Zencode DSL: human language smart contracts

Denis Roio[1]

Dyne.org foundation, Amsterdam,
jaromil@dyne.org,
WWW home page: https://dyne.org

Abstract. This paper presents a theoretical framework and a software solution to facilitate technological sovereignty and data commons management.
The goal is to improve people's awareness of how their data is processed by algorithms, as well facilitate developers to write applications that follow privacy by design principles.
The main requirement is that of distributed computing: Zencode must be capable of processing un/trusted code to execute advanced cryptographic functions, to be with any blockchain implementation as an interpreter of smart contracts.
Zencode is language to write portable scripts executed inside an isolated environment (the Zenroom VM) that can be ported to any platform, embedded in any language and made inter-operable with any blockchain.
The Zencode implementation is inspired by research on language-theoretical security, adopts Lua as direct-syntax parser to build a non-Turing complete domain-specific language (DSL) enforcing coarse-grained computations and recognition of data before processing.

Keywords: blockchain, language, smart-contracts, dsl, langsec

1 Introduction

Since DECODE project's inception, developing the Zencode language and releasing the Zenroom VM interpreter has been an extremely motivating ambition, as it concretely provides a solution for the techno-political implications illustrated by the AlgoSov.org observatory and researched in my Ph.D thesis on "Algorithmic Sovereignty".

I begin this paper illustrating the techno-political motivations for the development of Zenroom in the context of the DECODE project, an European H2020 grant (nr. 732546) coordinated by colleague Dr. Francesca Bria as its principal investigator.

I'll then proceed sharing my considerations on the state of the art of language design and security of execution in trust-less environments. The safe execution of untrusted code is required by most distributed ledger technologies (also commonly referred to as blockchain); it is as well a desirable feature for the reliability of cryptographic data manipulation for general use (certification, authentication and more advanced uses contemplated in Zenroom).

At last this paper consists a brief introduction of the Zencode DSL design and points to the Zenroom VM interpreter implementation to execute safely and efficiently simplified smart-rules describing cryptographic operation and data transformations using human readable language.

1.1 For the awareness of algorithms

The goal of the Zenroom VM and the Zencode language is ultimately that of realizing a simple, non-technical, human-readable language for smart-rules that are actually executed in a verifiable and provable manner within a controlled execution environment.

To articulate the importance of this quest and the relevance of the results presented, which I believe to be unique in the landscape of blockchain smart-contract languages, is important to remind us of the condition in which most people find themselves when participating in the regime of truth that is built by algorithms.

As the demand and production of well-connected vessels for the digital dimension has boomed, machine-readable code today functions as a literature informing the architecture in which human interactions happens and decisions are taken. The "telematic condition" is realized by an integrated data-work continuously engaging the observer as a participant. Such a "Gesamtdatenwerk" [1] may seem an abstract architecture, yet it can be deeply binding under legal, ethical and moral circumstances.

The comprehension of algorithms, the awareness of the way decisions are formulated, the implications of their execution, is not just a technical condition, but a political one, for which access to information cannot be just considered a feature, but a civil right [8]. It is important to understand this in relation to the "classical" application of algorithms executed in a centralized manner, but even more in relation to distributed computing scenarios posed by blockchain technologies, which theorize a future in which rules and contracts are executed irrevocably and without requiring any human agency.

The legal implications with regards to standing rights and liabilities are out of the scope here, while the focus is on ways humans, even when lacking technical literacy, can be made aware of what an algorithm does. Is it possible to establish the ground for a shared language that informs digital architects about their choices and inhabitants about the digital territory? Going past assumptions about the strong role algorithms have in governance and accountability [3], how can we inform digital citizens about their condition? When describing the virtualization of economic activity in the global context, Saskia Sassen describes the need we are observing as that of an analytical vocabulary:

> The third component in the new geography of power is the growing importance of electronic space. There is much to be said on this issue. Here, I can isolate one particular matter: the distinctive challenge that the virtualization of a growing number of economic activities presents not only to the existing state regulatory apparatus, but also to private-sector

> institutions increasingly dependent on the new technologies. Taken to its extreme, this may signal a control crisis in the making, one for which we lack an analytical vocabulary. [9]

The analysis of legal texts and regulations here shifts into an entirely new domain; it has to refer to conditions that only algorithms can help build or destroy. Thus, referring to this theoretical framework, the research and development of a free and open source language that is intelligible to humans becomes of crucial importance and, from an ethical standing point, DECODE as many other projects in the same space cannot be exempted from addressing it.

When we consider algorithms as contracts regulating relationships (between humans, between humans and nature and, nowadays more increasingly, between different contexts of nature itself) then we should adopt a representation that is close to how the human mind works and that is directly connected to the language adopted. Since algorithms are the systemic product of complex relationships between contracts and relevant choices made by standing actors [6], the ability to verify which algorithms are in place for a certain result to be visualized becomes very important and should be embedded in every application: to understand and communicate what algorithms and to describe and experiment their repercussions on reality.

For a deeper exploration of the techno-political implications raised by this document please refer to DECODE's blog-post on Algorithmic Sovereignty which also contains a series of historical examples of critical situations that help to understand the urgency we are facing.

DECODE goes in the direction of following a technical and scienifical restearch path and call for a new form of municipal rationality that contemplates technological sovereignty, citizen participation and ownership.

This narrative is echoing through world's biggest municipal administrations as we speak: a stance against the colonization of dense settlements by complex technical systems that are far from the reach of citizen's political control. The "Manifesto in favour of technological sovereignty and digital rights for cities" is now being considered as a standard guideline for ethics in governance by many cities of the world.

This whitepaper is then also a call for action to fellow programmers out there: we need to write code that is understandable by other humans and by animals, plants, all the living world we inoculate with our sensors and manipulate through automation. The term "smart" should really mean understandable, accessible, open and trustworthy [7]; then smart-contracts should be expressed in a language that most humans can understand. Good code is not what is skillfully crafted or most efficient, but what can be read by others, studied, changed, adapted.

Let's adopt intuitive name-spaces that can be easily matched with reality or simple metaphors, let's make sure that what we write is close to what we mean. Common understanding of algorithms is necessary, because their governance is an inter-disciplinary exercise and cannot be left in the hands of a technical elite.

2 Language Security

This section will establish the underpinnings of the Zencode language, starting from its most theoretical assumptions, to conclude with specific requirements. In order to do so, I will concentrate on the recent corpus developed by research on ”language-theoretic security” (LangSec). Here below we include a brief explanation condensed from the information material of the LangSec.org project hosted at IEEE. This research benefits from being informed by the experience of the exploit development community: exploitation is a practical exploration of the space of unanticipated state, its prevention or containment.

> In a nutshell [...] LangSec is the idea that many security issues can be avoided by applying a standard process to input processing and protocol design: the acceptable input to a program should be well-defined (i.e., via a grammar), as simple as possible (on the Chomsky scale of syntactic complexity), and fully validated before use (by a dedicated parser of appropriate but not excessive power in the Chomsky hierarchy of automata). [5]

LangSec is a design and programming philosophy that focuses on formally correct and verifiable input handling throughout all phases of the software development lifecycle. In doing so, it offers a practical method of assurance of software free from broad and currently dominant classes of bugs and vulnerabilities related to incorrect parsing and interpretation of messages between software components (packets, protocol messages, file formats, function parameters, etc.).

This design and programming paradigm begins with a description of valid inputs to a program as a formal language (such as a grammar). The purpose of such a disciplined specification is to cleanly separate the input-handling code and processing code. A LangSec-compliant design properly transforms input-handling code into a recognizer for the input language; this recognizer rejects non-conforming inputs and transforms conforming inputs to structured data (such as an object or a tree structure, ready for type- or value-based pattern matching). The processing code can then access the structured data (but not the raw inputs or parsers temporary data artifacts) under a set of assumptions regarding the accepted inputs that are enforced by the recognizer.

This approach leads to several advantages:

1. produce verifiable recognizers, free of typical classes of ad-hoc parsing bugs
2. produce verifiable, composable implementations of distributed systems that ensure equivalent parsing of messages by all components and eliminate exploitable differences in message interpretation by the elements of a distributed system
3. mitigate the common risks of ungoverned development by explicitly exposing the processing dependencies on the parsed input.

As a design philosophy, LangSec focuses on a particular choice of verification trade-offs: namely, correctness and computational equivalence of input processors.

2.1 Ad-hoc notions of input validity

Formal verification of input handlers is impossible without formal language-theoretic specification of their inputs, whether these inputs are packets, messages, protocol units, or file formats. Therefore, design of an input-handling program must start with such a formal specification. Once specified, the input language should be reduced to the least complex class requiring the least computational power to recognize. Considering the tendency of hand-coded programs to admit extra state and computation paths, computational power susceptible to crafted inputs should be minimized whenever possible. Whenever the input language is allowed to achieve Turing-complete power, input validation becomes undecidable; such situations should be avoided.

2.2 Parser differentials

Mutual misinterpretation between system components. Verifiable composition is impossible without the means of establishing parsing equivalence between different components of a distributed system. Different interpretation of messages or data streams by components breaks any assumptions that components adhere to a shared specification and so introduces inconsistent state and unanticipated computation [5]. In addition, it breaks any security schemes in which equivalent parsing of messages is a formal requirement, such as the contents of a certificate or of a signed message being interpreted identically, for example a X.509 Certificate Signing Request as seen by a Certificate Authority vs. the signed certificates as seen by the clients or signed app package contents as seen by the signature verifier versus the same content as seen by the installer (as in the recent Android Master Key bug [4]. An input language specification stronger than deterministic context-free makes the problem of establishing parser equivalence undecidable. Such input languages and systems whose trustworthiness is predicated on the component parser equivalence should be avoided. Logical programming using Prolog for instance, or languages like Scheme derived from LISP, or OCaml or Erlang would match then our requirements, but they aren't as usable as desired. As a partial solution to this problem the Zencode language parser (and all its components and eventually linked shared libraries) should be small, portable, self-contained and clearly versioned with a verifiable hash.

2.3 Mixing of input recognition and processing

Mixing of basic input validation ("sanity checks") and logically subsequent processing steps that belong only after the integrity of the entire message has been established makes validation hard or impossible. As a practical consequence, unanticipated reachable state exposed by such premature optimization explodes. This explosion makes principled analysis of the possible computation paths untenable. LangSec-style separation of the recognizer and processor code creates a natural partitioning that allows for simpler specification-based verification and management of code. In such designs, effective elimination of exploit-enabling implicit data flows can be achieved by simple systems memory isolation primitives.

2.4 Language specification drift

A common practice encouraged by rapid software development is the unconstrained addition of new features to software components and their corresponding reflection in input language specifications. Expressing complex ideas in hastily written code is a hallmark of such development practices. In essence, adding new input feature requirements to an already underspecified input language compounds the explosion of state and computational paths.

3 The Zencode Language

This section describes the salient implementation details of the Zencode DSL, the smart-rule language for DECODE, tailored on its use-cases and based on the Zenroom controlled execution environment (VM).

This section consists of three parts, each one explaining:

- the language model inherited from BDD / Cucumber
- the data validation model based on schema validation
- the memory model for safe computation

3.1 Syntax-Directed Translation

Lua is an interpreted, cross-platform, embeddable, performant and low-footprint language. Lua's popularity is on the rise in the last couple of years [2]. Simple design and efficient usage of resources combined with its performance make it attractive for production web applications, even to big organizations such as Wikipedia, CloudFlare and GitHub. In addition to this, Lua is one of the preferred choices for programming embedded and IoT devices. This context allows an assumption of a large and growing Lua codebase yet to be assessed. This growing Lua codebase could be potentially driving production servers and an extremely large number of devices, some perhaps with mission-critical function for example in automotive or home-automation domains.

Lua stability has been extensively tested through a number of public applications including the adoption by the gaming industry for untrusted language processing in "World of Warcraft" scripting. It is ideal for implementing an external DSL using C99 as a host language.

3.2 Behaviour Driven Development

In Behaviour Driven Development (BDD), the important role of software integration and unit tests is extended to serve both the purposes of designing the human-machine interaction flow (user journey in UX terms) and of laying down a common ground for interaction between designers and stakeholders. In this Agile software development methodology the software testing suite is based on natural language units that grant a common understanding for all participants and observers.

I'm very grateful to my friend and colleague Puria Nafisi Azizi for this brilliant intuition: adopting BDD for developing Zencode and implement a human-friendly language that does not depends on the underlying cryptographic implementation, allowing to share simple knowledge on how to include crypto scenarios components in different applications as well how to update them..

For our implementation of Zencode, definable as a dialect of BDD, the first step has been that of mapping series of interconnected cascading sentences of operations to the actual source code describing their execution to the Zenroom VM; this implementation has to be done manually with knowledge of Lua scripting and of the higher level functions that grant communication with the Zenroom VM.

Zencode then becomes a "textual frontend" that is easy to embed in graphical applications and whose purpose is to wire expressions and executions by means of utterances expressed in human language.

Referring to the Cucumber implementation of BDD, arguably the most popular in use by the industry to day and factual standard [10], the grammar of utterances is very simple and definable as a "cascading" flow indeed, since the fixed sequence of lines can follow only one fixed order:

"Given .. and* .. When .. and* .. Then print .."

This sequence is fixed and in simple terms consists of:

1. an read-only initialisation of states "Given (and)"
2. an read-write scenario based transformation of states "When (and)"
3. a write-only publishing phase of final states "Then (and)".

The Zenroom implementation simply defines fixed sequences of strings, mapping them to cryptographic functions, allowing the presence of variables that are expected to be arguments for the functions. These variables can then be changed by participants (frontend developers or application operators) as they are marked by inclusion a repeating sequence of two adjacent single quotes (' ').

The underlying parser is based on a finite state machine controlling the change of states and capable of executing security operations: data validation checks and memory wiping.

Zencode acts upon a positive, unique and non-flexible match of the first word of each new line, checks it complies with the current parser machine state and then proceeds parsing the whole phrase minus the variables, saving a pointer to the corresponding function if found along with the contents of variables if any.

As a result, one (or more, synonyms are supported) non-repeating line of parsed Zencode utterance is easy to translate across different spoken languages and corresponds to a declared function allowing the execution of Lua commands inside the Zenroom VM.

The current implementation addresses specific scenarios useful to the pilots in DECODE, while contemplating future extensions. Scenarios available:

- Simple symmetric transformations of cipher-text by means of HASH and KDF transformations
- Diffie-Hellman asymmetric key encryption (AES-GCM)
- Zero-Knowledge proof and blind-signing of credentials for unlinkable selective attribute revelations

Documentation, examples and an interactive online webassembly demo are all available online on the website dev.zenroom.org.

3.3 Declarative Schema Validation

In order to make the processing of Zencode more robust, all data used as input and output for its computations is validated according to predefined schemas. This makes the Zencode DSL a declarative language in which data recognition is operated before processing.

The data schemas are added on a per-usecase basis: they refer to specific cryptographic implementations as they are added in Zencode. Careful evaluation regarding their addition is made to realize if old schemas can be extended to include new requirements.

Schemas are expressed in a simple format using Lua scripting syntax and consist of:

- an importer from JSON data structures containing hex or base64 encoded complex data types
- an exporter of complex structured data types to big numbers encoded using hex or base64 and other common encoding formats

Every data structure processed in Zencode enters as a JSON or CBOR string input (IN), it is decoded and parsed, then checked for cryptographic validity (for instance checking point-on-curve) and stored in its validated data type (ACK) and at last is encoded back from defined data types to JSON or CBOR output string using encoding methods (OUT).

This creates three cascading sections in the HEAP of Zenroom and each section corresponds to the language steps in Zencode:

1. Given (IN)
2. When (ACK)
3. Then (OUT)

Providing a rigid structure to context-specific (or pilot-specific) implementations of Zencode scenarios: the parser should always operate data recognition in the Given/IN phase, operate transformations in the When/ACK phase and finally render output in the Then/OUT phase. This flow is locked with recurring HEAP checks to insure that different areas of memory are not accessed by the wrong section of Zencode, as well thatn the When/ACK phase is operated only on decoded memory and verified schemas.

4 Conclusion

This brief paper serves as an introduction to the motivation and design choices behind the Zencode DSL and, only partially, to the Zenroom VM. The production-ready implementation of Zencode should be seen as complementary to this paper and is publicly available under Affero General Public License v3 from the website Zenroom.org.

References

1. Roy Ascott. Planetary Technoetics: Art, Technology and Consciousness. *Leonardo*, 37(2):111–116, April 2004.
2. Andrei Costin. Lua code: Security overview and practical approaches to static analysis. 2017.
3. Nicholas Diakopoulos. Accountability in algorithmic decision making. *Commun. ACM*, 59(2):56–62, January 2016.
4. Jay Freeman. Exploit & Fix Android M̈aster Key"; Android Bug Superior to Master Key; Yet Another Android Master Key Bug. 2013.
5. Falcon Momot, Sergey Bratus, Sven M. Hallberg, and Meredith L. Patterson. The Seven Turrets of Babel: A Taxonomy of LangSec Errors and How to Expunge Them. pages 45–52, 2016.
6. Francesco Monico. Premesse per una costituzione ibrida.: La macchina, la bambina automatica e il bosco. *Aut/Aut, La condizione postumana*, 2014.
7. Caroline Irma Maria Nevejan et al. *Presence and the Design of Trust.* 2007.
8. A. Pelizza and S. Kuhlmann. Mining Governance Mechanisms. Innovation policy, practice and theory facing algorithmic decision-making. *Handbook of Cyber-Development, Cyber-Democracy, and Cyber-Defense*, 2017.
9. Saskia Sassen. *Losing Control? Sovereignty in an Age of Globalization.* Columbia University Press, 1996.
10. A Wynne. *The Cucumber Book: Behavior-Driven Development for Testers and Developers.* 2012.