: Deterministic execution allows participants in a distributed system to verify each other's work by re-executing the same contract and comparing results. Non-deterministic execution requires trusting that other participants executed correctly because you cannot reproduce their results. Determinism converts trust into verification.

Reproducible audits: Security audits of non-deterministic systems must account for execution variance. Auditors cannot be certain that the specific execution path they analyzed will occur in production. Deterministic execution means audit results apply

directly to production behavior. What you audit is what executes.

Eliminating race conditions: Concurrent execution creates race conditions where program behavior depends on timing. These are notoriously difficult to test and debug. Zenroom's strictly sequential execution model eliminates race conditions entirely. Contract execution has no concurrency, no threads, no async operations. This constraint may seem limiting but it prevents an entire class of vulnerabilities.

Consistent random number generation: Cryptographic operations require randomness but deterministic execution seems to preclude random behavior. Zenroom resolves this through seedable deterministic random number generation. The same seed produces the same sequence of random numbers. For cryptographic operations requiring true randomness, the caller provides entropy as an explicit input. This makes randomness sources visible and auditable rather than hidden in OS RNG implementations.

The determinism guarantee is testable. You can execute the same contract on different platforms and verify byte-identical outputs. This is not a theoretical property but a measurable characteristic of every Zenroom deployment.

3.5 Memory Safety and Side-Channel Resistance

Memory safety vulnerabilities account for a large fraction of exploits in systems software. Buffer overflows, use-after-free, double-free, null pointer dereferences, and similar errors remain common despite decades of research into prevention. Zenroom addresses memory safety through a combination of safe implementation language, bounded allocations, and explicit lifetime management.

Implementation language choice: Zenroom's core is implemented in C with Lua as the high-level scripting layer. C is not memory-safe by default but its behavior is well-specified and auditable. The cryptographic primitives use the AMCL library which has been extensively audited. The Lua VM provides automatic memory management with garbage collection for the contract execution layer. This combination allows manual control where performance matters while providing safety where complexity resides.

Bounded memory model: The three-compartment memory model (IN-ACK-OUT) enforces clear ownership semantics. Data cannot leak between compartments. The Given phase copies validated input into ACK, it does not pass references that could allow contracts to modify raw input. The Then phase copies results to OUT, preventing contracts from modifying computation state after output generation. These copies have performance costs but eliminate aliasing bugs.

Garbage collection: Lua's garbage collector reclaims memory automatically in the contract execution layer. This prevents use-after-free and double-free bugs that plague manual memory management. The collector is deterministic in the sense that collection runs at predictable points in execution, not triggered by memory pressure that varies between platforms.

Side-channel considerations: Complete side-channel resistance requires hardware support that general-purpose systems lack. Zenroom's approach is pragmatic: use constant-time implementations for sensitive operations where possible, document which operations have timing variations, and provide guidance on deployment patterns that mitigate observable channels.

The cryptographic primitives from AMCL include constant-time big integer arithmetic to resist timing attacks. The Zencode interpreter's statement execution time varies based on contract structure but this variation does not leak key material because the interpreter operates on public contract text, not on secret values.

Cache-based side channels are difficult to eliminate in software. Zenroom mitigates these through data-independent control flow in cryptographic operations. The signature verification code executes the same sequence of operations regardless of whether verification succeeds or fails. This prevents attackers from learning about keys through cache timing differences.

Memory safety and side-channel resistance exist on a spectrum. Zenroom's position on this spectrum is conservative. We implement protections that are verifiable and do not depend on compiler optimizations or runtime behavior that could change. Where complete protection is not achievable, we document the limitations so operators can make informed deployment decisions.

4 Language-Theoretic Security Foundation

Language-theoretic security, as formalized by researchers at IEEE and supported by DARPA's work on resilient software systems, provides a framework for understanding why most software vulnerabilities exist and how to prevent them systematically. The core insight is that security failures in input processing stem from the gap between what inputs a program should accept and what inputs it actually processes. Most programs have an informal notion of valid input implemented through scattered validation checks. This approach inevitably leads to exploitable edge cases.

The LangSec community has documented patterns of failure across decades of exploits. Web servers parse HTTP headers inconsistently. Certificate validators mishandle ASN.1 encoding. Image processors trust format metadata. PDF readers execute embedded code. File archive tools follow symlinks. Each exploit represents a case where the program processed input that should have been rejected but was not, because validation was incomplete, incorrectly ordered, or simply absent in some code path.

Zenroom's design applies LangSec principles from the foundation. The Zencode grammar is specified formally. The parser is a recognizer that either accepts or rejects inputs based on this grammar. Processing happens only after recognition succeeds. This separation is enforced structurally through the phase-based execution model. You cannot write a Zencode contract that processes unvalidated input because unvalidated data never reaches the processing phase.

4.1 The Four Threats in Language Design

The LangSec community identifies four categories of threats that arise from poor language design. Each represents a way that implementations diverge from specifications, creating exploitable inconsistencies. Zenroom's architecture addresses each threat explicitly.

4.1.1 Ad-hoc Input Validity

Most software does not formally specify what constitutes valid input. Instead, validation is implemented through checks scattered across the codebase. A function might verify that a string length is positive. Another function checks for null terminators. A third assumes the string is valid UTF-8. None of these checks constitute a complete specification of valid input. The combination of checks was never verified to be sufficient. New code paths added during maintenance may bypass checks. The result is that programs accept inputs their designers never intended.

Zenroom addresses this through declarative schema validation. Before any contract executes, all inputs must match declared schemas. A schema specifies the complete structure of valid input: field names, types, encoding, constraints. The schema language is the same human-readable format as contracts. An auditor reviewing a Zencode implementation sees explicitly what inputs are accepted.

The schemas are not documentation that might diverge from implementation. They are executed specifications. The validator parses input according to the schema and constructs typed data structures. If parsing fails at any point, execution halts. The contract code receives only data that exactly matches the schema. This eliminates the class of vulnerabilities where attackers craft inputs that pass some validation checks but violate assumptions in processing code.

Consider a concrete example. A contract processes digital signatures. The schema specifies that the public key must be a point on a particular elliptic curve, encoded in compressed format, exactly 33 bytes. The validator checks point-on-curve membership cryptographically. An input that is 33 bytes but not a valid curve point is rejected before the contract sees it. The contract author does not need to implement this check. The contract author does not need to remember to implement this check. The check is automatic because it is part of the schema specification.

4.1.2 Parser Differentials

Systems with multiple implementations of the same specification create opportunities for parser differential attacks. An attacker crafts input that one implementation interprets differently than another. Both implementations believe they are correctly following the specification but subtle ambiguities allow different interpretations. The attacker exploits this difference to bypass security checks.

The classic example is certificate validation. A certificate authority sees a domain name in a certificate signing request and verifies that the requester controls that domain. The CA's parser interprets the domain name one way. Later, clients validate the issued certificate using different parsers that interpret the domain name differently. The attacker obtains a certificate for a domain they do not control because the CA and clients disagree on what domain the certificate specifies.

Parser differentials have affected TLS implementations, web browsers, email systems, and blockchain consensus protocols. They are difficult to detect because each implementation appears correct in isolation. The vulnerability emerges from the interaction between implementations.

Zenroom eliminates parser differentials by having exactly one parser implementation. Zencode contracts execute on the Zenroom VM. There is no alternative implementation to disagree with. All platforms run the same parser compiled from the same source code. Different hardware architectures and operating systems are handled through platform-specific compilation but the parser logic is identical.

This approach has a cost. Organizations cannot implement their own Zencode parsers for integration purposes. They must use the Zenroom VM or accept that their implementation might not match the canonical behavior. This restriction is intentional. Allowing multiple implementations would reintroduce the parser differential threat that the single-implementation architecture eliminates.

The deterministic execution guarantee extends this protection to the entire processing pipeline. Not only does parsing produce identical results, all subsequent computation produces identical results. Two parties executing the same contract with the same inputs can verify that they reached the same conclusion by comparing outputs. If outputs differ, one party deviated from the specification or inputs differed.

4.1.3 Recognition-Processing Separation

Most programs mix input validation with processing. Code reads input, checks a property, processes based on that property, reads more input, checks another property, processes further. This interleaving makes it difficult to reason about what inputs the program accepts. It also creates race conditions where input changes between validation and use, or where one part of the program validates

input that another part later processes without revalidation.

The LangSec principle of recognition before processing requires clean separation. The recognizer determines whether input is valid according to a formal grammar. If recognition succeeds, the recognizer produces a data structure representing the parsed input. Only this data structure is passed to processing code. The processor never sees raw input. This separation ensures that processing operates only on validated data.

Zenroom implements this separation through the Given-When-Then phase structure. The Given phase is recognition. Schemas define valid input structure. The validator parses input and constructs typed objects in the ACK memory compartment. The raw input in the IN compartment is read-only during this phase. Invalid input causes execution to halt in the Given phase before any processing begins.

The When phase is processing. It operates on the typed objects created during recognition. It has no access to raw input. It cannot parse new input or reinterpret existing input differently. All processing operates on data that has already been validated.

The Then phase is output rendering. It takes computation results and formats them for output. It does not perform validation or processing. It is a pure transformation from internal representation to external format.

This separation is enforced by the VM memory model. The Given phase has read-only access to IN and write access to ACK. The When phase has read-write access to ACK only. The Then phase has read access to ACK and write access to OUT. A contract cannot bypass these restrictions because they are implemented by the VM, not by contract code that could contain mistakes.

The separation also simplifies security analysis. Auditors can verify that the Given phase implements correct validation by examining schemas. They can verify that the When phase implements correct processing by examining contract logic. They do not need to trace complex control flows to ensure that validation always occurs before processing because the phase structure enforces this ordering.

4.1.4 Specification Drift Prevention

Software evolves. New features are added. Edge cases are discovered and handled. Performance is optimized. Each change risks divergence between the implementation and any specification that exists. If the specification is informal or lives in documentation separate from code, drift is inevitable. Developers update code without updating documentation. Tests cover implementation behavior that may not match original intent.

The result is that specifications become unreliable over time. What the system does and what the specification says it does diverge. Security properties that were verified against the specification may no longer hold for the implementation. This drift is not malicious. It emerges from the normal evolution of complex systems where specifications and implementations are separate artifacts maintained through separate processes.

Zenroom addresses specification drift through the LangSec approach of making the language specification the implementation. Zencode is specified as a grammar. The parser implements this grammar directly through syntax-directed translation. There is no intermediate specification document that could drift. The grammar is the specification.

When new Zencode statements are added, they must be defined in the grammar. The parser must be extended to recognize them. The implementation of new functionality requires explicit grammar extension. This coupling prevents drift. You cannot add functionality without updating the specification because the specification is the grammar that the parser enforces.

Zencode contracts are also specifications. A contract describes what operations to perform on what data. The contract text is the specification of the computation. There is no separate design document that describes what the contract should do. The contract is both specification and implementation. This eliminates specification drift at the contract level.

The approach has limitations. It works when the specification can be expressed as a grammar and when processing can be implemented through syntax-directed translation. Not all systems fit this model. But for the domain Zenroom targets, cryptographic operations on structured data, the approach is well-suited. The specifications we need to write are precisely the kind that map naturally to grammars and schemas.

4.2 Non-Turing Complete by Design

Zencode is deliberately not Turing complete. It cannot express arbitrary computations. There is no general recursion. There is no unbounded iteration. Loops have maximum iteration counts specified at VM initialization. This limitation is not an oversight but a design choice with security implications.

Turing completeness means a language can express any computable function given sufficient time and memory. This generality is valuable for general-purpose programming but it comes with costs. Turing complete languages have the halting problem. You cannot determine in general whether a program will terminate or run forever. This makes resource exhaustion attacks trivial. An attacker writes a contract that appears benign but contains an infinite loop. The VM executing this contract cannot detect that the loop is infinite without actually running it forever.

Turing incomplete languages can guarantee termination. Zencode does this through several mechanisms:

Bounded iteration: The foreach construct allows iteration over collections but the maximum iteration count is fixed at VM initialization. A contract cannot loop more than this maximum number of times. The VM tracks iterations and terminates execution if the limit is reached.

No general recursion: Zencode does not support recursive function calls. Each statement executes once per traversal of the contract. There is no mechanism for a statement to invoke itself directly or indirectly.

Acyclic control flow: The branch construct for conditional execution does not support backward jumps. Control flows forward through the contract. Once a section executes, it does not execute again in the same contract execution.

These restrictions ensure that every Zencode contract terminates in bounded time. The execution time is proportional to the contract size and the maximum iteration count. You can compute an upper bound on execution time before running a contract. This makes denial-of-service through resource exhaustion much harder. An attacker cannot craft a contract that appears short but executes for an unbounded time.

The restriction to non-Turing complete computation also simplifies analysis. Many verification techniques work only on restricted computational models. Model checking, static analysis, and symbolic execution all become more tractable when the language cannot express arbitrary computation. Zenroom does not currently implement these verification techniques but the language design does not preclude them.

The practical impact of non-Turing completeness is less limiting than it might appear. Most cryptographic operations are not recursive. Signature creation, encryption, hash computation, credential verification all follow straightforward algorithms with known termination properties. The computations that Zencode needs to express fit naturally within the bounded model.

There are computations you cannot express in Zencode. You cannot implement a parser for an arbitrary context-free language. You cannot write an interpreter for another language. You cannot implement search algorithms with unbounded depth. If your use case requires these capabilities, Zencode is not suitable. For the domain of cryptographic operations on structured data, the limitations are rarely binding.

4.3 Formal Grammar and Syntax-Directed Translation

Zencode is specified as a formal grammar using a syntax-directed translation scheme. The grammar defines valid Zencode contracts. The translation scheme maps grammatical constructs to VM operations. This formalization serves multiple purposes: it provides a precise specification of the language, it enables parser generation from the grammar, and it makes the language amenable to formal analysis.

The grammar is context-free with some context-sensitive restrictions enforced during semantic analysis. Context-free grammars sit in the middle of the Chomsky hierarchy. They are more expressive than regular languages but less expressive than context-sensitive languages. This positioning is intentional. Context-free grammars are well-understood, parser generators for them are mature, and the parsing algorithms are efficient.

The context-free backbone of Zencode handles the statement structure. A contract is a sequence of statements. Each statement begins with a keyword (Given, When, Then, If, Foreach) followed by a pattern. Patterns contain literal text and variable placeholders marked by single quotes. The parser matches statement text against registered patterns and extracts variable values.

Context-sensitive restrictions enforce phase ordering and memory access rules. A contract must begin with scenario declaration. Given statements must precede When statements. When statements must precede Then statements. These rules cannot be expressed in pure context-free grammar but are enforced during parsing through state machine transitions.

The syntax-directed translation maps each recognized statement to a function call. When the parser matches a statement pattern, it invokes the corresponding function with extracted variables as arguments. This translation is direct. There is no intermediate representation. The parser output is a sequence of function calls to be executed. This directness eliminates a source of bugs. Optimizers and code generators introduce complexity where bugs hide. Zenroom avoids this complexity through simple translation.

The formalization also enables precise specification of language extensions. New Zencode scenarios are added by registering new statement patterns and their corresponding functions. The pattern registration is explicit and checked for ambiguity. If two patterns could match the same statement, the parser reports an error at definition time, not at execution time when the ambiguity could cause security issues.

The grammar approach also makes Zencode parseable by tools beyond the Zenroom VM. Static analysis tools can parse contracts without executing them. Documentation generators can extract statement patterns and produce reference materials automatically. The formal grammar is not hidden inside parser implementation code but is explicitly specified in a form that tools can consume.

This formalization does not make Zencode a heavyweight academic language. The grammar is simple enough that humans can read contracts and understand what they do. The formalization exists to ensure that human understanding matches machine execution, not to impose complexity.

The goal is comprehensibility backed by rigor, not rigor for its own sake.

5 The Zenroom Virtual Machine Architecture

The Zenroom Virtual Machine is not a general-purpose computing environment. It is a process VM designed specifically to execute cryptographic contracts in hostile environments where the code, the data, or both cannot be trusted. This specialization allows architectural decisions that would be inappropriate for general computation but are essential for security-critical cryptographic operations.

The VM architecture reflects eight years of production deployment feedback. Early versions focused on proving the concept of human-readable cryptographic contracts. Current versions address the operational requirements of systems processing real user credentials, financial transactions, and identity verification. The architecture has evolved but the core principles remain unchanged: isolation, determinism, and verifiability.

5.1 Process VM Design Principles

Zenroom implements a process virtual machine rather than a system VM. This distinction matters. System VMs like VMware or VirtualBox virtualize entire operating systems. Process VMs like the JVM or Zenroom virtualize single applications. System VMs provide isolation through hardware virtualization and hypervisors. Process VMs provide isolation through language-level constraints and runtime checks.

The process VM approach offers specific advantages for cryptographic contract execution:

Startup time: A system VM boots an operating system, initializing device drivers, filesystems, and network stacks. This takes seconds or minutes. A process VM initializes a runtime environment, loading only the components needed for contract execution. Zenroom starts in milliseconds. For systems processing thousands of contracts per second, this difference is not a detail but a requirement.

Resource footprint: System VMs allocate gigabytes of memory and virtual disk space. Process VMs allocate megabytes. Zenroom's entire binary including cryptographic primitives, language parser, and VM runtime compiles to under 3MB. This allows deployment on resource-constrained devices like mobile phones and embedded systems where system VM approaches are not feasible.

Attack surface: System VMs must virtualize hardware with all its complexity: interrupt handling, DMA, memory management units. Process VMs virtualize language execution with controlled semantics. Every feature a system VM provides is potential attack surface. Zenroom provides only what cryptographic contracts need: memory allocation, computation, and defined input-output channels.

Determinism: System VMs execute operating systems designed for interactive use. Timing, scheduling, and resource allocation vary based on system load. Process VMs can enforce deterministic execution because they control the entire execution environment. Zenroom guarantees that the same contract with the same inputs produces identical outputs across all platforms.

The process VM approach has costs. You cannot run arbitrary operating systems or applications inside Zenroom. You cannot access hardware devices or network resources. These limitations are intentional. They are features, not bugs. A cryptographic contract that needs network access or hardware interaction is a contract that has violated isolation boundaries.

5.2 Memory Model: IN-ACK-OUT Compartmentalization

Zenroom divides memory into three isolated compartments corresponding to the three execution phases. This compartmentalization is not a programming convenience but a security mechanism enforcing separation of concerns.

IN compartment: Holds raw input data as received from the calling application. During the Given phase, the parser reads from IN to validate and decode input. The IN compartment is read-only after initialization. Contract code cannot modify raw input. This prevents an entire class of attacks where malicious code modifies input during validation to bypass checks.

The IN compartment contains data passed through the keys and data parameters when invoking Zenroom. These inputs are typically JSON documents but Zenroom also accepts CBOR and MessagePack encodings. The parser does not trust the encoding declaration. It validates structure, checks schemas, and rejects malformed input before any processing begins.

ACK compartment: Holds validated, typed data structures created during the Given phase and modified during the When phase. This is the working memory where contract logic operates. The ACK compartment is read-write during When phase execution. Contract statements can create new objects, modify existing ones, and delete objects no longer needed.

The transition from IN to ACK involves validation and type conversion. A string representing a public key in IN becomes a validated elliptic curve point in ACK. A JSON array of numbers in IN becomes a typed array of big integers in ACK. This conversion enforces data schemas and ensures that processing operates only on validated data.

OUT compartment: Holds output data formatted for the calling application. During the Then phase, contract statements copy computation results from ACK to OUT,

applying encoding and formatting. The OUT compartment is write-only during Then phase. Previous output cannot be read or modified once written. This prevents contracts from examining output and modifying behavior based on what was already produced.

The compartmentalization is enforced by the VM runtime. Zencode statements are mapped to functions that have different access permissions in different phases. A Given phase function receives read access to IN and write access to ACK. A When phase function receives read-write access to ACK only. A Then phase function receives read access to ACK and write access to OUT. These permissions are not checked at runtime but are structural properties of how functions are registered in the VM.

This memory model prevents several attack patterns:

Time-of-check to time-of-use (TOCTOU): In systems where validation and processing share memory, attackers can modify data between validation and use. Zenroom's model makes this impossible because validated data is copied from IN to ACK. Subsequent changes to IN do not affect ACK.

Output tampering: In systems where output accumulates in writable memory, contracts can examine partial output and modify subsequent output to create inconsistencies. Zenroom's write-only OUT compartment prevents this.

State leakage: In systems where memory persists between executions, data from one contract can leak to subsequent contracts. Zenroom wipes all compartments on termination and starts fresh for each execution.

The compartmentalization does impose overhead. Data is copied between compartments rather than passed by reference. This copying has performance cost but the cost is predictable and bounded. The security benefits outweigh the performance impact for the use cases Zenroom targets.

5.3 State Machine and Phase Transitions

Zencode execution is controlled by a finite state machine that enforces phase ordering and valid statement sequences. The state machine is not a runtime optimization but a security mechanism that prevents malformed contracts from executing undefined behavior.

The state machine tracks the current execution phase and validates that each statement is permitted in that phase. The states are:

- init: Initial state before any contract execution
- rule: Processing rule directives
- scenario: Loading scenario modules
- given: Validating and loading input
- when: Processing and transforming data
- if: Conditional execution branch
- whenif: Processing inside conditional branch
- thenif: Output inside conditional branch
- endif: End of conditional branch
- foreach: Loop iteration
- whenforeach: Processing inside loop
- endforeach: End of loop
- then: Formatting and printing output

Valid transitions are defined explicitly. A contract in the given state can transition to when or then but not back to rule. A contract in the when state can transition to if, foreach, or then but not to given. Attempts to transition invalidly halt execution with an error.

The state machine also tracks nesting depth for control structures. If blocks and foreach loops can nest but only to implementation-defined limits. The default maximum nesting is configurable but the mechanism for enforcing limits is not. When a contract exceeds nesting limits, execution halts. This prevents stack overflow attacks and resource exhaustion.

Each state transition triggers validation hooks. When transitioning from given to when, the VM verifies that all required input has been loaded and validated. When transitioning to then, the VM ensures that output operations are permitted. These hooks catch contract errors early rather than allowing execution to proceed to an inconsistent state.

The state machine implementation uses a lightweight library that generates explicit transition functions. This is not table-driven state machine with runtime dispatch overhead. Each transition is a direct function call with inlined validation. The state machine adds minimal runtime cost while providing strong execution guarantees.

The state machine prevents several categories of contract errors:

Phase violations: Contracts that attempt to load input in the When phase or perform cryptographic operations in the Given phase are rejected. Phase separation is enforced mechanically.

Unbalanced control structures: Contracts with if blocks lacking endif or foreach loops lacking endforeach are detected during parsing before execution begins.

Infinite loops: The state machine tracks iteration count and terminates contracts that exceed the configured maximum. This is checked on every loop iteration, not after the fact.

5.4 No External Dependencies: Attack Surface Minimization

Zenroom has zero external runtime dependencies. The binary is statically linked. It does not dynamically load libraries at runtime. It does not call external programs.

It does not make system calls beyond the minimal set required for memory allocation and process termination.

This is not just aggressive static linking. It is a conscious decision about trust boundaries. Every external dependency is code you must trust. Libraries can contain vulnerabilities. System calls can have unexpected behavior. Runtime linking can be intercepted. Zenroom eliminates these risks by eliminating the dependencies.

The cryptographic primitives use Apache Milagro Crypto Library (AMCL) compiled directly into the Zenroom binary. AMCL is not a dynamically linked library but source code included in the build. This ensures that the version of AMCL is fixed at compile time and cannot vary between deployments. Different organizations running Zenroom 5.0 are running identical cryptographic code, not different versions of libraries pulled from system package managers.

The Lua VM that executes Zencode is similarly embedded. Zenroom uses Lua 5.4 compiled from source. It does not use the system Lua installation. This avoids version mismatches and ensures that Zencode semantics are consistent across platforms. Lua's garbage collector, string handling, and numeric operations behave identically whether Zenroom runs on Android or Linux or WebAssembly.

The zero dependency approach has consequences:

Binary size: Statically linking all dependencies produces larger binaries than dynamic linking. Zenroom is 3MB rather than the few hundred kilobytes a dynamically linked binary might be. This is a conscious tradeoff. Predictability is worth the space cost.

Update complexity: Security updates to dependencies require rebuilding Zenroom rather than updating system libraries. This is more work but it is also more controlled. You decide when to adopt updates rather than having them imposed by operating system package updates.

Platform portability: Without external dependencies, porting to new platforms requires only a C compiler and basic POSIX support. Zenroom runs on platforms where installing shared libraries would be difficult or impossible.

The minimal system call interface deserves specific attention. Zenroom makes no filesystem operations, no network operations, no process creation, no signal handling. On Linux, it can run under seccomp to enforce these restrictions at the kernel level. The seccomp profile permits only memory allocation, terminal I/O, and exit. Any attempt to call forbidden system calls terminates the process immediately.

This restriction is verifiable. You can inspect the Zenroom source and confirm that no filesystem access occurs. You can run it under system call tracing (strace on Linux) and verify that only permitted calls are made. The absence of capability is easier to verify than the correct implementation of permission checks.

5.5 Deterministic Random Number Generation

Cryptographic operations require randomness. Key generation needs random seeds. Nonces need random values. Padding schemes need random bytes. Yet Zenroom guarantees deterministic execution. These requirements appear contradictory but are reconciled through explicit randomness management.

Zenroom uses a cryptographically secure pseudorandom number generator (CSPRNG) based on SHA-512. The generator is deterministic: given the same seed, it produces the same sequence of random bytes. This allows deterministic contract execution while still providing cryptographically strong randomness.

The seed can be provided in three ways:

External seed: The calling application provides entropy through a configuration parameter. This entropy seeds the CSPRNG before contract execution begins. Two executions with the same seed produce identical results.

Platform entropy: If no external seed is provided, Zenroom reads from the platform entropy source (urandom on Unix, CryptGenRandom on Windows). This provides non-deterministic randomness suitable for production use but prevents reproducible execution.

Contract-specified seed: Zencode contracts can derive keys from passwords or other input data. The KDF operations use the input as a seed, making key generation deterministic for the same input.

The CSPRNG state is private to each Zenroom execution. One contract execution cannot observe or influence the CSPRNG state of subsequent executions. The state is wiped on process termination along with all other memory.

For operations requiring public verifiable randomness, Zenroom supports deterministic derivation from contract inputs. Signature schemes use RFC 6979 deterministic ECDSA, where the nonce is derived from the message and private key rather than generated randomly. This makes signatures deterministic and eliminates nonce reuse vulnerabilities.

The approach to randomness reflects a broader principle: make the sources of non-determinism explicit and controllable. If execution varies between runs, that variation should be traceable to explicit inputs, not hidden sources of entropy. This principle aids debugging, testing, and verification.

5.6 Garbage Collection and Memory Management

Zenroom uses two memory management strategies depending on the execution layer. The C implementation layer uses manual memory management with explicit allocation and deallocation. The Lua execution layer uses automatic garbage collection. This hybrid approach balances performance, safety, and predictability.

The C layer manages cryptographic primitives and VM infrastructure. These components require careful memory handling because they process sensitive data. Cryptographic keys must be wiped after use. Temporary buffers must not leak. Manual management allows explicit control over memory lifetimes and contents.

The C memory manager is a fixed-size pool allocator. Zenroom allocates a memory pool at initialization sized based on VM configuration. All subsequent allocations come from this pool. The pool is never expanded during execution. If the pool is exhausted, allocation fails and execution halts with an error. This prevents memory exhaustion attacks and makes memory usage predictable.

The Lua layer manages contract data structures. Lua provides automatic garbage collection using an incremental mark-and-sweep collector. The collector runs periodically during contract execution, reclaiming memory from objects no longer referenced. This frees contract authors from manual memory management while maintaining safety.

The garbage collector is deterministic in the sense that collection runs at predictable points in execution. The collector is triggered when memory usage exceeds thresholds. These thresholds are fixed at VM configuration time. Given the same input and the same VM configuration, garbage collection occurs at the same points in execution across all platforms.

Memory wiping occurs at multiple points:

After cryptographic operations: Functions that create or use keys explicitly wipe their stack frames and temporary buffers before returning. This limits the window where sensitive data exists in memory.

After phase transitions: When transitioning from Given to When or When to Then, the VM can optionally wipe the previous compartment. This is configurable because wiping has performance cost.

On process termination: The entire memory pool is wiped when Zenroom exits. This prevents memory scraping attacks that might recover sensitive data from process memory after termination.

The memory management approach does not prevent all possible side channels. Garbage collection timing varies based on allocation patterns and collection could leak information about computation. Wiping memory prevents recovery of values but does not prevent timing analysis during computation. These limitations are documented. Zenroom makes no claims about resistance to side-channel attacks by sophisticated adversaries with physical access.

6 The Zencode Language

Zencode is a domain-specific language for cryptographic operations that reads like structured English. This is not marketing language. Contracts written in Zencode can be read and understood by stakeholders with no programming experience. The syntax is constrained enough to be unambiguous while remaining natural enough to be comprehensible. This balance is the result of careful design and eight years of refinement based on production use.

The language design started with a question: what would a cryptographic contract look like if it had to be readable in a courtroom or regulatory hearing? Not pseudocode that requires translation. Not documentation that might diverge from implementation. The actual executable code, readable by non-technical stakeholders. This constraint shaped everything about Zencode.

6.1 Behavior-Driven Development Syntax

Zencode adopts the syntax of Behavior-Driven Development (BDD), specifically the Given-When-Then structure popularized by Cucumber and similar testing frameworks. BDD syntax was designed to bridge technical and non-technical stakeholders in software projects. Developers write executable specifications that business analysts can read and verify. This alignment of interests makes BDD syntax well-suited for cryptographic contracts where legal, compliance, and technical concerns intersect.

The BDD influence is visible in statement structure. Zencode contracts consist of statements beginning with keywords: Given, When, Then, And. These keywords organize statements into phases corresponding to contract execution stages. The structure is rigid by design. Flexibility in statement ordering would require complex grammar and ambiguous parsing. Rigidity enables simple parsing and clear semantics.

Consider a minimal Zencode contract:

```
Given I have a 'string' named 'message'
When I create the hash of 'message'
Then print the 'hash'
```

This contract can be read by someone unfamiliar with programming. The Given statement declares input. The When statement performs computation. The Then statement specifies output. The contract maps directly to its purpose: hash a message and return the result.

The BDD syntax provides scaffolding for statement composition. The And keyword allows multiple statements in a phase without repeating the phase keyword:

```
Given I have a 'string' named 'message'
And I have a 'string' named 'salt'
When I create the hash of 'message'
And I append 'salt' to 'hash'
Then print all data
```

This avoids repetitive Given statements while maintaining clear phase boundaries. The And keyword is syntactic sugar. Internally, "And I have" is processed identically to

"Given I have". The distinction exists for human readability, not parser requirements.

The BDD syntax also establishes conventions for variable references. Variables are enclosed in single quotes. This makes variable names visually distinct from statement keywords and clearly marks where data flows between statements. The choice of single quotes rather than dollar signs or other programming conventions maintains the natural language appearance while providing unambiguous parsing.

6.2 The Given-When-Then Paradigm

The three-phase structure is not arbitrary. It maps to the natural flow of computation: input, processing, output. More importantly, it maps to the security requirements of cryptographic contract execution.

Given phase: Input validation and loading. Every object entering contract execution must be declared and validated in this phase. The Given phase corresponds to the recognition phase in LangSec terminology. Raw input is parsed according to declared schemas, type-checked, and converted to internal representations. Invalid input halts execution before processing begins.

The Given phase enforces explicit declaration. You cannot process data you have not loaded. You cannot load data without specifying its type and encoding. This explicitness prevents an entire class of vulnerabilities where implicit assumptions about input lead to unexpected behavior.

Given statements declare what exists rather than creating new objects. "Given I have a 'string' named 'message'" states that the input contains a string called message. If the input does not contain such a string, execution fails. This declaration-before-use pattern catches errors early and makes contract requirements explicit.

When phase: Computation and transformation. All data manipulation happens in this phase. Cryptographic operations, arithmetic, string manipulation, object creation, and control flow all occur in When statements. The When phase operates on validated data from the Given phase. It cannot access raw input or modify the output being constructed.

The When phase allows object creation and modification. Unlike Given which declares existing objects, When creates new objects through computation. "When I create the hash of 'message'" generates a new object called hash from existing data. The distinction between declaration and creation clarifies data flow and makes contract logic easier to follow.

When statements can be conditional or iterative. If blocks allow branching based on runtime conditions. Foreach blocks allow iteration over collections. These control structures nest within the When phase but maintain phase semantics. Code inside an if block in When cannot perform Given-phase loading or Then-phase output.

Then phase: Output formatting and printing. This phase takes computation results from When and formats them for the calling application. Then statements select what data to output and how to encode it. "Then print the 'hash'" outputs the hash object. "Then print all data" outputs everything in the computation workspace.

The Then phase is write-only with respect to output. Once data is written to output, it cannot be read back or modified. This prevents contracts from examining partial output and changing behavior based on what has been printed. Output is accumulated and returned atomically when contract execution completes successfully.

The three-phase structure enforces separation of concerns at the language level. You cannot mix input validation with computation or computation with output formatting. Each phase has a defined purpose and limited scope. This constraint simplifies reasoning about contract behavior and prevents subtle bugs that emerge when concerns mix.

6.3 Advanced Control Flow: Branching and Iteration

Zencode is not a straight-line language. Contracts can branch on conditions and iterate over collections. These control structures introduce complexity but that complexity is bounded and explicit.

Conditional execution:

```
If I verify the 'signature'
When I create the 'credential'
Then print the 'credential'
Endif
```

The If statement evaluates a condition. If true, the statements between If and Endif execute. If false, they are skipped. The condition must be a statement that returns a boolean result. In this example, signature verification either succeeds or fails. The credential is created only if verification succeeds.

If blocks can contain When and Then statements. The phase structure is maintained within branches. You cannot put Given statements inside an If block because input validation cannot be conditional. Either the input is valid or it is not. Conditional validation would create ambiguous contract semantics.

If blocks can nest but only to implementation-defined depth. The default maximum nesting is configurable at VM initialization. Deep nesting is typically a sign of complex logic better expressed differently. The nesting limit prevents stack overflow and makes contract complexity visible.

Bounded iteration:

```
Given I have a 'string array' named 'messages'
Foreach 'message' in 'messages'
When I create the hash of 'message'
Endforeach
```

```
Then print all data
```

The Foreach statement iterates over elements of a collection. For each element, the loop body executes with the element bound to a variable. The loop terminates when all elements have been processed or when the maximum iteration count is reached.

The maximum iteration count is set at VM initialization and cannot be changed at runtime. This prevents infinite loops and makes loop termination decidable. If a contract requires more iterations than the maximum, execution halts with an error. This is a limitation but it is a deliberate one. Unbounded iteration would make Zencode Turing complete and bring all the verification problems that entails.

Foreach loops can contain If blocks and If blocks can contain Foreach loops. The nesting is explicit in the syntax and bounded by configuration. The state machine tracks nesting depth and enforces limits.

The control structures are constrained in ways that make verification tractable. No backward jumps means control flow is acyclic. No goto or arbitrary branching means the control flow graph is simple. No break or continue means loop bodies always complete or fail entirely. These constraints limit expressiveness but they make contract behavior predictable.

6.4 Declarative Schema Validation

Every data object in Zencode has a schema. The schema defines the structure, encoding, and constraints of the object. Schemas are declarative specifications that the VM enforces automatically. Contract authors declare schemas in Given statements. The VM validates input against schemas and rejects anything that does not match.

Simple schemas:

```
Given I have a 'string' named 'username'
Given I have a 'number' named 'age'
Given I have a 'hex' named 'publicKey'
```

These statements declare simple objects with basic schemas. A string is UTF-8 text. A number is a floating-point value. A hex is binary data encoded as hexadecimal. The VM validates that the input object has the declared encoding and converts it to internal representation.

The encoding declaration serves two purposes. It tells the VM how to decode input and it documents what the contract expects. An auditor reading the contract sees explicitly that publicKey must be hex-encoded. This is not hidden in implementation code or inferred from context.

Array schemas:

```
Given I have a 'string array' named 'names'
Given I have a 'number array' named 'values'
```

Arrays have homogeneous element types. All elements of a string array must be strings. All elements of a number array must be numbers. The VM validates each element against the declared type and rejects arrays with mixed types.

This homogeneity constraint simplifies processing. Code that iterates over a string array knows every element is a string. No runtime type checking is needed. The constraint also catches errors where mixed-type data is passed inadvertently.

Dictionary schemas:

```
Given I have a 'string dictionary' named 'configuration'
```

Dictionaries are key-value maps. The schema declares the value type. All values in a string dictionary must be strings. Keys are always strings. Dictionaries allow flexible data structures while maintaining type safety.

Cryptographic schemas:

Cryptographic objects have complex internal structures. A public key is not just binary data but a specific mathematical object with properties like point-on-curve membership. Cryptographic schemas encode these requirements.

```
Given I have my 'keyring'
Given I have a 'verifiable credential' named 'diploma'
```

The keyring schema is defined by the ECDH scenario. It specifies the structure of cryptographic keys for different algorithms. The verifiable credential schema is defined by the W3C scenario. It specifies the structure of W3C Verifiable Credentials including signatures, proofs, and metadata.

These schemas are scenario-specific. Loading a keyring requires declaring a scenario that defines the keyring structure. Scenarios are Zencode modules that register statement patterns and schema definitions. The modularity allows extending Zencode with new cryptographic schemes without modifying the core language.

Schema validation is not optional. Every object must match its declared schema. The VM validates on input and on object creation. When a When statement creates a new object, the VM validates that the result matches the expected schema for that operation. This validation catches implementation errors where cryptographic operations produce malformed output.

The declarative schemas serve as executable documentation. Reading a contract's Given statements shows exactly what input structure the contract expects. Reading a When statement that creates an object shows what schema the result will have. The schemas are not comments that might be wrong. They are enforced specifications that the VM checks.

6.5 Error Handling and Execution Guarantees

Zencode contracts either succeed completely or fail completely. There is no partial success. If any statement fails, the entire contract fails and no output is produced. This all-or-nothing semantics simplifies error handling and provides clear execution guarantees.

When a statement fails, execution halts immediately. The VM does not attempt recovery or continue execution. It returns an error message describing what failed and where. The error message includes the line number of the failing statement and the reason for failure. This makes debugging tractable.

Error messages are structured and consistent. They follow a format that makes parsing and automated handling possible:

```
[!] Zencode line 12 pattern not found:
    Given I have a 'nonexistent' named 'object'
```

The message identifies the line number, the type of error, and the problematic statement. Applications calling Zenroom can parse these messages and take appropriate action. The structured format also helps humans diagnose problems quickly.

The execution guarantees are straightforward:

Atomicity: A contract either completes fully or produces no output. There are no partial results. This matches transaction semantics in databases and makes contract execution composable.

Determinism: The same contract with the same input produces the same output every time. No hidden state, no timing dependencies, no platform-specific behavior. This makes contract execution reproducible and verifiable.

Isolation: One contract execution cannot affect another. No shared state, no side effects, no persistent data. Each execution starts fresh and ends clean.

Termination: Every contract terminates in bounded time. No infinite loops, no unbounded recursion, no resource exhaustion. The bounds are configurable but they are enforced.

These guarantees are not best-effort or probabilistic. They are structural properties of the VM and language design. Applications can rely on them for system design and security analysis.

6.6 Pattern Matching and Statement Resolution

Zencode statements are resolved through pattern matching against a registry of known patterns. Each scenario registers patterns that map statement text to implementation functions. The parser matches input statements against registered patterns and invokes the corresponding functions.

Pattern structure:

A pattern consists of literal text and variable placeholders:

```
When I create the hash of ''
```

The literal text is "When I create the hash of". The variable placeholder is marked by single quotes. When the parser encounters a statement matching this pattern, it extracts the variable value and passes it to the implementation function.

Patterns can have multiple variables:

```
When I create the result of '' + ''
```

The parser extracts both variable values and passes them as arguments. The order of arguments matches the order in the pattern.

Pattern normalization:

Before matching, statements are normalized. Articles (a, an, the) are removed. Pronouns (I) are removed. Multiple spaces are collapsed to single spaces. The string is converted to lowercase. This normalization makes patterns more forgiving of minor phrasing variations while keeping the grammar unambiguous.

For example, these statements all match the same pattern:

```
When I create the hash of 'message'
When create hash of 'message'
when I CREATE the HASH of 'message'
```

The normalization removes stylistic differences while preserving semantic content. This makes Zencode more natural to write without introducing ambiguity.

Pattern disambiguation:

If multiple patterns could match the same statement, the parser reports an error at pattern registration time, not at execution time. Ambiguous grammars are rejected during scenario loading before any contracts execute. This ensures that every statement has exactly one meaning.

Disambiguation happens through explicit pattern definitions. Scenario developers must ensure their patterns do not overlap with existing patterns. If a new scenario introduces ambiguity, the scenario loading fails with a clear error message indicating which patterns conflict.

Extensibility through scenarios:

New statements are added by writing scenarios that register new patterns. A scenario is a Lua module that calls registration functions:

```
Given("I have a 'string' named ''", function(name)
  -- implementation
end)
```

This registers a Given-phase pattern and associates it with an implementation function. When the parser encounters a matching statement, it calls the function with extracted variables.

Scenarios can register multiple patterns for the same operation to provide synonyms:

```
Given("I have a 'string' named ''", load_string)
Given("I have a 'string' called ''", load_string)
```

Both patterns invoke the same function. This allows natural phrasing variations without duplicating implementation. The synonyms are explicit in the scenario code, making the equivalence visible to anyone reading the implementation.

The pattern matching approach makes Zencode extensible without modifying the core language. New cryptographic schemes, new data structures, new operations all extend through scenarios rather than language changes. This allows domain experts to add functionality while preserving the core guarantees of the language and VM.

7 Cryptographic Capabilities

Zenroom provides cryptographic primitives through the Apache Milagro Crypto Library (AMCL) and additional implementations for post-quantum algorithms and advanced schemes. The cryptographic capabilities are not a feature checklist but a carefully selected set of operations required by production systems implementing digital identity, verifiable credentials, and blockchain integration.

The selection reflects current cryptographic transitions. Organizations must support legacy algorithms for existing infrastructure while preparing for post-quantum migration and implementing privacy-preserving techniques for credential systems. Zenroom provides this range not through optional plugins but as integrated functionality verified as a unit.

7.1 Primitives and Curve Support

7.1.1 Elliptic Curves: NIST P-256, secp256k1, BLS12-381, Ed25519

Elliptic curve cryptography provides the foundation for most modern public-key operations. Zenroom supports multiple curves chosen for specific use cases rather than attempting comprehensive coverage.

NIST P-256: The secp256r1 curve mandated by FIPS standards and widely deployed in enterprise systems. Organizations with compliance requirements for federal standards need P-256 support. The curve is well-studied and implemented in hardware security modules and trusted platform modules. Zenroom's P-256 implementation interoperates with systems using ECDSA signatures and ECDH key exchange following NIST specifications.

secp256k1: The curve used by Bitcoin and Ethereum. Blockchain applications require secp256k1 for transaction signing and address derivation. The curve has received extensive cryptanalysis from the cryptocurrency community. Zenroom implements deterministic ECDSA signatures following RFC 6979 to eliminate nonce reuse vulnerabilities that have compromised Bitcoin wallets.

BLS12-381: A pairing-friendly curve enabling advanced cryptographic schemes. The curve supports BLS signatures with efficient aggregation, making it suitable for consensus protocols and threshold signatures. Zenroom uses BLS12-381 for verifiable random functions, aggregate signatures, and as the foundation for BBS+ credentials with selective disclosure.

Ed25519: The Edwards curve variant of Curve25519 providing fast signature operations with strong security guarantees. Ed25519 signatures are deterministic by design and resistant to side-channel attacks in software implementations. The curve is widely deployed in SSH, TLS, and messaging protocols. Zenroom's Ed25519 implementation follows the original specification without extensions or modifications.

Each curve is implemented through AMCL with constant-time operations for sensitive computations. Point multiplication uses algorithms resistant to timing attacks. Point validation checks ensure that inputs represent valid curve points before cryptographic operations. These checks prevent invalid curve attacks where adversaries provide points not on the declared curve to extract key material.

The multi-curve support creates implementation complexity. Each curve requires separate code paths for arithmetic operations. Zenroom addresses this through a factory pattern where curve-specific operations are registered at initialization. Contracts specify which curve to use through scenario declarations. The VM validates that all operations in a contract use compatible curves and rejects mixed-curve operations.

7.1.2 Pairing-Based Cryptography

Pairing-friendly curves like BLS12-381 support bilinear pairings: mathematical operations that map pairs of elliptic curve points to elements of a finite field. Pairings enable cryptographic schemes impossible with traditional elliptic curves.

Zenroom implements pairings for several applications:

BLS signatures: Short signatures that can be efficiently aggregated. Multiple signatures on different messages by different signers can be combined into a single signature verified in one operation. This reduces bandwidth and verification time in consensus protocols where many participants sign the same data.

Threshold signatures: A signature scheme where signing requires cooperation among multiple parties. A threshold of participants must collaborate to produce a valid signature but any subset larger than the threshold can sign. This enables distributed trust without single points of failure.

Verifiable random functions: Functions that produce random outputs with proofs that the output was correctly

computed. VRFs are used in consensus protocols for leader election and in credential systems for unlinkable pseudonyms.

Pairing operations are computationally expensive compared to standard elliptic curve operations. A single pairing computation takes longer than dozens of point multiplications. Zenroom contracts using pairings must account for this cost. The bounded execution model limits how many pairing operations a contract can perform, preventing resource exhaustion but also constraining what schemes are practical.

7.2 Post-Quantum Cryptography

Current elliptic curve cryptography will be vulnerable to quantum computers once such computers reach sufficient scale. NIST has standardized post-quantum algorithms and mandated transition timelines. Zenroom implements the NIST selections to prepare for this transition.

7.2.1 ML-KEM (Kyber) Key Encapsulation

ML-KEM, previously known as Kyber during the NIST competition, provides key encapsulation using lattice-based cryptography. The mechanism allows establishing shared secrets resistant to quantum attack. ML-KEM-768 provides security roughly equivalent to AES-192 against both classical and quantum adversaries.

Zenroom implements ML-KEM for hybrid key exchange. A contract can establish a shared secret using both ECDH and ML-KEM, combining them such that breaking either scheme is required to compromise the session. This hedges against both premature quantum computers and undiscovered weaknesses in lattice assumptions.

The ML-KEM implementation follows the NIST specification exactly. The parameter sets, encoding formats, and algorithmic steps match the standard. This ensures interoperability with other ML-KEM implementations and simplifies security analysis by auditors familiar with the standard.

ML-KEM operations are significantly slower than ECDH. Key generation, encapsulation, and decapsulation all take longer and produce larger outputs. A ML-KEM-768 public key is approximately 1.2KB compared to 33 bytes for a compressed P-256 point. Contracts using ML-KEM must account for these size and performance characteristics.

7.2.2 ML-DSA (Dilithium) Signatures

ML-DSA, formerly Dilithium, provides digital signatures using lattice-based cryptography. The scheme offers post-quantum security with relatively small signatures compared to other post-quantum signature schemes. ML-DSA-65 signatures are approximately 3KB, much larger than 64-byte ECDSA signatures but smaller than the tens of kilobytes produced by some hash-based signature schemes.

Zenroom implements ML-DSA for signing operations that must remain secure against quantum adversaries. The implementation supports both randomized and deterministic variants. Deterministic signing derives randomness from the message and secret key, eliminating dependence on random number generation quality at signing time.

ML-DSA verification is reasonably fast but signing is slow compared to ECDSA. A contract that signs many messages may hit computational limits. The bounded execution model prevents runaway computation but also means that bulk signing operations may require splitting across multiple contract executions.

7.2.3 NTRU

NTRU is a lattice-based cryptosystem providing both encryption and key exchange. NTRU has a longer history than newer NIST selections, having been proposed in 1996 and extensively analyzed. Some organizations prefer NTRU's longer track record over newer schemes.

Zenroom's NTRU implementation provides encryption rather than key encapsulation. A contract can encrypt data to a NTRU public key and decrypt with the corresponding private key. The encryption produces larger ciphertexts than symmetric encryption but avoids the key distribution problem.

NTRU was not selected by NIST in the most recent standardization round, which considered other lattice schemes superior. Zenroom includes NTRU primarily for compatibility with existing systems that deployed it before the NIST selections were finalized. New deployments should consider ML-KEM and ML-DSA instead unless specific requirements mandate NTRU.

7.3 Advanced Signature Schemes

7.3.1 BBS+ Zero-Knowledge Proofs

BBS+ is a signature scheme enabling selective disclosure. A signer creates a signature over multiple attributes. The signature holder can prove knowledge of a valid signature while revealing only a subset of attributes. The proof reveals nothing about unrevealed attributes and is unlinkable: multiple proofs cannot be correlated to the same signature.

Zenroom implements BBS+ for verifiable credential systems. A credential issuer signs a set of claims about a subject. The subject can later prove specific claims to a verifier without revealing other claims or enabling tracking. For example, a credential containing name, birthdate, address, and citizenship can prove citizenship without revealing the other attributes.

The BBS+ implementation follows the scheme published by Dan Boneh, Xavier Boyen, and Hovav Shacham with extensions for multi-message signing. Signing and verification use BLS12-381 pairings. A signature is a single group element regardless of how many attributes it covers.

Selective disclosure proofs are zero-knowledge proofs of knowledge of a signature on committed values.

BBS+ provides strong privacy guarantees but requires careful protocol design. The credential issuer must not embed tracking identifiers in the signature. The verifier must not request attribute combinations unique to a single individual. These operational security concerns are not addressed by the cryptography alone but require proper system design.

7.3.2 Schnorr Signatures

Schnorr signatures provide a simpler and more elegant signature scheme than ECDSA. The signatures have security proofs in the random oracle model and enable efficient multi-signatures where multiple signers jointly sign a message. Schnorr signatures were patented until 2008 which delayed adoption, but the patents expired and Schnorr is now widely implemented.

Zenroom supports Schnorr signatures on multiple curves including secp256k1 and Ed25519. The secp256k1 implementation is relevant for Bitcoin where Schnorr signatures were adopted in the Taproot upgrade. Ed25519 is fundamentally a Schnorr signature variant, and Zenroom's implementation supports both the standard EdDSA variant and variants for specific protocols.

Schnorr multi-signatures enable interesting applications. Multiple parties can jointly sign without requiring a single party to collect all signatures and aggregate them. The signing protocol is interactive but results in a signature indistinguishable from a single-party signature. This provides privacy and efficiency benefits for multi-party authorization.

7.3.3 Deterministic ECDSA

Standard ECDSA signature generation requires a random nonce for each signature. If the nonce is predictable or reused, the private key can be recovered from signatures. Many ECDSA vulnerabilities in production systems resulted from poor nonce generation. The PlayStation 3 jailbreak, numerous Bitcoin wallet compromises, and various other incidents all exploited nonce failures.

RFC 6979 specifies deterministic ECDSA where the nonce is derived from the message and private key using HMAC-DRBG. This eliminates dependence on random number generation at signing time. The signatures are identical to standard ECDSA and verify with the same algorithms. Only signature generation differs.

Zenroom implements deterministic ECDSA for all supported curves. Contracts do not choose between random and deterministic variants. Deterministic generation is always used. This eliminates an entire vulnerability class without requiring contract authors to understand nonce generation security.

Deterministic ECDSA has one subtle concern: side-channel attacks may be easier when signature generation is deterministic because adversaries can replay the same signature generation and observe timing differences. Zenroom mitigates this through constant-time implementations of sensitive operations but makes no claims about resistance to sophisticated side-channel adversaries with physical access.

7.4 Zero-Knowledge Proof Systems

Beyond BBS+ selective disclosure, Zenroom supports several zero-knowledge proof constructions for specific applications.

Schnorr proofs of knowledge: A prover can demonstrate knowledge of a discrete logarithm without revealing it. These proofs are used in authentication protocols and as building blocks for more complex proofs. Zenroom provides Schnorr proof primitives that contracts can combine into application-specific protocols.

Range proofs: A prover can demonstrate that a committed value lies within a specific range without revealing the exact value. Range proofs are used in confidential transactions where amounts must be proven non-negative without revealing transaction values. Zenroom's range proof implementation uses Bulletproofs for logarithmic proof size.

Set membership proofs: A prover can demonstrate that a committed value belongs to a specific set without revealing which element. These proofs enable credential systems where the holder proves possession of a credential from an authorized issuer without revealing which credential.

The zero-knowledge implementations are not general-purpose proving systems like SNARKs or STARKs. Those systems require trusted setups or enormous proof sizes unsuitable for the constraint environments Zenroom targets. The implemented proofs are specialized constructions with well-understood security properties and practical performance characteristics.

7.5 Credential and Identity Cryptography

Zenroom implements cryptographic primitives specifically for digital identity and verifiable credential systems. These are not generic primitives but purpose-built schemes for credential issuance, presentation, and verification.

Coconut credentials: A threshold credential scheme where credential issuance is distributed among multiple authorities. No single authority can track credential usage and threshold signing means credentials remain valid even if some authorities go offline. Coconut combines BLS signatures with zero-knowledge proofs to provide privacy-preserving credentials with selective disclosure.

W3C Verifiable Credentials: Cryptographic support for the W3C Verifiable Credentials specification including JSON-LD signatures with Ed25519 and ECDSA, JWT-

encoded credentials, and selective disclosure using BBS+. The implementation handles credential schema validation, proof verification, and revocation checking.

Decentralized Identifiers (DIDs): Cryptographic operations for did:dyne and compatibility with other DID methods. This includes key rotation, DID document updates, and proof of control over DIDs. The implementation integrates with the W3C DID specification and supports multiple proof formats.

SD-JWT: Selective disclosure for JSON Web Tokens using hash-based disclosure. The issuer creates a JWT with salted hashes of claims. The holder selectively reveals claims by providing the salt and original value. Verifiers check that revealed claims hash correctly. This provides simpler selective disclosure than BBS+ without requiring pairing-based cryptography.

These credential schemes are complex protocols, not simple signature operations. They require carefully designed data flows and state management. Zenroom's scenario system encapsulates this complexity behind readable Zencode statements. A contract can issue or verify a credential without implementing the underlying protocol details.

8 Scenarios: Production-Ready Implementations

Scenarios are Zencode modules that implement domain-specific functionality. Each scenario provides statements, schemas, and cryptographic operations for a particular application area. Scenarios are not example code or tutorials but production implementations that have been deployed at scale.

The scenario architecture allows extending Zencode without modifying the core VM or language. New cryptographic schemes, new data formats, new protocol integrations all happen through scenarios. This modularity has practical benefits: domain experts can implement scenarios while the VM maintainers focus on security and stability. An organization needing custom cryptography can develop a private scenario without forking Zenroom.

8.1 W3C Verifiable Credentials and DIDs

The W3C scenario implements the Verifiable Credentials and Decentralized Identifiers specifications. These are not toy implementations for demos but complete support for credential issuance, presentation, and verification following the official specifications.

Credential issuance:

A contract can create a verifiable credential containing claims about a subject. The issuer signs the credential using their private key. The signature algorithm can be Ed25519, ECDSA, or BBS+ depending on requirements. Ed25519 and ECDSA produce simple signatures. BBS+ enables selective disclosure where the holder later reveals only specific claims.

The credential structure follows W3C specification: context declarations, credential type, issuance date, expiration date, credential subject with claims, and cryptographic proof. Zencode statements handle schema validation, ensuring credentials match their declared type and contain required fields.

Credential presentation:

A holder presents credentials to verifiers. For credentials signed with Ed25519 or ECDSA, presentation means providing the complete credential. The verifier checks the signature and validates against issuer's public key. For BBS+ credentials, presentation can be selective. The holder creates a zero-knowledge proof revealing chosen claims while hiding others. The proof is unlinkable: multiple presentations cannot be correlated.

Zencode contracts specify which claims to reveal. The VM generates the cryptographic proof automatically. The contract author does not implement zero-knowledge proof protocols. They write statements like "When I create selective disclosure of 'credential' revealing 'claim1' and 'claim2'". The underlying complexity is encapsulated in the scenario implementation.

DID operations:

DIDs are decentralized identifiers that do not depend on centralized registries. A DID resolves to a DID document containing public keys, service endpoints, and authentication methods. The W3C scenario supports creating DIDs, generating DID documents, and proving control over DIDs through cryptographic challenges.

The did:dyne method is implemented natively. This method uses deterministic generation where the DID identifier is derived from the public key. No blockchain or distributed ledger is required. The DID document is cryptographically linked to the identifier. Anyone can verify the DID document matches the identifier without external lookups.

Support for other DID methods varies. Methods requiring blockchain interaction or external resolver services cannot be fully implemented within Zenroom because the VM has no network access. The scenario provides the cryptographic operations (key management, signing, verification) but integration with external systems requires the calling application to handle network communication.

Revocation:

Credential revocation is supported through status lists. An issuer publishes a revocation list as a credential. Each bit in a bitstring represents a credential's revocation status. Verifiers check the status list before accepting a credential. This approach is specified in the W3C Status List 2021 specification.

Zenroom can verify credentials against status lists but cannot publish status lists to external systems. The calling application must handle distribution. This is a consequence of isolation: the VM cannot write to filesystems or make network requests. The division of responsibility is explicit in the architecture.

8.2 Blockchain Integration

Blockchain scenarios provide cryptographic operations for transaction signing, address derivation, and message verification. These are not full blockchain node implementations. Zenroom does not maintain blockchain state or validate consensus. It provides the cryptographic primitives applications need to interact with blockchains.

8.2.1 Bitcoin and UTXO Model

The Bitcoin scenario implements operations for transaction creation and signing. Bitcoin uses the UTXO (Unspent Transaction Output) model where transactions consume previous outputs and create new outputs. Transaction signing follows Bitcoin's complex rules for script evaluation and signature encoding.

Address derivation:

Zenroom generates Bitcoin addresses from public keys. Multiple address formats are supported: legacy P2PKH, P2SH, and native SegWit (bech32). The implementation handles encoding conversions and checksum validation for each format. Addresses are derived deterministically from keys, allowing key management systems to regenerate addresses without storing them.

Hierarchical Deterministic (HD) wallet support follows BIP32 and BIP44 specifications. A contract can derive child keys from a master key using derivation paths. This enables wallet implementations where a single seed generates many addresses. The derivation is deterministic and follows industry-standard paths for Bitcoin mainnet and testnet.

Transaction signing:

Bitcoin transaction signing is complex because transactions can have multiple inputs with different signing requirements. Each input references a previous output with a script defining spending conditions. Common scripts require ECDSA signatures over specific transaction data.

Zenroom computes transaction hashes according to Bitcoin's serialization rules including handling for SegWit transactions. The contract provides transaction data and keys. The scenario computes what data to sign for each input, generates signatures, and formats them according to Bitcoin script requirements.

The implementation supports standard transaction types but not arbitrary scripts. Complex scripts involving timelocks, multisignature schemes, or custom opcodes may require application-specific handling. The scenario provides building blocks that applications compose into full transaction construction.

Message signing:

Bitcoin message signing uses a format distinct from transaction signing. A message is prefixed with a magic string, hashed, and signed. The signature includes a recovery byte allowing public key recovery from signature alone. This format is used for proving address ownership without creating transactions.

8.2.2 Ethereum and EVM Compatibility

The Ethereum scenario implements operations for Ethereum transaction signing and smart contract interaction. Ethereum's account-based model differs from Bitcoin's UTXO model. Transactions transfer value between accounts or invoke contract code.

Transaction signing:

Ethereum transactions include nonce, gas parameters, recipient address, value, and optional data payload. The transaction is serialized using RLP (Recursive Length Prefix) encoding, hashed with Keccak-256, and signed with ECDSA on secp256k1. The signature includes a recovery byte and chain ID to prevent replay attacks across different Ethereum networks.

Zenroom computes transaction hashes and produces signatures. It does not maintain account state or calculate gas costs. The calling application must query blockchain state to determine appropriate nonce and gas values. This separation is deliberate: maintaining state would require network access and persistent storage that violate isolation guarantees.

EIP-712 typed data signing:

Ethereum applications use EIP-712 for structured data signing. Rather than signing raw bytes, applications sign typed data structures with domain separation. This prevents signature reuse across different applications and makes signed data human-readable in wallet interfaces.

The scenario implements EIP-712 encoding: domain separator construction, type hashing, and structured data encoding. A contract specifies data types and values. The scenario generates the correct hash for signing. This supports decentralized exchange orders, token permits, and other off-chain signed messages.

Smart contract verification:

Zenroom can verify data produced by Ethereum smart contracts when that data is provided as input. For example, a contract might verify Merkle proofs from Ethereum storage or validate signatures produced by on-chain operations. What Zenroom cannot do is query Ethereum nodes or verify blockchain state directly. It processes provided data but does not fetch data from external sources.

8.2.3 Smart Contract Verification

Beyond transaction operations, blockchain scenarios provide tools for verifying cryptographic proofs associated with smart contracts. These are patterns that appear across multiple blockchain platforms.

Merkle proofs:

Smart contracts often commit to large datasets using Merkle trees and provide proofs that specific data belongs to the committed set. Zenroom verifies Merkle inclusion proofs: given a Merkle root, a data element, and a proof path, verify that the element is included in the tree.

This enables applications where blockchain contracts commit to off-chain data. A contract publishes a Merkle root on-chain. Users download data off-chain and verify inclusion proofs. The verification happens in Zenroom without querying the blockchain. The application provides the Merkle root, data, and proof as inputs.

Signature aggregation:

Some blockchain protocols use signature aggregation to compress proofs. Multiple signatures are combined into a single aggregate signature. BLS signatures support this efficiently. Zenroom verifies aggregate signatures given the aggregate, the messages, and public keys.

This pattern appears in consensus protocols where validators sign blocks and their signatures are aggregated. Light clients verify aggregates without checking individual signatures. Zenroom provides the cryptographic verification without implementing full consensus protocol logic.

8.3 JWT/JWS/SD-JWT Token Management

The JOSE scenario implements JSON Object Signing and Encryption standards used widely for authentication tokens, API authorization, and federated identity. JWT (JSON Web Tokens) are ubiquitous in web applications. Zenroom provides production-grade implementation of JWT creation and verification.

JWT creation:

A contract creates a JWT by specifying claims and selecting a signature algorithm. Supported algorithms include EdDSA, ECDSA with multiple curves, and RSA. The scenario handles JSON serialization, Base64URL encoding, and signature generation. The output is a standard JWT that any compliant implementation can verify.

JWTs typically encode user identity and permissions for API access. An authentication service creates tokens after login. APIs verify tokens on each request. Zenroom can serve either role: creating tokens in authentication flows or verifying tokens in API gateways.

JWS verification:

JSON Web Signatures extend JWT with additional signature options. A JWS can have detached payloads or multiple signatures. Zenroom verifies JWS structures: parse header, decode payload, verify signature matches. The implementation validates algorithm parameters and rejects weak algorithms or malformed structures.

The scenario protects against common JWT vulnerabilities. It does not accept ”none” algorithm. It validates algorithm matches expected key type. It checks token expiration and not-before claims. These checks prevent attacks that have compromised production systems using JWT libraries with poor defaults.

SD-JWT selective disclosure:

Selective Disclosure JWT is a specification enabling holder-initiated selective disclosure without zero-knowledge proofs. The issuer creates a JWT where some claims are salted hashes. The holder receives claim values and salts. To present the token, the holder includes chosen claims and salts. The verifier checks hashes match disclosed values.

SD-JWT provides simpler selective disclosure than BBS+ because it uses only hashing, not pairings. The tradeoff is that disclosure reveals unique identifiers: multiple presentations can be linked if the issuer included tracking data. For use cases where linkability is acceptable, SD-JWT offers easier implementation.

Zenroom implements both issuing and verifying SD-JWT. A contract creating an SD-JWT specifies which claims to make selectively disclosable. The scenario generates hashes and packages salt values for the holder. A verifying contract receives the SD-JWT and disclosed claims and validates hashes.

8.4 Coconut Credentials and Selective Disclosure

Coconut is a threshold credential scheme developed specifically for the DECODE project. Unlike W3C credentials which are general-purpose, Coconut optimizes for privacy-preserving selective disclosure with threshold issuance. The scenario implements the complete Coconut protocol.

Threshold issuance:

Credentials are issued by multiple authorities. No single authority can issue a credential alone. This distributes trust and prevents single-party tracking. A subject requests credential issuance from n authorities. Each authority independently verifies the request and produces a partial credential. The subject combines t partial credentials into a full credential, where t is the threshold.

The threshold property means credentials remain valid if authorities go offline or become uncooperative. As long as t authorities remain honest and available, the system functions. This resilience matters for production systems where availability is critical.

Zenroom implements the authority side (verifying requests and producing partial credentials) and the subject side (combining partial credentials). The communication between subject and authorities happens outside Zen-

room. The scenario provides cryptographic operations while the application handles network protocols.

Selective disclosure:

Coconut credentials contain attributes as committed values. The holder proves knowledge of a valid credential and reveals chosen attributes. The proof is zero-knowledge: unrevealed attributes remain hidden. The proof is unlinkable: multiple presentations cannot be correlated even if the verifier colludes with issuers.

A Zencode contract specifies which attributes to reveal. The scenario generates the zero-knowledge proof automatically. The implementation uses Schnorr proofs and BLS signatures on BLS12-381. The cryptographic complexity is hidden behind statements like "When I create credential presentation revealing 'attribute1'".

Verification:

Verifiers receive presentations containing revealed attributes and zero-knowledge proofs. Verification checks: the proof is valid, the credential was issued by authorized authorities (verified through public keys), and the revealed attributes are correctly opened. Verification does not reveal unrevealed attributes or enable tracking.

The scenario handles verification mechanics. A contract declares which authorities are trusted (their public keys) and which attributes were revealed. The scenario validates the presentation. The contract receives a boolean result: verification succeeded or failed.

8.5 PVSS and Threshold Cryptography

PVSS (Publicly Verifiable Secret Sharing) enables distributing secrets among multiple parties with public verifiability. This is foundation for threshold cryptography where operations require cooperation among multiple parties. The scenario implements Feldman VSS and PVSS constructions.

Secret sharing:

A dealer distributes a secret among n participants such that any t participants can reconstruct the secret but fewer than t learn nothing. The shares are generated using Shamir secret sharing with polynomial evaluation. Each participant receives one share.

The publicly verifiable property means anyone can verify shares are correctly generated without learning the secret. Verification uses commitments to polynomial coefficients. Participants check their share matches the commitments. This prevents a malicious dealer from distributing invalid shares.

Zenroom implements the dealer role (generating shares and commitments) and participant role (verifying shares and reconstructing secrets). The scenario supports using PVSS for key generation where no single party knows the private key but threshold subsets can jointly sign.

Threshold signatures:

After PVSS establishes shared secrets, participants can jointly produce signatures without reconstructing the private key. Each participant produces a signature share using their share of the signing key. Combining threshold shares produces a valid signature.

This pattern enables distributed signing where no single party can sign alone but legitimate combinations of parties can. The scenario implements threshold signing for BLS signatures. The signature is identical to a single-party signature, maintaining privacy about the threshold nature of signing.

Distributed key generation:

PVSS enables generating keys where no party knows the full private key. Each party runs PVSS as dealer distributing random shares. The final key is the sum of contributions. The public key is computable from commitments. No party learns the private key.

This construction is used in distributed custody systems where key compromise requires threshold collusion. Zenroom provides the cryptographic operations. Applications must handle network communication among participants during the interactive protocol.

9 Security Audit and Verification

Security verification of cryptographic software requires more than code review. Static analysis catches certain bug classes but misses others. Formal verification proves properties about models that may not match implementation. Real security confidence comes from multiple verification approaches applied systematically. Zenroom uses testing, fuzzing, determinism validation, and adversarial scenarios to build confidence in implementation correctness.

The testing approach is pragmatic rather than theoretical. We do not claim formal proof of security. Such proofs would apply to mathematical models, not the C and Lua implementation running on actual hardware. Instead we test extensively, document limitations honestly, and provide mechanisms for organizations to verify behavior in their specific deployment contexts.

9.1 Testing Methodology

Zenroom's test suite spans multiple categories reflecting different verification goals. Unit tests verify individual functions. Integration tests verify scenario implementations. Determinism tests verify identical behavior across platforms. Vector tests verify cryptographic correctness against published test vectors. Each category serves distinct purposes and catches different bug classes.

Unit tests:

Core VM functionality is tested at the function level. Memory allocation, string handling, encoding conversions, and data structure operations all have dedicated tests. These tests run quickly and provide immediate feedback during

development. When a change breaks basic functionality, unit tests catch it before deeper integration issues emerge.

The unit tests are written in Lua using Zenroom's embedded Lua environment. This tests the VM in its actual deployment configuration, not through special testing interfaces. If the tests pass, the same code paths work in production contracts.

Integration tests:

Scenario implementations are tested through complete contract execution. A test creates input data, executes a contract using that data, and verifies output matches expected results. These tests verify that scenarios correctly implement their specifications.

Integration tests use the BATS (Bash Automated Testing System) framework. Each test is a shell script that invokes Zenroom with specific inputs and checks outputs. This testing approach is simple and portable. The same tests run on development machines, continuous integration servers, and production platforms.

Cryptographic vector tests:

Standard cryptographic test vectors verify implementation correctness. NIST provides test vectors for algorithms like SHA-256, AES, and elliptic curve operations. The IETF provides vectors for protocols like ECDH and EdDSA. Test vectors from academic papers verify newer schemes like BBS+ and Coconut credentials.

Vector tests are critical because cryptographic implementation errors often produce outputs that appear correct but are cryptographically weak. A signature algorithm with an implementation error might produce signatures that verify with the same implementation but not with others. Vector tests catch these interoperability failures by comparing against independent implementations.

Determinism validation:

Deterministic execution is verified by running the same contract multiple times and comparing outputs byte-for-byte. Tests run contracts with fixed random seeds to ensure random number generation is deterministic. Tests run on different platforms (Linux, macOS, Windows) to verify cross-platform determinism. Tests run with different memory configurations to verify garbage collection does not introduce non-determinism.

Determinism tests have found bugs that other testing would miss. Early versions had a bug where dictionary iteration order varied based on hash function behavior that differed between architectures. Unit tests did not catch this because they ran on a single platform. Determinism tests comparing x86 and ARM outputs found the issue immediately.

Performance regression tests:

Cryptographic operations have expected performance characteristics. Signature generation and verification should complete in milliseconds. Pairing operations take longer but still bound to reasonable time. Performance tests detect implementation changes that accidentally introduce computational overhead.

These tests are not about optimizing speed but about detecting regressions. A change that makes signature verification 10x slower probably introduced a bug, not just inefficiency. Performance tests catch these issues during development rather than after deployment.

9.2 Fuzzing and Adversarial Testing

Fuzzing feeds malformed or unexpected inputs to software and monitors for crashes or undefined behavior. For cryptographic software, fuzzing tests parser robustness, input validation, and error handling. Zenroom uses multiple fuzzing approaches targeting different components.

Parser fuzzing:

The Zencode parser is fuzzed with randomly mutated contracts. Valid contracts are generated and then mutated: keywords are changed, quotes are removed, statements are duplicated or reordered. The fuzzer monitors whether the parser gracefully rejects invalid input or crashes. Parser crashes indicate bugs that could be exploited through malicious contracts.

Parser fuzzing has found issues in statement pattern matching and error message generation. These bugs did not affect correct contracts but could be triggered by malformed input. Fixing them improves robustness for production deployment where input may be hostile.

Input data fuzzing:

Contracts expect input data in specific formats. The fuzzer generates data that violates these expectations: wrong types, missing fields, excessively large values, negative numbers where positive expected. Schema validation should reject invalid data before contracts process it. Fuzzing verifies this rejection is robust.

Input fuzzing revealed cases where schema validation was incomplete. Some encoding validators did not check for overlong encodings. Some array validators did not enforce maximum length. These issues were fixed before affecting production deployments.

Cryptographic primitive fuzzing:

Lower-level cryptographic functions are fuzzed with random inputs. Point addition on elliptic curves is fuzzed with invalid points. Signature verification is fuzzed with malformed signatures. Hash functions are fuzzed with extreme input sizes. These tests verify that cryptographic code handles edge cases correctly.

Cryptographic fuzzing uses specialized tools like libFuzzer that instrument code to track code coverage and guide fuzzing toward unexplored paths. This approach finds edge cases that random fuzzing would miss.

Adversarial contract testing:

Beyond random fuzzing, specific adversarial scenarios test known attack patterns. Contracts that attempt to exhaust memory through excessive allocation. Contracts that attempt to bypass phase restrictions. Contracts that attempt timing attacks by measuring execution duration of cryptographic operations. These scenarios verify that security mechanisms actually work under attack.

Adversarial testing revealed the importance of computational bounds. Early versions allowed contracts to create arbitrarily large arrays. An adversary could craft a contract that appeared simple but generated gigabytes of output. The current iteration and memory limits prevent this class of attack.

9.3 Memory Safety Validation

Memory safety vulnerabilities like buffer overflows and use-after-free bugs are common in C code. Zenroom is implemented in C for performance and portability but C provides no automatic memory safety. Verification approaches detect memory errors during testing before they appear in production.

Address Sanitizer:

AddressSanitizer (ASan) is a compiler instrumentation tool that detects memory errors at runtime. Zenroom builds with ASan enabled run tests that would crash or behave unpredictably if memory errors exist. ASan catches buffer overflows, use-after-free, double-free, and similar errors.

All tests run under ASan during continuous integration. This catches memory errors in new code before merges. The performance overhead of ASan makes it unsuitable for production but acceptable for testing.

Valgrind:

Valgrind is a runtime analysis tool that detects memory leaks, invalid reads and writes, and use of uninitialized memory. Valgrind is slower than ASan but catches additional error classes. Zenroom tests run under Valgrind periodically to verify memory management correctness.

Valgrind testing found memory leaks in error handling paths. Normal contract execution freed all allocations but error paths skipped cleanup. These leaks did not affect short-lived executions but would accumulate in long-running processes. The leaks are now fixed.

Static analysis:

Static analysis tools like Clang Static Analyzer examine code without executing it, finding potential bugs through code path analysis. Static analysis complements runtime testing by finding issues in code paths that tests might not exercise.

Static analysis found potential null pointer dereferences in error handling and missing checks for allocation failure. Not all static analysis warnings indicate real bugs but investigating them improves code quality and robustness.

Manual code review:

Automated tools do not catch all errors. Manual code review by multiple developers verifies that code follows security guidelines, handles errors properly, and implements algorithms correctly. Cryptographic implementations receive extra scrutiny because implementation errors can completely break security.

Code review focuses on security-critical components: cryptographic primitives, memory management, parser validation, and VM isolation boundaries. Changes to these components require review by maintainers familiar with their security requirements.

9.4 Known Limitations and Mitigations

No software is perfectly secure. Claiming otherwise would be dishonest. Zenroom has known limitations that organizations should understand when evaluating deployment. These limitations are documented rather than hidden and mitigations are provided where possible.

Side-channel resistance:

Zenroom's cryptographic implementations use constant-time algorithms where feasible but complete side-channel resistance requires hardware support that general-purpose systems lack. Timing attacks that measure execution duration may leak information about cryptographic keys. Cache-based attacks may leak information through memory access patterns.

Organizations concerned about side-channel attacks should deploy Zenroom on hardware with appropriate protections. Use HSMs for key operations that require highest security. Deploy in environments where adversaries cannot measure timing or observe cache behavior. These operational mitigations are more effective than software-only approaches.

Quantum computing:

Current elliptic curve cryptography will be vulnerable when large-scale quantum computers exist. Zenroom provides post-quantum algorithms (ML-KEM, ML-DSA) to prepare for this transition but these are newer and less tested than traditional cryptography. Organizations should assess their timeline for quantum risk and adopt post-quantum algorithms when risk models justify the change.

Hybrid approaches combine traditional and post-quantum cryptography. A key exchange using both ECDH and ML-KEM is secure if either remains unbroken. This hedges against both premature quantum computers and undiscovered weaknesses in lattice assumptions.

Implementation bugs:

Despite testing, bugs remain in any complex software. Zenroom's approach is to minimize trusted code, isolate components, and provide rapid update mechanisms. The zero-dependency architecture means updates do not de-

pend on external maintainers. Organizations can apply patches quickly when vulnerabilities are discovered.

The small codebase aids security. The entire VM and cryptographic library compile to under 3MB. This is orders of magnitude smaller than typical application stacks. Smaller code is easier to audit and contains fewer places for bugs to hide.

Cryptographic assumptions:

All cryptography depends on mathematical assumptions about hard problems. Elliptic curve cryptography assumes discrete logarithms are hard. Pairings assume specific problems in extension fields are hard. Post-quantum schemes assume lattice problems are hard. If these assumptions are wrong, the cryptography fails.

Zenroom cannot prevent cryptographic assumptions from being invalidated. What it provides is cryptographic agility: the ability to replace algorithms when necessary. When a weakness is found in a cryptographic scheme, organizations can update to new algorithms without replacing their entire infrastructure.

9.5 Incident Response and Updates

Security vulnerabilities will be discovered. The question is not if but when and how the response is handled. Zenroom's incident response prioritizes rapid remediation, clear communication, and verifiable fixes.

Vulnerability reporting:

Security vulnerabilities should be reported privately to allow fixing before public disclosure. Zenroom maintains a security contact and responds to reports within 48 hours. Reporters receive acknowledgment and updates on remediation progress. Responsible disclosure periods allow time for fixes and deployment before public announcement.

Public disclosure happens after fixes are available. Security advisories describe the vulnerability, affected versions, and remediation steps. Advisories include enough detail for organizations to assess impact without providing exploitation instructions for attackers.

Update distribution:

Security updates are distributed through the same channels as regular releases: GitHub releases, package managers, and direct download from the Zenroom website. Organizations should subscribe to security announcements to receive notification of critical updates.

The zero-dependency architecture simplifies updates. Organizations deploy a single binary without worrying about library version compatibility. Testing an update verifies the new binary works correctly. Rollback means deploying the old binary. This simplicity enables rapid response when vulnerabilities require immediate patches.

Version verification:

Each Zenroom release is cryptographically signed. Organizations can verify that binaries are authentic and unmodified. The signing key is published through multiple channels. Verification prevents attackers from distributing backdoored versions.

Deterministic builds allow additional verification. Organizations can compile Zenroom from source and verify the resulting binary matches the official release. Byte-for-byte reproducibility means successful compilation proves the binary corresponds to the public source code.

Lessons learned:

After incidents, post-mortems analyze what went wrong and how to prevent similar issues. These analyses improve testing, strengthen code review, and enhance security practices. The goal is not to avoid blame but to systematically improve security through learning from mistakes.

10 Deployment and Integration

Zenroom deployment spans embedded devices with kilobytes of RAM to cloud servers with gigabytes. This range is possible because the VM imposes minimal requirements: a C compiler, basic POSIX functions, and memory allocation. No threading, no dynamic linking, no filesystem access, no network stack. Platforms meeting these minimal requirements can run Zenroom.

The portability is not theoretical. Production deployments run on iOS phones, Android devices, Raspberry Pi boards, ESP32 microcontrollers, Linux servers, Windows desktops, and WebAssembly in browsers. The same VM source compiles for all these targets with platform-specific compilation flags but no code changes.

10.1 Platform Portability

10.1.1 Mobile: iOS and Android

Mobile deployments face constraints that desktop deployments ignore. Limited memory requires careful allocation. Background execution restrictions limit long-running operations. Operating system updates can break binary compatibility. Zenroom's design addresses these constraints.

iOS:

iOS applications link Zenroom as a static library. The library is compiled with Xcode targeting specific iOS versions. The compilation produces universal binaries supporting multiple ARM architectures: arm64 for recent devices, armv7 for older devices, simulator builds for development.

iOS memory constraints are handled through fixed memory pool configuration. Applications set maximum memory at initialization based on available resources. The VM operates within this limit or fails cleanly. iOS does not allow applications to handle out-of-memory conditions

gracefully. Setting appropriate limits prevents system kills.

iOS security features like Address Space Layout Randomization (ASLR) and code signing work transparently with Zenroom. The static library is included in the application bundle and signed with the application. No special handling is required.

Android:

Android applications can use Zenroom through JNI (Java Native Interface) or NDK (Native Development Kit) directly. The JNI approach provides Java classes wrapping Zenroom functions. Applications call Java methods which invoke native code. The NDK approach links Zenroom as a native library loaded at runtime.

Android builds target multiple ABIs: armeabi-v7a, arm64-v8a, x86, x86_64. The build system produces separate binaries for each ABI. Android package managers select the appropriate binary for the device architecture. This multi-ABI support ensures compatibility across the Android ecosystem.

Android memory management differs from iOS. The system can request applications release memory when pressure is high. Zenroom's fixed pool approach means memory usage is predictable. Applications can communicate available memory to the system accurately.

10.1.2 Server: Linux, Windows, macOS

Server deployments typically have abundant resources compared to mobile but face different challenges: multiple simultaneous executions, long-running processes, diverse deployment environments.

Linux:

Linux is the primary development platform and most common deployment target. Zenroom builds with gcc or clang using standard POSIX APIs. The resulting binary has no dependencies beyond libc. Static linking is supported for deployments where even libc compatibility is a concern.

Linux security features integrate well with Zenroom. Seccomp profiles restrict system calls to the minimal set Zenroom requires. SELinux policies can confine Zenroom processes. Namespaces and cgroups provide additional isolation in containerized deployments.

The Linux build supports both x86_64 and various ARM architectures. Cloud providers increasingly offer ARM instances with better performance-per-watt than x86. Zenroom runs identically on both architectures due to careful attention to platform-specific behavior.

Windows:

Windows builds use MinGW or Visual Studio. The MinGW approach produces binaries compatible with standard Windows deployment. The Visual Studio approach integrates better with Windows development toolchains but requires additional build dependencies.

Windows system call interfaces differ significantly from POSIX. Zenroom abstracts these differences through platform-specific compilation units. The same VM logic runs but OS interaction uses Windows APIs rather than POSIX.

Windows deployment often requires digitally signed binaries for enterprise distribution. Zenroom releases include signed binaries. Organizations can verify signatures before deployment and some Windows security policies require signature verification.

macOS:

macOS builds are similar to Linux but with Apple-specific considerations. The system requires code signing for notarization. Applications downloaded from outside the App Store must be notarized or users receive security warnings.

macOS security features like System Integrity Protection and Gatekeeper work with Zenroom if binaries are properly signed. Unsigned development builds work but trigger security prompts. Production deployments should use signed binaries.

Apple Silicon (ARM) and Intel (x86) both run Zenroom. Universal binaries containing both architectures simplify distribution. The same binary runs on either architecture with Rosetta translation if needed, though native execution is preferred.

10.1.3 Embedded: ARM Cortex, ESP32

Embedded deployments operate under extreme resource constraints. Devices may have 256KB of RAM and 1MB of flash storage. These constraints require careful configuration but Zenroom can operate successfully.

ARM Cortex-M:

Cortex-M microcontrollers are common in IoT devices. These are bare-metal environments without operating systems. Zenroom builds for Cortex-M use specialized toolchains like arm-none-eabi-gcc. The build produces firmware images flashed to device storage.

Cortex-M deployment requires minimal libc support. Newlib or similar minimal C libraries provide required functions. Memory allocation uses static pools rather than dynamic allocation. The fixed memory model fits embedded constraints naturally.

Cryptographic operations on Cortex-M are slower than on application processors but remain practical. A signature verification takes milliseconds rather than microseconds. For many IoT applications this is acceptable. Device authentication happens infrequently and users tolerate brief delays.

ESP32:

ESP32 microcontrollers from Espressif combine ARM cores with WiFi and Bluetooth. These are common in IoT projects. Zenroom builds for ESP32 use the ESP-IDF

framework and Xtensa or RISC-V toolchains depending on chip variant.

ESP32 has more resources than Cortex-M: typically 512KB RAM and 4MB flash. This allows more complex contracts and larger data processing. ESP32 can perform cryptographic operations needed for device authentication and secure communication.

ESP32 deployment often uses RTOS (Real-Time Operating System) like FreeRTOS. Zenroom runs as a task under the RTOS. Multiple contracts can execute concurrently in different tasks though each executes deterministically. The RTOS handles task scheduling while Zenroom provides cryptographic operations.

10.1.4 Browser: WebAssembly

WebAssembly enables Zenroom execution in web browsers without plugins. The JavaScript API provides cryptographic operations directly in the browser. This enables applications where sensitive data never leaves the client device.

Compilation:

WebAssembly builds use Emscripten to compile C source to WASM bytecode. The resulting module loads in any modern browser. The module size is approximately 500KB compressed, acceptable for web deployment. Loading is asynchronous to avoid blocking browser rendering.

The WASM module exposes JavaScript functions wrapping Zenroom API calls. JavaScript code invokes these functions to execute contracts. Input and output are JavaScript objects converted to JSON for Zenroom processing. This provides natural integration with web application code.

Limitations:

WebAssembly sandboxing prevents system calls that Zenroom does not use anyway. File access, network access, and direct memory access are all prohibited. These restrictions match Zenroom's isolation requirements. The browser sandbox provides additional security layer.

WASM memory management differs from native platforms. The heap is a linear memory block grown by the module. Zenroom's fixed pool allocation maps to WASM linear memory. The JavaScript environment handles memory allocation for the WASM module.

WebAssembly performance is typically 50-70% of native code. Cryptographic operations in WASM are slower than native but remain practical. Browser-based credential presentation taking 100ms rather than 50ms is acceptable. Users perceive both as instantaneous.

10.2 Language Bindings

Language bindings allow applications written in high-level languages to invoke Zenroom without writing C code. Each binding provides idiomatic interfaces for its language while wrapping the same underlying VM.

10.2.1 JavaScript/Node.js

JavaScript bindings support both browser (through WASM) and Node.js (through native modules). The API is consistent across environments. Applications can target both with the same code.

The bindings use promises for asynchronous execution. Contract execution returns a promise that resolves with output or rejects with errors. This matches JavaScript conventions for asynchronous operations. Error handling uses try-catch blocks with structured error objects.

Node.js bindings use N-API for native module integration. N-API provides ABI stability across Node.js versions. Modules compiled for one Node.js version work with subsequent versions. This simplifies distribution and maintenance.

10.2.2 Python

Python bindings use ctypes to load Zenroom as a shared library. Functions are declared with appropriate signatures and called from Python code. The bindings handle data conversion between Python objects and C types.

The API is synchronous following Python conventions. Asynchronous execution can be implemented at application level using threads or async frameworks. The binding focuses on correct data marshalling and error handling rather than imposing execution model.

Python bindings support both CPython and PyPy. Memory management integrates with Python's garbage collector. Buffers allocated by Zenroom are freed when Python objects are collected. This prevents memory leaks from improper cleanup.

10.2.3 Rust

Rust bindings use FFI (Foreign Function Interface) to call Zenroom C functions. The bindings provide safe Rust interfaces wrapping unsafe FFI calls. Rust's ownership system ensures memory safety at the binding boundary.

The bindings handle conversion between Rust types and C types. Strings are converted to null-terminated C strings. Buffers are passed as raw pointers with explicit lengths. The bindings manage allocation and deallocation to prevent leaks.

Error handling uses Rust's Result type. Functions return Result containing either success value or error information. This integrates naturally with Rust error handling patterns. The question mark operator propagates errors through call chains.

10.2.4 Golang

Go bindings use cgo to call C functions. The bindings declare C function signatures in Go source files. The Go compiler generates glue code for type conversion and calling conventions.

Memory management requires care in cgo. Go's garbage collector does not track C allocations. The bindings explicitly free C memory when Go objects are finalized. Finalizers are registered on Go objects holding C resources.

The bindings provide synchronous and concurrent APIs. Synchronous calls block until execution completes. Concurrent calls use goroutines and channels for asynchronous execution. This matches Go concurrency patterns while maintaining Zenroom's deterministic execution per contract.

10.3 API Design and Best Practices

The Zenroom API is designed for simplicity and safety. The core API has three functions: execute a contract given data and keys, return output or errors, clean up resources. This minimal surface area reduces integration complexity.

Input handling:

Inputs are provided as byte buffers containing JSON, CBOR, or MessagePack. The calling application serializes data to these formats. Zenroom parses them during contract execution. This separation means the API does not need to understand application data structures.

Binary data is base64-encoded within JSON. This avoids encoding issues when passing binary through text-based formats. Applications can pass cryptographic keys and signatures without specialized handling. The base64 encoding overhead is acceptable for the sizes Zenroom processes.

Output handling:

Output is returned as a byte buffer containing JSON. The calling application parses this buffer to extract results. Structured errors are returned in the same format. An execution error produces JSON describing what failed and where.

The JSON output is deterministically ordered. Object keys are sorted alphabetically. This makes output suitable for hashing or signature generation. Two executions producing identical results produce byte-identical JSON.

Resource management:

Memory allocated by Zenroom must be freed by the calling application. The API provides explicit free functions. Failing to free memory causes leaks. Language bindings handle this automatically through finalizers or destructors.

Some platforms have automatic cleanup. Mobile applications terminating release all memory. Server applications running long-term must manage cleanup explicitly. The binding documentation specifies cleanup requirements for each platform.

Thread safety:

Each Zenroom execution is independent. Multiple threads can execute different contracts concurrently. The VM has no shared global state. This parallelism scales to available CPU cores without locks or contention.

The same contract executed simultaneously in different threads produces identical outputs. Thread execution order does not affect determinism. Applications can parallelize contract execution without worrying about race conditions.

10.4 Performance Characteristics

Performance depends on contract complexity, input size, and platform capabilities. General patterns exist that help organizations plan deployments.

Contract parsing:

Parsing a contract takes microseconds to low milliseconds depending on contract length. Parsing is one-time cost at contract load. Some applications compile contracts once and execute many times. Others parse fresh for each execution accepting the overhead for operational simplicity.

Cryptographic operations:

Operation performance varies by algorithm and platform:

- ECDSA signature generation: 1-5ms on modern CPUs, 10-50ms on embedded
- ECDSA signature verification: 2-10ms on modern CPUs, 20-100ms on embedded
- Ed25519 signature generation: 0.1-0.5ms on modern CPUs, 1-5ms on embedded
- Ed25519 signature verification: 0.2-1ms on modern CPUs, 2-10ms on embedded
- BLS signature verification: 5-20ms on modern CPUs, pairing operations are expensive
- ML-KEM key generation: 10-50ms depending on security parameter
- ML-DSA signature generation: 50-200ms, post-quantum operations are slower

These ranges reflect typical performance. Actual timing depends on specific platforms, compiler optimizations, and thermal conditions on mobile devices.

Memory usage:

Memory usage grows with contract complexity and data size. A simple contract processing 1KB of data uses approximately 1MB total including VM overhead. Complex contracts with large arrays or many cryptographic operations may use 10MB or more. The fixed pool allocation makes memory usage predictable and configurable.

Scalability:

Single-threaded performance is sufficient for many applications. A server processing 1000 transactions per second with 5ms average execution time needs only 5 concurrent executions. Thread-level parallelism handles this easily. Applications with higher throughput requirements can scale horizontally, running multiple Zenroom processes across multiple servers.

11 Real-World Applications and Case Studies

Production deployments provide evidence that designs work under real conditions. Pilot projects demonstrate concepts. Production systems prove viability at scale with actual users, real data, and operational requirements. Zenroom deployments span government digital identity, financial infrastructure, and distributed systems. These are not demonstrations but operational systems.

The case studies that follow are not marketing claims but documented implementations. Each has source code, deployment documentation, and operational history. Organizations evaluating Zenroom can examine these systems and contact operators for references.

11.1 EUDI-ARF Wallet Certification (CREDIMI)

The European Digital Identity framework (EUDI) establishes requirements for digital identity wallets across EU member states. The Architecture Reference Framework (ARF) specifies technical and security requirements. CREDIMI provides certification infrastructure ensuring wallets meet these requirements.

Technical architecture:

CREDIMI uses Zenroom for cryptographic operations in the certification process. Wallet implementations present credentials for verification. The certification system verifies credentials against schemas, validates cryptographic signatures, and checks compliance with ARF specifications.

The system processes W3C Verifiable Credentials using multiple signature schemes. Legacy credentials use ECDSA or EdDSA signatures. Modern credentials use BBS+ for selective disclosure. The certification process verifies that wallets correctly implement each scheme and properly handle edge cases.

Zenroom's deterministic execution is critical here. Certification results must be reproducible. An auditor reviewing a certification decision must obtain identical results when re-executing verification contracts. This reproducibility is required by the ARF but difficult to achieve with non-deterministic systems.

Operational requirements:

Certification processes credentials from multiple member states with different implementations. Interoperability testing verifies that credentials issued by one state's system can be verified by another's. Zenroom's strict schema validation catches incompatibilities that would cause production failures.

The system operates under high assurance requirements. Code undergoes security review. Deployments follow hardening guidelines. Operational procedures document incident response. These requirements match enterprise security practices and Zenroom's design supports them naturally.

Deployment scale:

CREDIMI certification infrastructure processes thousands of credentials during testing cycles. The throughput requirements are modest by server standards but the correctness requirements are absolute. A false positive in certification could allow non-compliant wallets into production. A false negative could reject compliant wallets causing project delays.

The deterministic execution and comprehensive testing give confidence in certification results. When a credential fails verification, the error message specifies exactly what failed. This diagnostic capability reduces debugging time and improves interoperability.

11.2 Digital Identity Infrastructure (DIDROOM)

DIDROOM implements W3C Decentralized Identifier and Verifiable Credential specifications as production infrastructure. Organizations use DIDROOM to issue, manage, and verify credentials for employee identity, customer authentication, and service access control.

System architecture:

DIDROOM provides three components: issuer service, holder wallet, and verifier service. Each uses Zenroom for cryptographic operations. The issuer creates credentials signed with organizational keys. The holder stores credentials and creates presentations. The verifier validates presentations against issuer public keys.

The system supports multiple credential formats. Simple credentials use JSON-LD with EdDSA signatures. Privacy-preserving credentials use BBS+ for selective disclosure. The holder wallet allows users to choose which attributes to reveal during presentation. The same credential can be presented differently to different verifiers.

DID management uses did:dyne method implemented in Zenroom. Organizations generate DIDs deterministically from keys. DID documents are signed and published. Updates to DID documents (key rotation, service endpoint changes) are authenticated through control proofs generated by Zenroom contracts.

Privacy characteristics:

Selective disclosure prevents verifiers from learning unnecessary information. An age verification credential containing birthdate, address, and citizenship can prove age without revealing other attributes. The BBS+ implementation ensures the proof is unlinkable: multiple presentations to the same verifier cannot be correlated.

Unlinkability is verified through testing. The test suite generates multiple presentations from the same credential and verifies that presentation proofs contain no common elements. This testing validates that the cryptographic implementation achieves privacy properties claimed by the protocol.

Integration patterns:

DIDROOM integrates with existing identity systems. Organizations with LDAP directories or OAuth providers can use DIDROOM as credential issuance layer. Employees authenticate through existing systems. The issuer service creates verifiable credentials based on authenticated attributes. Employees use these credentials for external service access without involving the corporate authentication system.

This integration pattern separates authentication from authorization. Employees prove identity to external services through credentials without the services contacting the employer. This reduces coupling and improves privacy. Employers do not learn which services employees access.

11.3 Global Passport Project

The Global Passport Project implements digital passport verification for immigration and border control. The system verifies passport authenticity, validates holder identity, and checks travel authorization without requiring central database lookups.

Cryptographic verification:

Modern passports contain chips with digital signatures from issuing governments. The signature covers biometric data and passport details. Verification requires checking the signature against government public keys and validating the certificate chain to a trust anchor.

Zenroom performs signature verification using algorithms specified in ICAO Doc 9303 (the standard for machine-readable travel documents). The system supports RSA and ECDSA signatures on multiple curves. Public key infrastructure uses X.509 certificates with country-specific parameters.

The verification contracts are human-readable. Border control personnel can examine contracts and understand verification logic. This transparency supports audit requirements and builds confidence in automated systems. When verification fails, the error message indicates which check failed: invalid signature, expired certificate, revoked credential.

Offline operation:

Border control points may have limited connectivity. The system operates offline using pre-distributed public key lists. Public keys are distributed through secure channels and updated periodically. The verification contract checks signatures against local public key database without network access.

This offline capability is enabled by Zenroom's isolation. The VM has no network access so contracts designed for online operation cannot accidentally work offline and vice versa. The architecture forces explicit design decisions about connectivity requirements.

Performance requirements:

Immigration lines require processing passengers in seconds. The system must verify passport chips, check biometric matching, and validate travel authorization quickly enough to avoid delays. Cryptographic verification using Zenroom takes single-digit milliseconds. The limiting factor is chip communication and biometric processing, not cryptographic operations.

The deterministic execution helps debugging. When verification is slower than expected, performance testing with instrumentation shows exactly which operations take time. The deterministic behavior means performance issues are reproducible and root causes are identifiable.

11.4 Hyperledger Sawtooth Integration (Sawroom)

Sawroom integrates Zenroom with Hyperledger Sawtooth, a blockchain platform for enterprise applications. The integration allows Sawtooth transaction families to use Zenroom for transaction validation and state updates. This brings Zencode's readable contracts to blockchain applications.

Transaction family implementation:

Sawtooth transaction families define application logic for specific use cases. A supply chain transaction family implements operations for tracking goods. A financial transaction family implements payment and settlement operations. Each family has validation rules and state transition logic.

Sawroom implements transaction families using Zencode contracts. Transaction validation is specified as readable contracts. Auditors can review validation logic without understanding blockchain internals. The contract reads transaction payload, validates according to business rules, and computes state updates.

The integration preserves Zenroom's isolation. Transaction processors run Zenroom in subprocess with transaction data passed as input. Zenroom executes validation contract and returns verdict. The transaction processor applies state updates if validation succeeds. Malicious transactions cannot compromise the node because Zenroom has no access to node resources.

Consensus integration:

Blockchain consensus requires deterministic transaction execution. Validators must reach identical conclusions about transaction validity. Zenroom's deterministic execution aligns perfectly with this requirement. Validators run identical contracts on identical inputs and produce identical outputs.

The determinism is verified through testing across the Sawtooth network. Nodes run on different platforms: Linux on x86, Linux on ARM, containerized deployments. All nodes must reach consensus on transaction validity. The cross-platform determinism testing that validates Zenroom also validates consensus.

Smart contract transparency:

Traditional blockchain smart contracts are opaque to non-developers. Zencode contracts executing in Sawroom are readable by business stakeholders. A supply chain contract specifying custody transfer rules can be reviewed by lawyers and logistics managers. The contract is simultaneously specification and implementation.

This transparency affects governance. Changing contract logic requires proposing new Zencode contracts. Stakeholders review proposals and vote on adoption. The review process is accessible to non-technical participants because contracts are readable. This democratizes blockchain governance beyond developer communities.

11.5 Municipal Digital Democracy (DECODE Project)

The DECODE project for the European Union investigated technologies for digital democracy. Municipal governments in Amsterdam and Barcelona piloted systems where citizens participate in collective decisions through cryptographic petition systems and participatory budgeting. Zenroom provided cryptographic operations ensuring privacy and integrity.

Petition signing:

Citizens sign petitions supporting or opposing municipal policies. Signatures must be verifiable (proving real citizens signed) but anonymous (preventing tracking of citizen political activity). Standard digital signatures fail because signatures are linkable: collecting signature lists enables tracking who signs what.

DECODE implemented blind signature petitions using cryptography from Coconut credentials. Citizens obtain credentials from the municipality proving citizenship. Credentials are blinded so the municipality does not learn which citizen holds which credential. Citizens use credentials to sign petitions. Signatures prove valid citizenship without revealing identity.

Zenroom implements the blind signature protocol. The readable contracts allow citizens to audit the cryptography protecting their privacy. Trust does not require believing experts. The contracts explicitly show that the municipality cannot link signatures to citizens.

Participatory budgeting:

Municipalities allocate budget to projects based on citizen votes. Voting must be private (no one learns how individuals voted) and verifiable (anyone can verify the count is correct). The system uses cryptographic voting where votes are encrypted but the tally is publicly computable.

The voting contracts use additive homomorphic encryption. Votes are encrypted. The encrypted votes are homomorphically summed. The sum is decrypted revealing only the tally, not individual votes. Zenroom computes vote encryption and tally verification. The contracts show explicitly what information is revealed and what remains private.

Real-world deployment:

The Barcelona and Amsterdam pilots involved thousands of citizens making decisions on real municipal budget allocation and policy questions. The systems operated under political pressure and public scrutiny. Technical failures would undermine trust in digital democracy.

Zenroom's stability and transparency proved essential. When citizens questioned how the system worked, contracts could be shown and explained. When verification was needed, any participant could re-execute contracts and verify results. The human-readable nature of Zencode made the cryptography accessible to political discourse rather than confined to technical circles.

Lessons learned:

Digital democracy requires more than correct cryptography. Citizens must trust the system. Trust requires transparency and accessibility. Zencode's readable contracts provided transparency that proprietary systems could not match. The open source nature allowed independent audit by civil society organizations.

The project demonstrated that complex cryptographic protocols can be deployed at municipal scale. The constraints of democratic legitimacy align with Zenroom's design philosophy: transparency, verifiability, and accessible audit. These same principles serve enterprise and government applications where stakeholders demand accountability.

12 Comparative Analysis

Technical decisions require understanding tradeoffs. No technology is universally superior. Each approach optimizes for different priorities. This section compares Zenroom and Zencode against alternatives that organizations commonly evaluate. The comparisons focus on measurable differences rather than subjective preferences.

12.1 Zencode vs. Solidity

Solidity is the dominant language for Ethereum smart contracts. Organizations building blockchain applications often evaluate whether to use Solidity on Ethereum or alternative approaches. The comparison is not apples-to-

apples because the systems target different architectures, but decision points emerge.

Readability:

Solidity syntax resembles JavaScript. Developers familiar with C-style languages can read Solidity code. However, understanding what Solidity code does requires understanding EVM semantics, gas costs, storage layouts, and calling conventions. A function that appears simple may have subtle behaviors from reentrancy, delegate calls, or storage access patterns.

Zencode syntax is English-like statements. Non-developers can read contracts and understand high-level logic. However, Zencode is not Turing complete. Arbitrary computation requires dropping to Lua. The readability advantage is real for the operations Zencode supports but disappears for computation it cannot express.

Execution model:

Solidity executes on blockchain networks. Execution is public and permanent. Every node in the network executes every contract. This provides strong consistency and censorship resistance but makes privacy difficult and imposes gas costs that grow with network activity.

Zencode executes in isolated VM instances. Execution is private to the calling application. No network is involved. This provides privacy and predictable costs but requires application-level distribution for multi-party scenarios. Organizations wanting blockchain properties must combine Zenroom with separate blockchain infrastructure.

Development ecosystem:

Solidity has extensive tooling: IDEs, debuggers, testing frameworks, formal verification tools. The Ethereum ecosystem includes libraries, standards, and established patterns. Developers can find examples and Stack Overflow answers for common problems.

Zencode ecosystem is smaller. Tooling focuses on contract execution rather than development environments. The scenario system provides reusable components but fewer third-party libraries exist. Organizations may need to implement domain-specific functionality rather than importing existing packages.

When to choose which:

Choose Solidity for applications requiring public blockchain properties: global state, censorship resistance, network effects from existing deployments. Accept gas costs, public execution, and EVM complexity.

Choose Zencode for applications requiring private cryptographic operations, readable contracts for audit, or execution outside blockchain context. Accept limitations on computation and smaller ecosystem.

Many applications need both. Blockchain for coordination and Zenroom for private computation. Sawroom demonstrates this pattern: Zencode for readable business logic, Sawtooth for distributed consensus.

12.2 Zencode vs. Traditional Scripting

Organizations with cryptographic requirements could use Python, JavaScript, or other scripting languages with cryptographic libraries. Why would they choose Zencode instead?

Cryptographic correctness:

Traditional languages with crypto libraries require developers to use libraries correctly. APIs have parameters that must be set appropriately. Algorithms have modes that must be chosen carefully. Padding schemes have security implications. The libraries provide building blocks but developers must assemble them correctly.

Zencode provides high-level operations that encapsulate correct usage. Signing a credential uses appropriate algorithms, encodings, and serialization automatically. Developers specify what to sign, not how to implement signing. This reduces implementation errors but limits flexibility for non-standard cryptography.

Determinism:

Traditional scripting languages are not deterministic. Dictionary iteration order varies. Floating-point behavior differs across platforms. Random number generation pulls from OS sources. Code executing on different machines produces different outputs even with identical inputs.

Zencode execution is deterministic by design. The same contract with the same inputs produces identical outputs on all platforms. This enables distributed verification but requires constraints that traditional languages do not impose.

Isolation:

Traditional scripts run with the permissions of the invoking process. They can read files, make network requests, spawn subprocesses. This flexibility enables wide applicability but creates security concerns when executing untrusted code.

Zencode execution is isolated by the VM. Contracts cannot access external resources. This security enables executing contracts from untrusted sources but prevents operations requiring system access.

Auditability:

Traditional scripts are auditable if reviewers understand the language and libraries. Cryptographic code reviews require specialists familiar with implementation pitfalls. The audit must verify not just logic correctness but cryptographic correctness.

Zencode contracts are auditable by non-specialists for high-level logic. The contract shows what operations execute but encapsulates implementation details. Auditors verify business logic matches requirements. Crypto-

graphic correctness is verified once in the VM implementation rather than per-contract.

When to choose which:

Choose traditional scripting for general-purpose computation, integration with existing systems, or operations requiring system access. Accept responsibility for correct cryptographic implementation and platform-specific behavior.

Choose Zencode for cryptographic operations requiring determinism, isolation, or audit by non-specialists. Accept limitations on general computation and system interaction.

12.3 Zencode vs. Formal Verification Languages

Formal verification languages like F*, Dafny, or Coq prove program properties mathematically. Some cryptographic implementations use formal verification to ensure correctness. How does Zencode compare to formally verified approaches?

Assurance level:

Formal verification provides mathematical proof that code meets specifications. Proofs are checked mechanically. If the specification captures security properties and the proof is valid, the implementation is correct with respect to the specification.

Zencode provides no formal proofs. Correctness is established through testing, code review, and production use. Bugs may exist that testing has not found. This is lower assurance than formal verification.

Specification complexity:

Formal verification requires writing specifications in formal logic. Specifications must be precise enough for mechanical checking. Writing specifications is specialized work requiring expertise in formal methods.

Zencode contracts are executable specifications. Writing contracts requires understanding Zencode syntax but not formal logic. The contracts are specifications and implementations simultaneously. This accessibility comes at the cost of lower assurance.

Verification effort:

Formally verifying cryptographic implementations requires substantial effort. A verified implementation might take months or years to develop. The verification effort exceeds the implementation effort.

Zencode implementation is conventional software development with extensive testing. Development timelines are typical for systems software. The lack of formal verification means less certainty but faster development.

Applicability:

Formal verification applies best to fixed algorithms with clear specifications. A verified implementation of AES or SHA-256 provides high assurance. Verifying complex protocols or business logic is more difficult.

Zencode contracts implement business logic and protocol flows. The logic changes as requirements evolve. Formal verification would require reverification after each change. Testing-based verification allows incremental changes with regression testing.

When to choose which:

Choose formal verification for cryptographic primitives in highest-assurance applications. Accept significant development time and specialized expertise requirements.

Choose Zencode for application-level cryptographic operations where formal verification effort is impractical. Rely on testing, auditing, and production experience for assurance.

Note that Zencode contracts use formally verified primitives where available. The AMCL library includes some verified components. Zencode provides accessible interface while underlying implementation may have formal assurance.

12.4 Total Cost of Ownership Analysis

Cost considerations extend beyond licensing fees. Total cost of ownership includes development, deployment, maintenance, training, and audit. How do costs compare across alternatives?

Development costs:

Solidity development requires blockchain expertise. Developers must understand gas optimization, storage patterns, and security vulnerabilities specific to smart contracts. The learning curve is steep. Experienced Solidity developers command premium salaries.

Zencode development requires understanding the language and scenarios but not low-level cryptographic implementation. Developers can be productive more quickly. However, the smaller ecosystem means less existing code to reuse. Custom functionality requires more development.

Traditional scripting requires cryptographic expertise. Developers must understand algorithm selection, parameter choices, and common pitfalls. Incorrect usage of crypto libraries is common and dangerous. Expert review is essential.

Audit costs:

Solidity contract audits are expensive. The specialized nature of blockchain security and the high cost of vulnerabilities make audits essential. Audits cost tens of thousands to hundreds of thousands of dollars depending on complexity.

Zencode contract audits can be performed by non-cryptographic specialists for business logic verification. The readable syntax reduces audit time. Cryptographic

correctness is verified in the VM rather than per-contract. This separation reduces audit costs for contracts while requiring VM audit.

Traditional script audits require cryptographic expertise if the code performs cryptographic operations. The cost is similar to Solidity audits for comparable functionality.

Operational costs:

Blockchain deployment costs scale with usage due to gas fees. High-frequency applications pay substantial ongoing costs. These costs are inherent to public blockchain architecture.

Zenroom deployment costs are conventional infrastructure costs. Organizations pay for compute resources. Costs scale with usage but at server computing rates rather than blockchain transaction fees.

Traditional scripts have similar operational costs to Zenroom. The main difference is whether scripts execute in isolated VM or full runtime environment.

Maintenance costs:

Blockchain smart contracts are immutable once deployed. Fixes require deploying new contracts and migrating state. This rigidity prevents easy updates but also prevents unauthorized changes.

Zenroom contracts can be updated as needed. Updates require testing and deployment but no blockchain migration. The flexibility reduces maintenance costs but requires operational security to protect against unauthorized changes.

Traditional scripts have similar maintenance characteristics to Zenroom. The main consideration is whether cryptographic libraries need updates when vulnerabilities are discovered.

Training costs:

Blockchain development requires specialized training. Developers learn blockchain concepts, Solidity, and security patterns. The training investment is substantial.

Zencode is more accessible. Developers learn a constrained language focused on cryptographic operations. The learning curve is gentler. However, deep expertise still requires understanding cryptographic concepts.

Traditional scripting builds on existing language knowledge but requires cryptographic training for secure usage.

Lock-in costs:

Public blockchains create ecosystem lock-in. Migrating deployed contracts to different chains is difficult. The DeFi ecosystem demonstrates this: protocols are tied to specific chains.

Zenroom is portable. Contracts run on any platform with a Zenroom implementation. Organizations can change deployment platforms without changing contracts. However, switching to different smart contract systems requires rewriting logic.

Traditional scripts have minimal lock-in beyond the chosen language. Migrating Python to JavaScript requires rewriting but the cryptographic operations remain similar.

Overall assessment:

No approach dominates on all cost dimensions. Organizations must weigh factors based on their specific requirements:

Blockchain makes sense for applications where public coordination justifies gas costs and where ecosystem lock-in is acceptable for network effects.

Zencode makes sense for private cryptographic operations where readable contracts reduce audit costs and where operational flexibility outweighs smaller ecosystem.

Traditional scripting makes sense for general-purpose applications where cryptographic operations are incidental and where development teams have existing expertise.

13 Risk Assessment and Compliance

Organizations deploying cryptographic systems face regulatory requirements, industry standards, and internal governance policies. Compliance is not optional for enterprise and government deployments. This section addresses how Zenroom maps to common compliance frameworks and where organizations must implement additional controls.

13.1 GDPR and Privacy by Design

The General Data Protection Regulation requires privacy by design and default. Systems processing personal data must implement technical measures protecting privacy from the architecture level. Zenroom's design aligns with GDPR principles in several ways but does not constitute full GDPR compliance alone.

Data minimization:

GDPR Article 5(1)(c) requires processing only data necessary for the purpose. Zencode's schema validation enforces explicit declaration of what data enters processing. Contracts cannot access data not declared in Given statements. This technical constraint supports data minimization policies.

Selective disclosure credentials implement data minimization cryptographically. A credential holder proves only necessary attributes without revealing others. An age verification reveals age range without birthdate. This is stronger than policy-based minimization because the verifier cannot access undisclosed data even if they wanted to.

Purpose limitation:

GDPR Article 5(1)(b) requires processing data only for specified purposes. Zencode contracts are purpose-specific. Each contract implements defined operations on defined data. The contract text serves as documentation of processing purpose. Auditors can verify that processing matches declared purpose by reviewing contracts.

Storage limitation:

GDPR Article 5(1)(e) requires retaining personal data only as long as necessary. Zenroom's stateless execution means the VM retains no data between executions. Memory is wiped on termination. This architecture prevents accidental data retention in VM state. Applications using Zenroom remain responsible for storage decisions but the VM does not contribute to retention.

Integrity and confidentiality:

GDPR Article 5(1)(f) requires appropriate security. Zenroom provides cryptographic operations implementing security controls. Encryption protects confidentiality. Signatures ensure integrity. The isolated execution prevents data leaks through VM compromise. These are technical measures supporting the security requirement.

Data portability:

GDPR Article 20 grants individuals right to data portability. Verifiable credentials implement cryptographic portability. Credential holders possess credentials in standard formats. They present credentials to verifiers of their choice. The credentials are not locked in issuer systems. This architecture supports portability requirements.

Limitations:

Zenroom is infrastructure, not a complete GDPR solution. Organizations must implement consent management, data subject rights workflows, breach notification procedures, and data protection impact assessments. The VM provides technical primitives but not legal compliance processes. A GDPR-compliant system uses Zenroom as a component, not as the complete solution.

13.2 Common Criteria and Security Certifications

Common Criteria evaluates security products against standardized requirements. Organizations requiring certified products ask whether Zenroom has CC certification. The answer is nuanced.

Certification status:

Zenroom itself does not hold Common Criteria certification. CC evaluation is expensive and time-consuming. Open source projects rarely pursue CC certification due to cost. Organizations requiring certified components must evaluate whether to pursue certification for their deployment or use Zenroom in applications where certification is not mandatory.

Evaluation Assurance Levels:

Common Criteria defines EAL1 through EAL7 assurance levels. Higher levels require more rigorous evaluation. Zenroom's architecture and development practices align with requirements for mid-level EALs. The deterministic execution, isolation, and testing methodology support evaluation if an organization pursues certification.

The formal grammar specification and documented architecture facilitate evaluation. Evaluators need clear specifications to assess. The whitepaper and technical documentation provide materials evaluators require. Organizations pursuing certification have foundation to build on.

Protection profiles:

Common Criteria uses protection profiles defining security requirements for product categories. No existing protection profile precisely matches Zenroom's capabilities. Organizations might evaluate against cryptographic module profiles or trusted execution environment profiles depending on use case.

Alternative certifications:

FIPS 140-2 and 140-3 certify cryptographic modules. Zenroom uses AMCL which has been separately evaluated but the Zenroom integration is not certified. Organizations can operate Zenroom in FIPS mode using only approved algorithms and key sizes. This provides partial compliance with FIPS requirements.

Industry-specific requirements:

Financial services have PCI-DSS. Healthcare has HIPAA. Each industry has standards. Zenroom provides technical controls supporting these standards but does not constitute compliance. Organizations map Zenroom capabilities to specific requirements in their compliance frameworks.

13.3 Supply Chain Security

Software supply chain attacks compromise development, build, or distribution processes. Organizations must assess supply chain risks when adopting any software component. Zenroom addresses supply chain security through several mechanisms.

Source code availability:

Zenroom source code is publicly available under AGPL license. Organizations can inspect code for backdoors or vulnerabilities. This transparency is fundamental supply chain security. Proprietary software requires trusting vendor claims. Open source enables independent verification.

The public development process on GitHub allows monitoring changes. Organizations can review commit history, contributor identities, and code review discussions. Suspicious changes would be visible in public record.

Build reproducibility:

Zenroom builds are deterministic. Organizations can compile from source and verify the resulting binary matches official releases byte-for-byte. This reproducibility proves

the binary corresponds to public source code. Backdoored binaries would fail reproduction.

Reproducible builds defend against compromised build infrastructure. If the build server were compromised and malicious code injected into binaries, independent compilation would detect the mismatch. Organizations can trust binaries because they can verify them.

Dependency management:

Zenroom has zero external runtime dependencies. The attack surface for supply chain compromise is the code in the repository. There are no package dependencies that could be compromised. This minimal dependency approach reduces supply chain risk.

Build-time dependencies exist: compilers, build tools. Organizations must trust their build environment. The deterministic builds allow verifying that different build environments produce identical results, providing confidence that build tools are not injecting malicious code.

Cryptographic signatures:

Official releases are cryptographically signed. Organizations verify signatures before deployment. This prevents distribution of modified binaries by attackers who compromise distribution channels. The signing key is protected and published through multiple channels.

Vulnerability disclosure:

Zenroom maintains a security contact and responsible disclosure process. Discovered vulnerabilities are addressed promptly. Security advisories provide detailed information allowing organizations to assess impact. This transparency allows organizations to make informed risk decisions.

SBOM and provenance:

Software Bill of Materials documents components in a software package. Zenroom's minimal dependencies simplify SBOM. The SBOM contains Zenroom itself, AMCL, and Lua. Each component is versioned and sourced from known repositories. Supply chain provenance is straightforward to document.

13.4 Cryptographic Agility and Algorithm Transitions

Cryptographic algorithms have lifespans. SHA-1 was deprecated. 1024-bit RSA is weak. Quantum computers will break elliptic curves. Organizations need cryptographic agility: the ability to transition to new algorithms when old algorithms become unsafe.

Algorithm selection flexibility:

Zenroom supports multiple algorithms for each operation. Signatures can use ECDSA, EdDSA, RSA, Schnorr, or post-quantum schemes. Organizations choose algorithms matching current threat models. When threats evolve, they can transition to stronger algorithms.

The scenario system allows adding new algorithms without VM changes. When quantum-safe algorithms emerge, they can be implemented as new scenarios. Existing contracts using old algorithms continue working. New contracts use new algorithms. Gradual transition is possible.

Migration patterns:

Hybrid approaches ease transitions. Credentials can carry multiple signatures using different algorithms. Verifiers accept any valid signature. During transition, old verifiers use old signatures, new verifiers use new signatures. Both work during migration period.

The readable contract syntax helps transition planning. Organizations can identify which contracts use which algorithms by reading contract text. Migration impact assessment is straightforward. Contracts are documentation of algorithm usage.

Post-quantum readiness:

The ML-KEM and ML-DSA implementations prepare for quantum transition. Organizations can begin using post-quantum algorithms today for data with long confidentiality requirements. The hybrid approaches allow combining traditional and post-quantum cryptography hedging against risks in both.

Deprecation management:

When algorithms are deprecated, Zenroom can disable them in new deployments while maintaining compatibility for verification of old signatures. A contract can verify historical ECDSA signatures while creating new signatures using post-quantum schemes. This asymmetry supports gradual migration.

Standardization tracking:

Zenroom follows cryptographic standards from NIST, IETF, and W3C. When standards evolve, implementations are updated. Organizations benefit from standards compliance without implementing standards themselves. The scenario system encapsulates standards compliance.

13.5 Audit Trail and Accountability

Accountability requires recording who did what when and providing evidence for later review. Cryptographic systems must support audit requirements without undermining privacy. The balance is delicate.

Execution determinism enables audit:

Deterministic execution means audit records can be verified by re-execution. An auditor receives contract, inputs, and claimed outputs. They re-execute the contract with the same inputs. If outputs match, the execution is verified. This cryptographic audit is stronger than trusting execution logs.

The determinism also enables dispute resolution. If parties disagree about execution results, a neutral third party can execute the contract and determine the correct result.

The mathematical certainty of deterministic execution eliminates ambiguity.

Immutable contracts as audit evidence:

Zencode contracts are text documents. They can be stored immutably: signed, timestamped, and archived. Audit trails reference specific contract versions. Years later, auditors can examine the exact contract that executed.

The human-readable syntax means archived contracts remain comprehensible. Binary bytecode or optimized representations may become unreadable as tools evolve. Text contracts with natural language syntax remain accessible to future auditors.

Credential provenance:

Cryptographic signatures provide credential provenance. A verifiable credential contains issuer signatures proving issuance. The signature proves who issued, what was issued, and when. This provenance is cryptographic evidence, not self-reported logs.

The unlinkable presentation proofs maintain privacy while providing verification. Auditors verify that credentials were properly issued and validated without learning individual credential usage patterns. The cryptography separates audit capabilities from surveillance capabilities.

Accountability without surveillance:

Traditional audit approaches log all operations and require accessing logs to verify behavior. This creates privacy risks. Logs become surveillance data. Zenroom enables accountability through verifiable computation instead of logging.

A contract execution produces outputs and optionally cryptographic proofs of correct execution. Third parties verify proofs without accessing execution logs. The zero-knowledge properties of the cryptography allow proving correct execution without revealing inputs or intermediate states.

Regulatory requirements:

Financial regulations require audit trails. Healthcare requires access logs. Each domain has requirements. Zenroom provides cryptographic primitives supporting audit requirements. Applications must implement appropriate logging and access controls at the application layer. The VM focuses on cryptographic correctness while applications handle regulatory compliance.

14 Future Directions

14.1 Roadmap and Planned Enhancements

14.2 Research Collaborations

14.3 Community and Ecosystem Growth

14.4 Long-Term Support and Maintenance Commitment

15 Conclusion

The security landscape facing organizations has never been more complex. Post-quantum cryptography transitions loom. Privacy regulations tighten. Supply chain attacks multiply. Digital identity infrastructure becomes critical. Traditional approaches to cryptographic software development are not keeping pace with these challenges.

Zenroom and Zencode represent a different approach. The design starts from language-theoretic security principles, applies them rigorously through architecture, and produces a system where security properties emerge from structure rather than defensive coding. This is not theoretical research but production infrastructure proven at scale.

Core achievements:

The technical contributions are measurable. A process VM with three-compartment memory isolation enforcing recognition-processing-output separation. A domain-specific language with formal grammar where contracts are simultaneously specifications and implementations. Deterministic execution enabling reproducible verification across platforms. Zero external dependencies reducing attack surface. Cryptographic operations from legacy algorithms through post-quantum schemes integrated in a single codebase.

These technical properties solve concrete problems. Verifiable credentials with selective disclosure enable privacy-preserving identity without centralized tracking. Blockchain integration provides readable smart contracts accessible to non-developers. Municipal democracy systems give citizens cryptographic guarantees about voting integrity. Border control systems verify passports offline without database dependencies. These are operational systems processing real users' data with real consequences.

Design philosophy vindicated:

The fundamental bet was that security through simplicity and transparency beats security through complexity and obscurity. Eight years of production use validate this bet. The human-readable contracts enable audit by stakeholders who matter: regulators, citizens, lawyers, compliance officers. The isolation architecture prevents entire vulnerability classes rather than defending against them after the fact. The deterministic execution makes contract behavior verifiable by anyone with sufficient interest.

This philosophy has costs. Zencode is not Turing complete. The VM cannot access filesystems or networks. General computation requires different tools. These limitations are features. They force clarity about what the system does and what it cannot do. Organizations know exactly what they are getting because the constraints are explicit.

Production maturity:

Pilot projects demonstrate concepts. Production systems prove viability. Zenroom deployments span EU digital identity certification, municipal democracy in major European cities, blockchain platforms, and government border control. The diversity of deployments demonstrates that the design generalizes beyond narrow use cases.

The maturity shows in operational characteristics. Performance is predictable: simple operations take milliseconds, complex ones take tens of milliseconds. Memory usage is bounded and configurable. Error messages are structured and diagnostic. Updates deploy without breaking existing contracts. These operational qualities matter more than peak performance or minimal footprint.

For decision makers:

CTOs evaluating Zenroom should ask whether their requirements align with Zenroom's strengths. If you need private cryptographic operations with audit requirements, readable contracts for compliance review, deterministic execution for reproducibility, or deployment flexibility across platforms from embedded to cloud, Zenroom deserves serious consideration.

If you need general-purpose computation, blockchain properties like public state and censorship resistance, or ecosystem lock-in benefits from established platforms, other approaches may fit better. The comparative analysis section provides frameworks for mapping requirements to technical choices.

CISOs evaluating security should recognize that Zenroom follows security-by-design principles that formal methods researchers and language security experts advocate. The LangSec foundation is not marketing but peer-reviewed research. DARPA's formal methods initiatives align with the approach. The architecture prevents problems rather than detecting and blocking attacks.

This does not mean Zenroom is perfectly secure. No complex software is. The known limitations section documents where protections are incomplete. Side-channel resistance is limited without hardware support. Implementation bugs exist despite testing. Cryptographic assumptions could be invalidated. The honest acknowledgment of limitations builds realistic security assessments.

Strategic considerations:

Cryptographic infrastructure has decades-long lifespans. Organizations deploying identity systems or cryptographic protocols today commit to supporting them for years or decades. This long timeline makes certain properties essential.

Cryptographic agility matters because algorithms will be broken. Systems deployed today must survive post-quantum transitions and unexpected weaknesses in current schemes. Zenroom's multi-algorithm support and scenario-based extensibility provide migration paths.

Auditability matters because regulations and security requirements evolve. Contracts written today will be audited under future requirements. The human-readable syntax and immutable text representation ensure contracts remain comprehensible to future auditors using future tools.

Portability matters because deployment targets change. A system might start on servers, migrate to mobile apps, expand to embedded devices, or move to edge computing. Zenroom's minimal dependencies and broad platform support enable evolution without rewrites.

The next generation of cryptographic infrastructure:

Digital identity, verifiable credentials, privacy-preserving computation, and decentralized trust are not future technologies but present requirements. Organizations need infrastructure supporting these capabilities today. Traditional approaches using general-purpose languages with crypto libraries are not sufficient. The complexity is too high, the audit burden too great, the error potential too large.

Zenroom demonstrates that purpose-built infrastructure for cryptographic operations can be simpler, more auditable, and more secure than general-purpose alternatives. The domain-specific approach trades generality for correctness. The readable syntax trades implementation flexibility for audit accessibility. The isolation architecture trades system integration for security guarantees.

These tradeoffs align with the requirements of critical cryptographic infrastructure. Systems processing identity credentials, financial transactions, or democratic decisions need correctness more than flexibility, auditability more than performance, and security more than convenience.

Call to evaluation:

This whitepaper provides technical foundation for informed evaluation. The architecture section explains how the system works. The security analysis documents what protections exist and what limitations remain. The case studies demonstrate production viability. The comparative analysis frames decision criteria.

Organizations should evaluate Zenroom through proof-of-concept deployments. The barrier to experimentation is low: download a binary, write contracts, test with actual data. The deterministic execution means proof-of-concept results predict production behavior. Time invested in evaluation produces reliable understanding of operational characteristics.

The open source nature enables deep evaluation. Organizations can inspect source code, review test suites, reproduce builds, and verify cryptographic implementa-

tions. The transparency removes barriers to trust. Security through obscurity is not an option. Security through verifiable correctness is the standard.

Final assessment:

Zenroom is production-ready infrastructure for cryptographic operations requiring human-readable contracts, deterministic execution, and verifiable computation. It is not a universal solution. It is a specialized tool solving specific problems well.

For organizations with requirements matching Zenroom's strengths, adoption provides measurable benefits: reduced audit costs through readable contracts, increased security through isolation architecture, operational flexibility through platform portability, and cryptographic agility through multi-algorithm support.

The technical foundations are sound. The production deployments are real. The development community is active. The architectural principles align with formal methods research and security best practices. Organizations evaluating next-generation cryptographic infrastructure should include Zenroom in their assessment.

Acknowledgments

A Zencode Formal Grammar Specification

This appendix provides the complete formal grammar of the Zencode language in Extended Backus-Naur Form (EBNF) following the ISO/IEC 14977 standard. This formalization serves multiple purposes: it provides a precise, unambiguous specification of valid Zencode contracts; it enables automated parser generation; it facilitates formal verification and static analysis; and it documents the language for implementers and tool developers.

The grammar is context-free at its core with context-sensitive restrictions enforced during semantic analysis. This positioning in the Chomsky hierarchy is intentional. Context-free grammars are well-understood, efficient to parse, and amenable to formal analysis. The context-sensitive extensions handle phase ordering and memory access rules that cannot be expressed in pure context-free form but are essential for security.

A.1 Grammar Structure

The Zencode grammar consists of several layers:

Lexical layer: Defines tokens like identifiers, keywords, literals, and delimiters. Keywords are reserved words (Given, When, Then, If, Foreach) that trigger state transitions in the parser. Variable placeholders are marked by single quotes surrounding identifier strings.

Syntactic layer: Defines valid statement structures. A contract is a sequence of statements organized into phases. Each phase has specific keywords and permitted statement types. The grammar enforces that statements appear in valid order and that control structures are properly nested.

Semantic layer: Enforces constraints not expressible in pure context-free grammar. Phase ordering requires that Given statements precede When statements which precede Then statements. Memory access permissions vary by phase. Control flow depth is bounded. These constraints are verified during parsing through state machine transitions.

A.2 Phase Structure

The grammar reflects Zencode's phase-based execution model:

contract = [preamble] , body

preamble = { rule | scenario }

body = [given_phase] , [when_phase] , [then_phase]

The preamble contains optional directives that configure parsing and execution. The rule directive sets VM behavior like input encoding or version requirements. The scenario directive loads domain-specific statement libraries.

The body contains the actual computational contract. The Given phase declares and validates inputs. The When phase performs transformations. The Then phase formats outputs. Each phase is optional but if present they must appear in this order.

A.3 Statement Patterns

Zencode statements follow a pattern-matching syntax:

statement_pattern = pattern_element , { pattern_element }

pattern_element = literal_text | variable_placeholder

variable_placeholder = "'" , variable_name , "'"

A pattern consists of literal text interspersed with variable placeholders. The parser matches input statements against registered patterns from loaded scenarios. When a match succeeds, variable values are extracted and passed to the corresponding implementation function.

This pattern-matching approach enables natural language-like syntax while maintaining formal parsability. The single-quote delimiters make variable boundaries explicit, preventing ambiguity in pattern matching.

A.4 Control Flow

The grammar defines two control structures:

Conditional branching:

if_block = if_statement , [whenif_phase] , [thenif_phase] , endif_statement

Conditional execution within When and Then phases. The condition is evaluated during execution. If true, the whenif and thenif statements execute. The branch does not loop; it executes at most once per contract execution.

Iteration:

foreach_block = foreach_statement , [whenforeach_phase] , endforeach_statement

Bounded iteration over collections. The maximum iteration count is set at VM initialization. The loop body cannot modify the collection being iterated. Foreach blocks can contain if blocks but cannot be nested recursively beyond implementation limits.

A.5 Context-Sensitive Constraints

Several constraints are enforced by the parser but cannot be expressed in the context-free grammar:

① Phase ordering must be: scenario/rule, then Given, then When, then Then

② Control structures must be properly paired (if with endif, foreach with endforeach)

③ Statement patterns must match registered patterns from loaded scenarios

④ Variable names must be valid identifiers without spaces

⑤ Nesting depth of control structures is bounded by VM configuration

⑥ Memory access permissions vary by phase and are checked by the VM

These constraints are verified during parsing through the state machine that tracks current phase and validates transitions.

A.6 Extensibility

The grammar allows extension through scenario modules. New scenarios register additional statement patterns without modifying the core grammar. A pattern registration specifies:

- The statement text with variable placeholders
- Which phase(s) the statement is valid in
- The implementation function to invoke when the pattern matches

Pattern registration is checked for ambiguity. If two patterns could match the same input, the parser rejects the registration. This prevents undefined behavior from ambiguous grammars.

A.7 Complete Grammar Specification

The complete EBNF grammar is provided in the accompanying file zencode.ebnf. This machine-readable specification can be used with parser generators (ANTLR, Yacc, PEG) or verification tools. The grammar includes:

- All terminal and non-terminal symbols
- Production rules for every language construct
- Lexical definitions for tokens
- Examples of valid contracts
- Documentation of semantic constraints

The grammar is maintained as part of the Zenroom source distribution. Changes to the language require corresponding updates to the formal grammar. This coupling prevents specification drift and ensures the grammar accurately reflects implementation behavior.

A.8 Verification Applications

The formal grammar enables several verification approaches:

Static analysis: Tools can parse contracts without executing them, checking for syntax errors, undefined variables, or unreachable code. The grammar makes contract structure explicit and amenable to automated checking.

Contract validation: Organizations can verify that contracts conform to policy requirements by analyzing the Abstract Syntax Tree produced by parsing. For example, checking that contracts do not use forbidden operations or access prohibited data.

Documentation generation: The grammar structure allows automated extraction of all valid statement patterns, producing comprehensive reference documentation directly from the implementation.

Test generation: Formal grammars can seed fuzz testing by generating valid contracts that exercise different code paths. The grammar defines the valid input space, making coverage-guided fuzzing more effective.

Cross-validation: Different tools can parse the same contract and verify they produce equivalent ASTs. This helps prevent parser differentials between tools that claim to support Zencode.

The grammar formalization is not academic exercise but practical engineering. It makes Zencode behavior precise, testable, and verifiable by third parties without requiring access to implementation source code.

B Zencode Syntax Reference

This appendix provides a practical reference for writing Zencode contracts. The formal grammar in the previous appendix defines what is syntactically valid. This reference shows what statements exist, what they do, and how

to use them. The organization follows the phase structure: Given, When, Then, with control flow and directives covered separately.

B.1 Directive Statements

Directives configure contract execution and must appear before the Given phase. They control parsing, validation, and output behavior.

Scenario declaration:

```
Scenario 'scenario_name'
Scenario 'ecdh': 'ethereum': 'credential'
```

Loads scenario modules providing domain-specific statements. Multiple scenarios can be declared on one line separated by colons. Common scenarios: ecdh, eddsa, credential, ethereum, bitcoin, w3c, coconut.

Rule directives:

```
Rule check version 5.0
Rule input encoding base64
Rule input format json
Rule output encoding hex
Rule output format cbor
Rule unknown ignore
```

Version checking ensures contract compatibility. Encoding rules specify how input and output data is encoded. Format rules specify serialization format (json, cbor, msgpack). The unknown ignore rule suppresses errors for unrecognized input fields.

B.2 Given Phase: Input Loading and Validation

The Given phase loads and validates input data. Every piece of data used in the contract must be loaded explicitly.

Identity declaration:

```
Given I am 'Alice'
Given I am known as 'Bob'
Given my name is in a 'string' named 'username'
```

Declares the identity of the contract executor. This affects which nested JSON objects are accessible with the "my" keyword.

Loading simple objects:

```
Given I have a 'string' named 'message'
Given I have a 'number' named 'amount'
Given I have a 'hex' named 'publicKey'
Given I have a 'base64' named 'signature'
Given I have an 'integer' named 'counter'
```

Loads atomic values with specified encoding. Supported encodings: string, number, integer, hex, base64, base58, url64, binary, time.

Loading arrays:

```
Given I have a 'string array' named 'names'
Given I have a 'number array' named 'values'
Given I have a 'hex array' named 'keys'
```

Arrays are homogeneous collections. The encoding specifies element type.

Loading dictionaries:

```
Given I have a 'string dictionary' named 'config'
Given I have a 'number dictionary' named 'balances'
```

Dictionaries are key-value maps where keys are strings and values have specified encoding.

Loading from nested objects:

```
Given I have my 'keypair'
Given I have my 'string' named 'email'
Given I have a 'number' named 'price' in 'product'
Given I have a 'string' named 'city' inside 'address'
```

The "my" keyword accesses objects nested under the declared identity. The "in" or "inside" keywords access objects nested under named parents.

Loading cryptographic schemas:

```
Given I have my 'keyring'
Given I have a 'public key' from 'Alice'
Given I have a 'verifiable credential' named 'diploma'
Given I have a 'bitcoin address' named 'wallet'
```

Schemas are complex objects with structure defined by scenarios. The schema name matches the scenario terminology.

B.3 When Phase: Data Processing and Transformation

The When phase performs computation, manipulates data, and executes cryptographic operations.

Creating new objects:

```
When I create the random 'nonce'
When I create the random object of '256' bits
When I create the array of '10' random numbers
When I set 'result' to '42' as 'number'
When I write string 'Hello' in 'greeting'
```

Creates objects with specified values or random generation. The "set" and "write" statements create variables with explicit values.

Arithmetic operations:

```
When I create the result of 'a' + 'b'
When I create the result of 'x' - 'y'
When I create the result of 'price' * 'quantity'
When I create the result of 'total' / 'count'
When I create the result of 'value' % 'modulus'
```

Basic arithmetic. Operations work on numbers. Results are named "result" and typically renamed immediately.

String manipulation:

```
When I append 'suffix' to 'text'
When I split the leftmost '4' bytes of 'data'
When I split the rightmost '8' bytes of 'data'
```

String operations. Append concatenates strings. Split extracts substrings from left or right.

Object manipulation:

```
When I rename the 'oldName' to 'newName'
When I copy 'source' to 'destination'
When I delete 'temporary'
When I remove 'item' from 'collection'
```

Rename changes object names. Copy duplicates objects. Delete removes objects. Remove extracts from collections.

Array operations:

```
When I randomize the 'items' array
When I pick the random object in 'choices'
When I create the copy of element '3' from array 'list'
When I create the flat array of contents in 'nested'
When I create the flat array of keys in 'dictionary'
```

Randomize shuffles arrays. Pick selects random element. Copy element extracts by index. Flatten operations convert nested structures to flat arrays.

Hashing:

```
When I create the hash of 'message'
When I create the hash of 'data' using 'sha512'
When I create the HMAC of 'message' with key 'secret'
When I create the key derivation of 'password'
When I create the pbkdf2 of 'password' with salt 'salt'
```

Cryptographic hashing. Default hash is SHA-256. Specific algorithms can be requested. HMAC provides keyed hashing. KDF and PBKDF2 derive keys from passwords.

Signature operations:

```
When I create the signature of 'message'
When I create the ecdh signature of 'data'
When I create the eddsa signature of 'document'
When I verify the 'signature'
When I verify the 'message' has a signature in 'signature' by 'Alice'
```

Signing creates signatures using the executor's private key. Verification checks signatures against public keys. The algorithm matches the scenario.

Encryption and decryption:

```
When I encrypt the 'plaintext' to 'Bob'
When I decrypt the 'ciphertext' from 'Alice'
When I encrypt the secret message 'text' with 'password'
```

Asymmetric encryption to public keys. Symmetric encryption with passwords. Decryption requires appropriate private key or password.

Key operations:

```
When I create the ecdh key
When I create the ecdh key with secret 'seed'
When I create the eddsa key
When I create the public key
When I create the key derivation of 'master' with path 'm/0/1'
```

Key generation. Random generation or deterministic from seed. Public key extraction. Hierarchical derivation for HD wallets.

Credential operations:

```
When I create the verifiable credential
When I create the credential signature
When I create the credential proof
When I verify the 'credential'
When I create the selective disclosure of 'credential' revealing 'attribu
```

W3C Verifiable Credential operations. Issuance, signing, proof generation, verification, and selective disclosure.

Blockchain operations:

```
When I create the bitcoin address
When I create the ethereum address
When I create the bitcoin transaction
When I create the ethereum transaction
When I sign the 'transaction'
```

Blockchain-specific address derivation and transaction creation. Signing produces blockchain-compatible signatures.

B.4 Control Flow

Control structures enable conditional execution and iteration.

Conditional execution:

```
If I verify the 'signature'
When I create the 'result'
Then print the 'result'
Endif

If 'balance' is found in 'accounts'
When I compute the 'transaction'
Endif
```

If blocks execute conditionally. The condition must be a boolean statement. When and Then statements can appear inside If blocks.

Iteration:

```
Foreach 'item' in 'items'
```

```
When I create the hash of 'item'
Endforeach

Foreach 'name' in 'users'
When I create the credential for 'name'
Endforeach
```

Foreach loops iterate over array elements. The loop variable is bound to each element in sequence. Maximum iterations are bounded by VM configuration.

B.5 Then Phase: Output Generation

The Then phase selects and formats output data.

Printing objects:

```
Then print the 'result'
Then print my 'output'
Then print all data
Then print data
```

Print statements copy objects to output. Specific objects can be selected. Print all data outputs everything in the ACK workspace.

Output encoding:

```
Then print the 'data' as 'hex'
Then print the 'signature' as 'base64'
Then print the 'key' as 'base58'
```

Output encoding can be specified per object. This overrides the rule output encoding directive.

B.6 Statement Modifiers and Prefixes

Statements accept optional modifiers that adjust parsing behavior without changing semantics.

Common prefixes:

- I, that, the, a, an: articles and pronouns for natural reading
- have, my, known, valid: possession and validation indicators
- all, inside, in: collection and nesting indicators

These words are syntactic sugar. ”Given I have a ’string’” is equivalent to ”Given have string”. The readable form is preferred for human consumption.

And keyword:

```
Given I have a 'string' named 'first'
And I have a 'string' named 'second'
And I have a 'number' named 'count'
```

And continues the previous phase. It is equivalent to repeating the phase keyword. Used to avoid repetitive Given/When/Then statements.

B.7 Common Patterns

Practical contract patterns that appear frequently in production use.

Hash and sign:

```
Given I have a 'string' named 'document'
Given I have my 'keyring'
When I create the hash of 'document'
When I create the signature of 'hash'
Then print the 'signature'
```

Standard signing pattern: hash the input, sign the hash.

Verify and process:

```
Given I have a 'string' named 'message'
Given I have a 'signature'
Given I have a 'public key' from 'sender'
If I verify the 'message' has a signature in 'signature' by 'sender'
When I process the 'message'
Then print the 'result'
Endif
```

Conditional processing based on signature verification.

Credential issuance:

```
Scenario 'credential'
Given I am 'issuer'
Given I have my 'keyring'
Given I have a 'string dictionary' named 'claims'
When I create the verifiable credential
When I create the credential signature
Then print the 'verifiable credential'
```

W3C Verifiable Credential issuance flow.

Selective disclosure:

```
Scenario 'credential'
Given I am 'holder'
Given I have my 'verifiable credential'
Given I have a 'string array' named 'revealed'
When I create the selective disclosure revealing 'revealed'
Then print the 'credential presentation'
```

BBS+ selective disclosure presentation.

B.8 Error Messages and Debugging

Understanding error messages helps debug contract issues.

Pattern not found:

```
[!] Zencode line 12 pattern not found:
    Given I have a 'nonexistent' named 'object'
```

No registered pattern matches the statement. Check spelling, scenario loading, and statement syntax.

Schema validation failure:

```
[!] Zencode line 5 schema validation failed:
    Given I have a 'number' named 'text'
```

Input data does not match declared schema. The object exists but has wrong type or encoding.

Missing object:

```
[!] Zencode line 8 object not found:
    When I create the hash of 'missing'
```

Referenced object was not loaded in Given phase. Add appropriate Given statement.

Phase violation:

```
[!] Zencode line 15 invalid phase transition:
    Given I have a 'string' (after When)
```

Statement appears in wrong phase. Given statements cannot appear after When statements.

B.9 Best Practices

Guidelines for writing maintainable contracts.

Explicit loading: Load only data needed for computation. Avoid loading unnecessary input that increases attack surface.

Immediate renaming: When statements create objects with generic names like ”result” or ”random_object”, rename immediately to descriptive names.

Clear variable names: Use descriptive names matching domain terminology. ”userCredential” is better than ”vc1”.

Comments: Use hash comments to document complex logic or business rules.

Error handling: Use If blocks to handle verification failures gracefully rather than allowing contract to fail.

Minimal scope: Keep contracts focused on single purpose. Multiple small contracts are better than one large contract doing many things.

Schema validation: Let Given phase validation catch bad input. Do not implement redundant validation in When phase.

This syntax reference covers common operations. Complete statement lists are generated from scenario implementations. Use the statement discovery feature to list all available patterns for loaded scenarios.

C Cryptographic Algorithm Details

This appendix provides technical specifications for cryptographic algorithms implemented in Zenroom. The information serves multiple purposes: security analysis, interoperability verification, and implementation validation. Each algorithm section includes mathematical foundations, implementation details, parameter choices, and security considerations.

C.1 Hash Functions

SHA-256:

- Standard: FIPS 180-4
- Output size: 256 bits
- Block size: 512 bits
- Security level: 128 bits (birthday bound)
- Implementation: AMCL hash library

SHA-256 is the default hash function for operations not specifying an algorithm. The implementation follows the FIPS specification exactly including padding, endianness, and initialization vectors. Message digests are deterministic: the same input always produces the same hash.

SHA-512:

- Standard: FIPS 180-4
- Output size: 512 bits
- Block size: 1024 bits
- Security level: 256 bits (birthday bound)
- Implementation: AMCL hash library

SHA-512 provides higher security margin than SHA-256 at cost of larger output and slower computation. Used for CSPRNG seeding and when explicit algorithm selection requests it.

SHAKE256:

- Standard: FIPS 202
- Output size: Variable (extendable-output function)
- Security level: Up to 256 bits
- Implementation: Keccak-based

SHAKE256 is an extendable-output function based on Keccak. Used for key derivation where variable-length output is beneficial.

C.2 Elliptic Curve Cryptography

NIST P-256 (secp256r1):

- Curve equation: $y^2 = x^3 * 3x + b$ over prime field $\mathbb{F}_p$
- Prime: $p = 2^{256} * 2^{224} + 2^{192} + 2^{96} * 1$
- Order: $n = 2^{256} * 2^{224} + 2^{192} * 0x...$ (slightly less than 2^{256})
- Cofactor: 1
- Security level: 128 bits
- Implementation: AMCL ECP module

Point operations use projective coordinates for efficiency. Point multiplication uses windowed non-adjacent form (wNAF) for scalar multiplication. Point validation checks

that points satisfy the curve equation and have the correct order.

secp256k1:

- Curve equation: $y^2 = x^3 + 7$ over prime field $\mathbb{F}_p$
- Prime: $p = 2^{256} * 2^{32} * 977$
- Order: $n =$ 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEBAAEDCE6AF48A03BBFD25E8CD0364141
- Cofactor: 1
- Security level: 128 bits
- Implementation: AMCL ECP module

The curve used by Bitcoin and Ethereum. The special form of the prime enables optimized field arithmetic. Endomorphism acceleration is available but not currently used in Zenroom.

Ed25519:

- Curve: Edwards curve $*x^2 + y^2 = 1 * (121665/121666)x^2y^2$
- Base field: $\mathbb{F}_p$ where $p = 2^{255} * 19$
- Group order: $2^{252} + 27742317777372353535851937790883648493$
- Cofactor: 8
- Security level: 128 bits
- Implementation: EdDSA with SHA-512

Ed25519 uses twisted Edwards curve arithmetic. The curve has complete addition formulas resistant to timing attacks. Signatures are deterministic following EdDSA specification. Public keys are 32 bytes, signatures are 64 bytes.

BLS12-381:

- Curve type: Barreto-Lynn-Scott pairing-friendly curve
- Embedding degree: 12
- Base field: $\mathbb{F}_q$ where q is 381-bit prime
- Group orders: Both G_1 and G_2 have prime order r (255 bits)
- Security level: 128 bits
- Implementation: AMCL BLS module

BLS12-381 supports efficient pairing computation enabling advanced schemes. The curve provides 128-bit security with reasonable performance. G1 is curve over base field, G2 is curve over extension field. Pairings map $G_1 \ G_2 \rightarrow G_T$ where G_T is multiplicative group in $\mathbb{F}_{q^{12}}$.

C.3 Signature Schemes

ECDSA:

- Standard: FIPS 186-4 (verification), RFC 6979 (deterministic signing)
- Curves: P-256, secp256k1
- Hash function: SHA-256
- Signature size: 64 bytes (r, s components)
- Security: Relies on discrete logarithm problem

Signing process:

1. Compute message hash $h = H(m)$
2. Derive deterministic nonce k from private key and hash using HMAC-DRBG (RFC 6979)
3. Compute point $R = kG$ where G is generator
4. Set $r = x_R \bmod n$ where x_R is x-coordinate of R
5. Compute $s = k^{*1}(h + r \cdot d) \bmod n$ where d is private key
6. Return signature (r, s)

Verification checks that $R = (h \cdot s^{*1})G + (r \cdot s^{*1})Q$ where Q is public key, and that x-coordinate of R equals r.

EdDSA:

- Standard: RFC 8032
- Curve: Ed25519
- Hash function: SHA-512
- Signature size: 64 bytes
- Security: Relies on discrete logarithm problem

EdDSA signing is deterministic by design. The nonce is computed as $r = H(H(k)_{32..64}||m)$ where k is private key and m is message. This eliminates randomness failures that have broken ECDSA implementations.

Schnorr Signatures:

- Standard: BIP 340 (for secp256k1)
- Curves: secp256k1, Ed25519-compatible
- Hash function: SHA-256 (tagged hash for Bitcoin)
- Signature size: 64 bytes
- Security: Relies on discrete logarithm problem with proven security in random oracle model

Schnorr signatures enable key aggregation and multi-signatures. The signature equation is $s = k + H(R||P||m) \cdot x$ where k is nonce, $R = kG$, P is public key, m is message, and x is private key.

BLS Signatures:

- Curve: BLS12-381
- Signature size: 96 bytes (G1 element) or 192 bytes (G2 element)
- Security: Relies on computational Diffie-Hellman problem in pairing groups
- Properties: Signatures can be aggregated efficiently

BLS signature on message m with private key x is $\sigma = x \cdot H(m)$ where H maps messages to curve points. Verification uses pairing: $e(\sigma, G) = e(H(m), P)$ where $P = xG$ is public key. Multiple signatures on different messages can be aggregated: $\sigma_{agg} = \sum \sigma_i$.

C.4 Post-Quantum Cryptography

ML-KEM (Kyber):

- Standard: NIST FIPS 203
- Parameter sets: ML-KEM-512, ML-KEM-768, ML-KEM-1024
- Security levels: 128, 192, 256 bits respectively
- Key sizes: Public keys 800-1568 bytes, private keys 1632-3168 bytes
- Ciphertext sizes: 768-1568 bytes
- Based on: Module Learning With Errors (MLWE) problem

ML-KEM-768 is the default parameter set providing 192-bit security against quantum adversaries. The scheme uses structured lattices with polynomial rings $\mathbb{Z}_q[X]/(X^{256} + 1)$ where $q = 3329$.

Key generation produces public key $(A, t = As+e)$ where A is public matrix, s is secret vector, and e is error vector. Encapsulation generates shared secret K and ciphertext (u, v) encrypting K. Decapsulation recovers K using private key.

ML-DSA (Dilithium):

- Standard: NIST FIPS 204
- Parameter sets: ML-DSA-44, ML-DSA-65, ML-DSA-87
- Security levels: 128, 192, 256 bits
- Public key sizes: 1312-2592 bytes
- Signature sizes: 2420-4595 bytes
- Based on: Module Learning With Errors and Module Short Integer Solution

ML-DSA-65 provides 192-bit security with reasonable signature size. Signatures are larger than classical schemes but verification is efficient. The scheme uses rejection sampling to ensure signatures leak no information about the private key.

NTRU:

- Parameter sets: Various (ntru-hps-2048-509, ntru-hrss-701)
- Security levels: 128-256 bits
- Key sizes: 935-1230 bytes (public), 935-1450 bytes (private)
- Ciphertext sizes: 699-1138 bytes
- Based on: Shortest Vector Problem in NTRU lattices

NTRU has longer history than ML-KEM but was not selected as NIST standard. Included for compatibility with systems deployed before standardization. Uses polynomial ring operations in $\mathbb{Z}_q[X]/(X^n * 1)$.

C.5 Key Derivation and Password-Based Cryptography

PBKDF2:

- Standard: RFC 8018
- Hash function: HMAC-SHA-256 or HMAC-SHA-512
- Iterations: Configurable, minimum 10000 recommended
- Salt: Random, minimum 128 bits
- Output: Variable length

PBKDF2 derives keys from passwords using iterated hashing. The iteration count slows down brute force attacks. Formula: $\text{DK} = T_1 \| T_2 \| ... \| T_\ell$ where $T_i = U_1 \oplus U_2 \oplus ... \oplus U_c$ and $U_j = \text{HMAC}(U_{j^*1})$ with $U_1 = \text{HMAC}(\text{password}, \text{salt} \| i)$.

HKDF:

- Standard: RFC 5869
- Hash function: SHA-256 or SHA-512
- Phases: Extract (PRK from input key material), Expand (derive keys from PRK)
- Output: Variable length up to 255 HashLen

HKDF provides key derivation suitable for Diffie-Hellman outputs and other key material. Extract phase: $\text{PRK} = \text{HMAC-Hash}(\text{salt}, \text{IKM})$. Expand phase: $T_i = \text{HMAC-Hash}(\text{PRK}, T_{i^*1} \| \text{info} \| i)$ where T_0 is empty string.

BIP32 Hierarchical Derivation:

- Standard: Bitcoin BIP 32
- Applications: HD wallets, key hierarchies
- Key types: Normal and hardened derivation
- Path notation: m/purpose'/coin_type'/account'/change/address_index

BIP32 enables deriving child keys from parent keys. Normal derivation allows public key derivation without private key. Hardened derivation requires private key. Master key derived from seed using HMAC-SHA-512. Child keys derived using HMAC-SHA-512 with parent key and index.

C.6 Encryption Schemes

AES-GCM:

- Standard: NIST SP 800-38D
- Key sizes: 128, 192, 256 bits

- IV size: 96 bits recommended
- Tag size: 128 bits
- Mode: Authenticated encryption with associated data (AEAD)

AES-GCM combines AES-CTR encryption with GMAC authentication. The mode provides both confidentiality and integrity. Implementation uses constant-time operations to resist side-channel attacks. Associated data is authenticated but not encrypted.

ECIES (Elliptic Curve Integrated Encryption Scheme):

- Components: ECDH key agreement, KDF, symmetric encryption, MAC
- Curves: P-256, secp256k1, Ed25519
- KDF: HKDF-SHA-256
- Encryption: AES-256-GCM
- MAC: Included in GCM tag

ECIES encrypts to recipient's public key. Sender generates ephemeral key pair, performs ECDH with recipient public key, derives encryption key using KDF, encrypts plaintext, includes ephemeral public key in ciphertext. Recipient uses private key with ephemeral public key to recover shared secret and decrypt.

C.7 Zero-Knowledge Proof Primitives

Schnorr Proof of Knowledge:

Proves knowledge of discrete logarithm x such that $Y = xG$ without revealing x.

Protocol:

① Prover generates random r, computes $R = rG$
② Prover computes challenge $c = H(G||Y||R)$
③ Prover computes response $s = r + cx$
④ Verifier checks $sG = R + cY$

Non-interactive variant uses Fiat-Shamir transform where challenge is hash of commitment and public values.

BBS+ Signatures with Selective Disclosure:

- Curve: BLS12-381
- Security: Based on q-SDH assumption
- Signature size: Single G1 element (48 bytes compressed)
- Proof size: Depends on number of attributes (approximately 300-500 bytes)

BBS+ signs multiple messages with single signature. Holder creates zero-knowledge proof revealing chosen messages while hiding others. Proof shows knowledge of signature on committed messages. Implementation follows IETF draft specification.

C.8 Random Number Generation

CSPRNG:

- Algorithm: HMAC-DRBG with SHA-512
- Standard: NIST SP 800-90A
- Seed size: 440 bits (security strength + entropy)
- Reseed interval: Never during single contract execution
- Prediction resistance: Provided by proper seeding

The CSPRNG state is initialized once per contract execution from provided seed or platform entropy. State is never shared between executions. Implementation includes instantiate, generate, and uninstantiate functions. State is wiped on VM termination.

C.9 Implementation Validation

All cryptographic implementations are validated through:

- Known Answer Tests (KAT) from standards documents
- Interoperability tests with other implementations
- NIST test vectors where available
- Academic paper test vectors for newer schemes
- Continuous integration test suite execution

Implementation compliance with standards is verified mechanically. Deviations from standards are documented and justified. Custom implementations include rationale and security analysis.

C.10 Algorithm Selection Guidelines

Organizations should select algorithms based on threat model and requirements:

For new systems:

- ECDSA or EdDSA for signatures (EdDSA preferred for simplicity)
- P-256 for FIPS compliance, secp256k1 for blockchain, Ed25519 for performance
- SHA-256 for hashing unless specific requirements dictate otherwise
- Consider ML-KEM and ML-DSA for long-term security

For blockchain integration:

- secp256k1 for Bitcoin/Ethereum compatibility
- Keccak-256 for Ethereum hashing
- BLS12-381 for aggregate signatures in consensus

For privacy-preserving credentials:

- BBS+ on BLS12-381 for selective disclosure

- Ed25519 for simple credentials without selective disclosure
- Coconut for threshold issuance requirements

For post-quantum transition:

- ML-KEM-768 for key encapsulation (hybrid with ECDH)
- ML-DSA-65 for signatures where size is acceptable
- Plan migration timeline aligned with organizational risk assessment

D Performance Benchmarks

E Security Checklist for Implementers

This appendix provides a practical checklist for organizations implementing systems using Zenroom. The checklist covers deployment configuration, operational security, contract design, integration patterns, and monitoring. Each item includes rationale and implementation guidance.

E.1 Pre-Deployment Planning

Threat model documentation:

- Document what threats the system must defend against
- Identify trust boundaries between components
- Classify data sensitivity and protection requirements
- Define adversary capabilities (insider, network, physical access)
- Map threats to Zenroom security features

Rationale: Zenroom provides specific protections. Understanding which threats you face determines whether those protections are sufficient.

Compliance requirements identification:

- List applicable regulations (GDPR, HIPAA, PCI-DSS, etc.)
- Document specific technical requirements from each regulation
- Identify gaps between Zenroom capabilities and requirements
- Plan application-layer controls for unmet requirements
- Obtain legal review of compliance approach

Rationale: Zenroom is infrastructure, not complete compliance solution. Knowing gaps allows planning.

Algorithm selection:

- Choose cryptographic algorithms matching threat model
- Consider post-quantum requirements for long-term data
- Plan migration timeline for algorithm deprecation
- Document algorithm choices and rationale
- Verify chosen algorithms are supported by required scenarios

Rationale: Algorithm choices affect security posture for years. Explicit decisions prevent defaults that may not fit requirements.

E.2 Build and Deployment Configuration

Binary verification:

- Verify cryptographic signatures on official releases
- Reproduce builds from source and compare to official binaries
- Document build toolchain versions used
- Establish trust in binary provenance before deployment
- Use checksums for distribution integrity verification

Rationale: Supply chain attacks target build and distribution. Verification catches compromised binaries.

Memory configuration:

- Set maximum memory pool size appropriate for platform
- Configure maximum iteration count for loops
- Set maximum nesting depth for control structures
- Test contracts execute within configured limits
- Document memory requirements for typical contracts

Rationale: Memory limits prevent resource exhaustion. Correct configuration allows legitimate use while blocking attacks.

Scenario loading:

- Load only scenarios needed for application
- Disable unused scenarios to minimize attack surface
- Verify scenario versions match tested versions
- Document which scenarios are required and why
- Test that contracts fail gracefully when scenarios are missing

Rationale: Each loaded scenario adds code that could contain vulnerabilities. Minimal loading reduces risk.

Seccomp configuration (Linux):

- Deploy seccomp profiles restricting system calls
- Verify only required system calls are permitted
- Test that contracts execute successfully under restrictions

- Monitor seccomp violations in logs
- Document allowed system calls and rationale

Rationale: Seccomp provides kernel-level enforcement of isolation. Even if VM is compromised, system calls are blocked.

E.3 Contract Design and Review

Input validation:

- Declare all input schemas explicitly in Given phase
- Use most restrictive schema that accepts valid input
- Validate string encodings (hex, base64) catch malformed input
- Test contract behavior with malformed input
- Document expected input structure and constraints

Rationale: Given phase is the recognition phase from LangSec. Validation here prevents malicious input from reaching processing.

Minimal permissions:

- Load only data needed for contract operation
- Avoid "Given I have all data" patterns that load everything
- Output only results needed by caller
- Do not print intermediate computation unless required
- Review each Given statement for necessity

Rationale: Principle of least privilege. Contracts with minimal access have minimal damage potential if exploited.

Error handling:

- Use If blocks to handle expected failures gracefully
- Provide meaningful error messages for debugging
- Avoid error messages that leak sensitive information
- Test error paths as thoroughly as success paths
- Document expected failure modes

Rationale: Unhandled errors cause contract failure. Proper handling provides better user experience and security.

Determinism verification:

- Test contracts produce identical output for identical input
- Test on multiple platforms (x86, ARM, different OS)
- Verify random number generation uses proper seeding
- Document any intended non-determinism
- Include determinism tests in contract test suite

Rationale: Non-determinism causes verification failures in distributed systems. Determinism is required for consensus and audit.

Code review:

- Review contracts for business logic correctness
- Verify cryptographic operations match requirements
- Check that schemas match actual data structures
- Ensure variable names are clear and not misleading
- Have contracts reviewed by non-authors

Rationale: Human review catches logic errors automated testing misses. Fresh eyes find issues authors miss.

E.4 Integration Security

Input sanitization:

- Validate data before passing to Zenroom
- Escape special characters in strings
- Verify JSON structure before contract execution
- Set reasonable size limits on input data
- Log rejected inputs for security monitoring

Rationale: Defense in depth. Application validation catches issues before they reach Zenroom.

Output validation:

- Parse Zenroom JSON output safely
- Validate output matches expected schema
- Handle execution errors appropriately
- Do not trust output encoding without verification
- Sanitize output before displaying to users

Rationale: Applications must validate external input, including from Zenroom. Trust but verify.

Process isolation:

- Run Zenroom in separate process, not embedded in-process
- Use minimal privileges for Zenroom process
- Limit process resources (CPU, memory, file descriptors)
- Terminate processes after contract execution completes
- Monitor process behavior for anomalies

Rationale: Process boundaries provide additional isolation. Compromised VM cannot directly access application memory.

Secret management:

- Do not pass secrets in logs or error messages
- Use secure channels for key material input

- Wipe sensitive data from memory after use
- Rotate keys according to security policy
- Document key lifecycle and storage

Rationale: Zenroom wipes its memory but applications must manage secrets properly in their context.

Network isolation:

- Deploy Zenroom on hosts without outbound internet access where possible
- Use network segmentation to isolate cryptographic operations
- Monitor network traffic for unexpected connections
- Block DNS resolution in container environments
- Document network architecture and isolation boundaries

Rationale: Zenroom makes no network calls but defense in depth requires network-level controls.

E.5 Operational Security

Logging and monitoring:

- Log contract execution start and completion
- Record execution duration for performance monitoring
- Log input hash (not content) for audit trail
- Alert on execution failures or timeouts
- Retain logs according to compliance requirements

Rationale: Logs provide audit trail and detect operational issues. Balance logging with privacy requirements.

Update procedures:

- Subscribe to Zenroom security announcements
- Test updates in staging before production deployment
- Maintain rollback capability for problematic updates
- Document installed version and update history
- Plan emergency update procedures for critical vulnerabilities

Rationale: Updates fix vulnerabilities but can introduce regressions. Procedures balance speed with safety.

Backup and recovery:

- Back up contract source code and configuration
- Store keys and credentials securely
- Document recovery procedures for system failure
- Test recovery procedures periodically
- Maintain offline backups for critical data

Rationale: Systems fail. Recovery procedures minimize downtime and data loss.

Incident response:

- Define security incident criteria
- Document response procedures and contacts
- Establish communication channels for incidents
- Plan forensic data collection procedures
- Conduct incident response exercises

Rationale: Incident response planning reduces chaos during actual incidents. Procedures enable effective response.

Performance monitoring:

- Baseline normal execution times for contracts
- Alert on significant performance degradation
- Monitor memory usage patterns
- Track error rates and failure modes
- Investigate anomalies promptly

Rationale: Performance changes may indicate attacks or system degradation. Early detection prevents issues.

E.6 Testing and Validation

Functional testing:

- Test all contract code paths including error cases
- Verify outputs match expected results
- Test boundary conditions and edge cases
- Include negative tests with invalid input
- Automate tests for regression detection

Rationale: Testing verifies contracts behave as intended. Automated tests catch regressions.

Security testing:

- Test with malformed input attempting to break parsing
- Test with extremely large input attempting resource exhaustion
- Test with input designed to trigger overflow or underflow
- Fuzz contracts with mutated inputs
- Document security test cases and results

Rationale: Adversarial testing finds vulnerabilities before attackers do.

Integration testing:

- Test complete workflow from application through Zenroom
- Verify error handling across integration boundary

- Test timeout and resource limit scenarios
- Validate data flow through entire system
- Test failure recovery mechanisms

Rationale: Integration bugs appear at component boundaries. Testing full workflows catches integration issues.

Performance testing:

- Measure execution time under normal load
- Test behavior under maximum load
- Verify memory usage stays within limits
- Test concurrent execution if applicable
- Document performance characteristics

Rationale: Performance testing identifies bottlenecks and validates capacity planning.

E.7 Documentation Requirements

System architecture:

- Document how Zenroom fits in overall architecture
- Diagram data flow through components
- Identify trust boundaries and security controls
- Document deployment topology
- Maintain architecture documentation current

Rationale: Architecture documentation enables security review and guides future changes.

Contract documentation:

- Document purpose of each contract
- Describe expected inputs and outputs
- Explain business logic in plain language
- Document cryptographic operations and algorithms
- Include examples of valid contract execution

Rationale: Contract documentation enables audit and helps future maintainers.

Operational runbooks:

- Document deployment procedures
- Describe monitoring and alerting
- Define troubleshooting procedures
- Document backup and recovery procedures
- Include emergency contact information

Rationale: Runbooks enable operations team to manage system effectively.

Security documentation:

- Document threat model and security requirements
- Describe implemented security controls
- Maintain inventory of cryptographic keys
- Document security testing results
- Track security incidents and resolutions

Rationale: Security documentation supports audit and compliance requirements.

E.8 Compliance Checklist

For organizations with specific compliance requirements:

GDPR compliance:

- Data processing purposes documented in contracts
- Retention periods enforced at application level
- Data minimization verified through contract review
- Subject rights (access, deletion) implemented
- Data protection impact assessment completed

FIPS 140 compliance:

- Only approved algorithms enabled
- Key sizes meet FIPS requirements
- Random number generation uses approved sources
- Self-tests implemented for critical functions
- Security policy documented

Common Criteria evaluation:

- Protection profile selected and documented
- Security functional requirements mapped
- Assurance level determined
- Evidence documentation prepared
- Evaluation facility engaged

This checklist is not exhaustive but covers critical security considerations. Organizations should adapt it to their specific requirements and risk tolerance.