The Zencode Whitepaper

Secure Crypto Language

by Denis Roio (Ph.D)

version 0.11, September 2019

The Zencode Whitepaper v0.11

Author: Denis Roio <jaromil@dyne.org>

This work has been funded by the European Union’s Horizon 2020 research and innovation program, grant nr. 732546 (DECODE).

The original source of distribution for this article, also providing its most up to date version, is the Internet website https://zenroom.org

Abstract

Zencode is a project inspired by the discourse on data commons and technological sovereignty. The established goal is that of improving people's awareness of how their data is processed by algorithms, as well facilitate the work of developers to create applications that follow privacy by design principles.

The main use case taken in consideration is that of distributed computing capable of processing untrusted code and executing advanced cryptographic functions, for instance it can be used with (but not limited to) any distributed ledger (blockchain) implementation as an interpreter of smart contracts.

The Zencode language makes it easy and less error-prone to write portable scripts implementing end-to-end encryption with operations executed inside an isolated environment (the Zenroom VM) that can be easily ported to any platform, embedded in any language and made inter-operable with any blockchain.

The Zencode implementation is heavily inspired by modern research in language-theoretical security, it adopts Lua as direct-syntax parser to build a non-Turing complete domain-specific language enforcing coarse-grained of computations and recognition of data before processing. Its interpreter, the Zenroom VM, supports secure isolation and protects its hosts from errors, it has no access to the calling process, the network, underlying operating system or filesystem.

Zenroom VM is a process virtual machine: a restricted execution environment designed to process safely any Zencode instruction, even when malicious. Upon any failure during phases of interpretation of code, validation of data or execution of operations Zenroom aborts returning meaningful error messages that help programmers assess what problem had occurred.

Zencode language scenarios are written following a declarative approach and provide functional tools to manipulate efficiently even complex data structures.

Table of Contents

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.eu observatory and researched in my Ph.D thesis on "Algorithmic Sovereignty".

I now begin this document 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.

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 the pars-construens of this whitepaper will describe how the Zencode language and the Zenroom VM interpreter have been implemented to execute safely and efficiently simplified smart-rules describing cryptographic operation and data transformations using human readable language.

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" (Ascott, 1990) 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 (Pelizza and Kuhlmann, 2017). 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 (Diakopoulos, 2016), 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.(Sassen, 1996)

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 (Monico, 2014), 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 (Nevejan and others, 2007); 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.

State of the art

In the DECODE project the main way to communicate between nodes is via a language rather than an API. All read and write operations affecting entitlements and accessing attributes can be expressed using the Zencode language whose ambition is to become a robust open standard for authorization of operations on personal data. The Zencode language will aim to naturally avoid complex constructions and define sets of transformations that can be then easily represented with visual metaphors.

To understand where this vision comes from, I'll proceed analyzing what exists already in practice and what are the theoretical approaches that can be followed in order to progress from a rather stalled state of the art mostly consisting in the adoption of heavily machine-based and error-prone, stateful and touring-complete scripting languages.

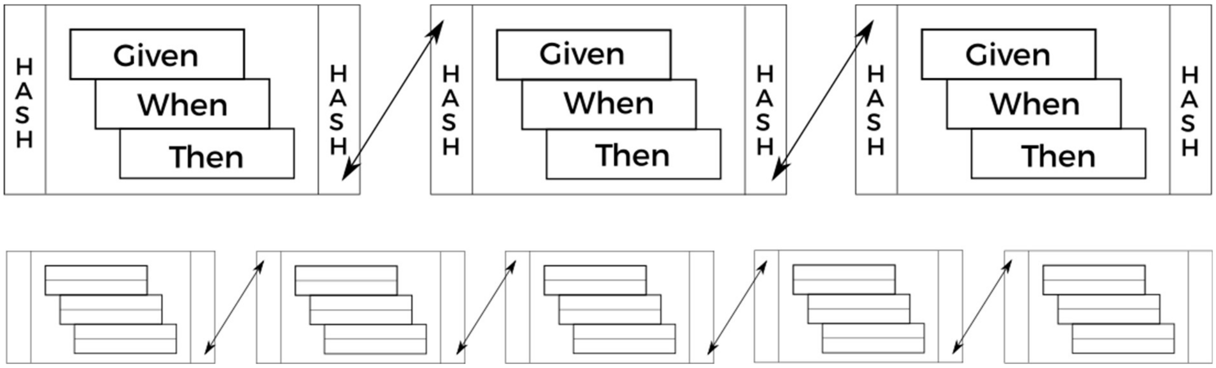
More considerations about the consensus algorithm of a DLT networks are intentionally left out of this document, since they are very specific issues concerning DLT implementations. Assuming an ideal condition where all network behavior is validated for being fully deterministic, I will use this whitepaper to focus exclusively on the task of language design for distributed computations.

Said that, I will now proceed to set the scenario and implications of Distributed Ledger Technologies (DLT) which in my opinion are establishing the notion of a new element in the classic way computer memory is organized and referred to.

A new memory model

In computing science the concepts of HEAP and STACK are well known and represent the different areas of memory in which a single computer can store code, address it while executing it and store data on which the code can read and write. With the advent of "virtual machines" (abstract computing machines like JVM or BEAM, not virtualised operating systems) the implementation of logic behind the HEAP and STACK became more abstract and not anymore bound to a specific hardware architecture, therefore leaving more space for the portability of code and creative memory management practices (like garbage collection). It is also thanks to the use of virtual machines that high level languages became closer to the way humans think, rather than the way machines work, benefitting creativity, awareness and auditability (McCartney, 2002). This is an important vector of innovation for the Zencode language implementation, since it is desirable for this project to implement a language that is close to the way humans think.

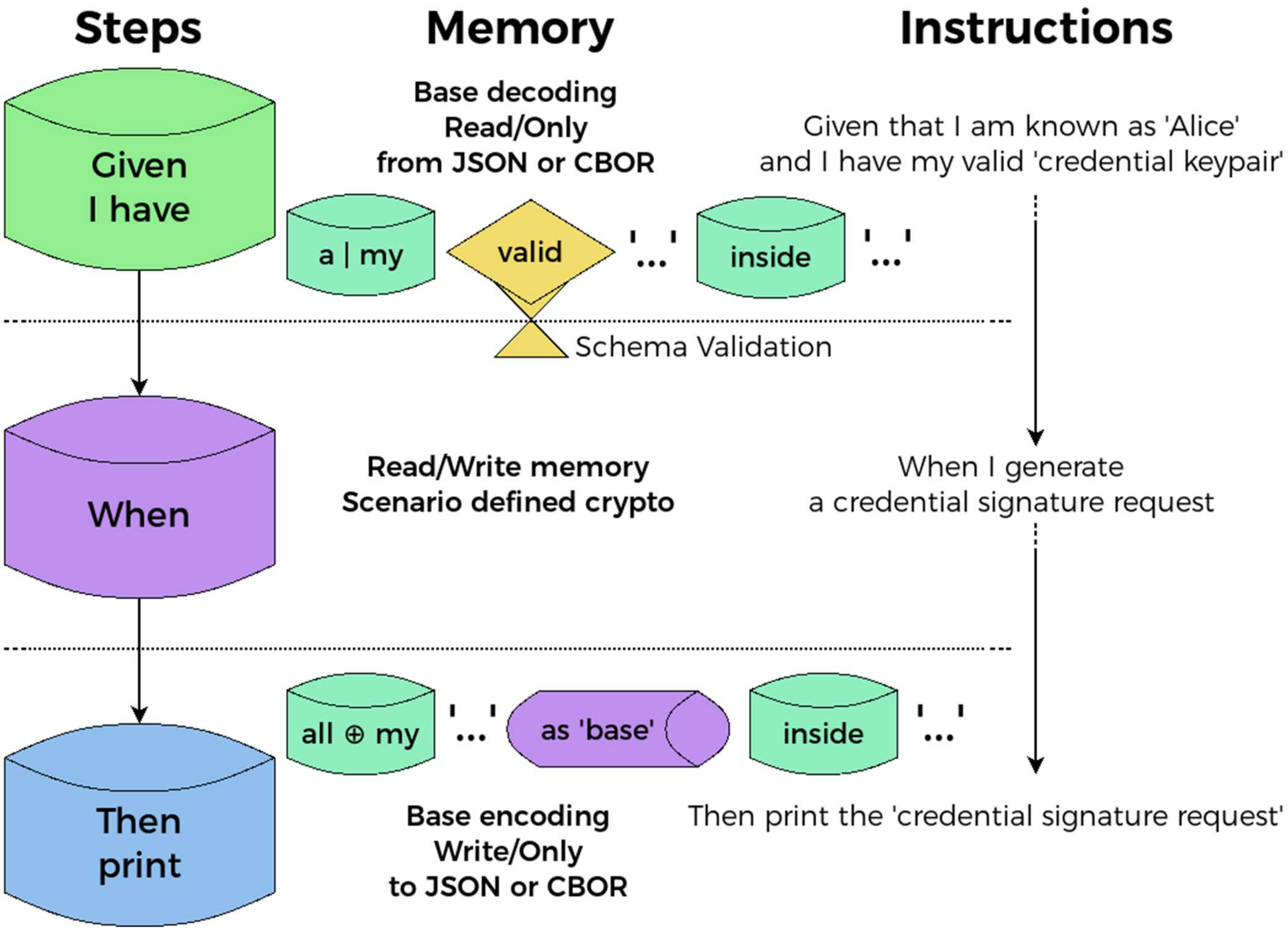
With the advent of distributed computing technology and blockchain implementations there is a growing necessity to conceive the HEAP and STACK differently (Pizka and Rehn, 2002), mostly because there are many more different conditions for memory bound to its persistence, read/write speed, mutability, distribution etc.



A "distributed ledger" (DLT) is a resilient and immutable memory space that can be addressed only by deterministic operations and can be written only when these operations lead to same results even when executed on very different machines. The DLT memory space is then a log of "signed events" whose authenticity can be verified by any node being part of the network.

Looking at the engineering of distributed systems from this new perspective brings the intuition that developing new ways to manage the HEAP memory space can be very beneficial when conscious about the implications of the distributed dimension of computation.

This is a how the Zenroom VM manages its memory:



More details of this figure will become clear as I'll proceed illustrating the way the Zencode language is structured after the model of a Behavior Driven Development language (BDD). What is relevant to say now is that the HEAP depicted above implies 3 sequential states (Given -> When -> Then) which are respectively read-only, read-write and write-only.

Blockchain languages

I'll engage a brief exploration of the main language implementations working on DLT. Far from being an exhaustive overview, this section highlights the characteristics of the two most prominent implementations and criticize the widespread approach to building virtual machines that are based on assembler-like operation codes and low-level languages.

The conclusion of this section is that DLT languages so far existing are designed with a product-oriented mindset, starting from the implementation of a virtual machine that can process OP_CODEs. Higher level languages build upon it, parsing higher level syntactics and semantics and compiling them into a series of OP_CODEs. This is the natural way most languages like ASM, C and C++ have evolved through the years.

Arguably, a task-oriented mindset should be assumed when re-designing a new DLT language: that would be the equivalent of a human-centered research and design process. The opportunity for innovating the field lies in abandoning the OP_CODE approach and instead build an External Domain Specific Language (Fowler, 2010) using an existing grammar to do the Syntax-Directed Translation. The Semantic Model can be then a coarse-grained implementation that can sync computations with blockchain-based deterministic conditionals.

Bitcoin's SCRIPT

Starting with the "SCRIPT" implementation in Bitcoin (Nakamoto, 2008) and ending with the Ethereum Virtual Machine implementation (Wood, 2014), it is clear that blockchain technologies were developed with the concept of "distributed computation" in mind. The scenario is that of a network of computers that, at any point in time, can execute the same code on a part of the distributed ledger and that execution would yield to the same results, making the computation completely deterministic.

The distributed computation is made by blockchain nodes that act as sort of "virtual machines" and process "operation codes" (OP_CODE) just like a computer does. These OP_CODES in fact resemble assembler language operations.

In Bitcoin the so called SCRIPT implementation had an unfinished number of "OP_CODE" commands (operation codes) at the time of its popularization and, around the 0.6 release, the feature was in large part deactivated to ensure the security of the network, since it was assessed by most developers involved that the Bitcoin implementation of SCRIPT was unfinished and represented threats to the network. Increasing the complexity of code that can be executed by nodes of an open network is always a risk, since code can contain arbitrary operations and commands that may lead to unpredictable results affecting both the single node and the whole network. The shortcomings of the SCRIPT in Bitcoin were partially addressed: its space for OP_RETURN (Roio et al., 2015) became the contested ground for payloads (Sward et al., 2017) that could be interpreted by other VMs, as well the limit was partially circumvented by moving more complex logic in touch with the Bitcoin blockchain (Aron, 2012), for instance using the techniques adopted by Mastercoin (Willett, 2013) and "sidechains" as Counterparty (Bocek and Stiller, 2018) or "pegged sidechains" (Back et al., 2014) implementations. All these are implementations of VMs that run in parallel to Bitcoin, can "peg" their results on the main Bitcoin blockchain and still execute more complex operations in another space, where tokens and conditions can be created and affect different memory spaces and distributed ledgers.

Languages implemented so far for this task are capable of executing single OP_CODEs: implementations are very much "machine-oriented" and focused on reproducing the behaviour of a turing-complete machine (Wegner et al., 2012) capable of executing generic computing tasks.

The Ethereum VM

The Ethereum Virtual Machine is arguably the most popular implementation of a language that can be computed by a distributed and decentralised network of virtual machines that have all their own HEAP and STACK, but all share the same immutable distributed ledger on which "global" values and the code (contracts) manipulating them can be inscribed and read from.

Computation in the EVM is done using a stack-based bytecode language that is like a cross between Bitcoin Script, traditional assembly and Lisp (the Lisp part being due to the recursive message-sending functionality). A program in EVM is a sequence of opcodes, like this:

```
PUSH1 0 CALLDATALOAD SLOAD NOT PUSH1 9 JUMPI STOP JUMPDEST PUSH1 32 CALLDATALOAD PUSH1 0 CALLDATALOAD
SSTORE
```

The purpose of this particular contract is to serve as a name registry; anyone can send a message containing 64 bytes of data, 32 for the key and 32 for the value. The contract checks if the key has already been registered in storage, and if it has not been then the contract registers the value at that key. The address of the new contract is deterministic and calculated on the sending address and the number of times that the sending account has made a transaction before.

The EVM is a simple stack-based architecture. The word size of the machine (and thus size of stack item) is 256-bit. This was chosen to fit a simple word-addressed byte array. The stack has a maximum size of 1024. The machine also has an independent storage model; this is similar in concept to the memory but rather than a byte array, it is a word- addressable word array. Unlike memory, which is volatile, storage is nonvolatile and is maintained as part of the system state. All locations in both storage and memory are well-defined initially as zero.

The machine does not follow the standard von Neumann architecture. Rather than storing program code in generally-accessible memory or storage, it is stored separately in a virtual ROM that can only be interacted with via a specific instruction. The machine can have exceptional execution for several reasons, including stack underflows and invalid instructions. Like the out-of-gas (OOG) exception, they do not leave state changes intact. Rather, the machine halts immediately and reports the issue to the execution agent (either the transaction processor or, recursively, the spawning execution environment) which will deal with it separately (Wood, 2014).

The resulting implementation consists of a list of OP_CODEs whose execution requires a "price" to be paid (Ethereum's currency for the purpose is called "gas"). This way an incentive is created for running nodes: a fee is paid to nodes for computing the contracts and confirming the outcomes of their execution. This feature technically defines the Ethereum VM as implementing an almost Turing-complete machine since its execution is conditioned by the availability of funds for computation. This approach relies on the fact that each operation is executed at a constant unit of speed.

On top of these OP_CODEs the "Solidity" language was developed as a high-level language that compiles to OP_CODE sequences. Solidity aims to make it easier for people to program "smart contracts". But it is arguable that the Solidity higher-level language, widely present in all Ethereum related literature, carries several problems: the shortcomings of its design can be indirectly related to some well-known disasters provoked by flaws in published contracts. To quickly summarise some flaws:

- there is no garbage collector nor manual memory management
- floating point numbers are not supported
- there are known security flaws in the compiler
- the syntax of loops and arrays is confusing
- every type is 256bits wide, including bytes
- there is no string manipulation support
- functions can return only statically sized arrays

To overcome the shortcomings and create some shared base of reliable implementations, programmers using Solidity currently adopt "standard" token implementation libraries with basic functions that are proven to be working reliably: known as ERC20, the standard is made for tokens to be supported across different wallets and to be reliable. Yet even with a recent update to a new version (ERC232) the typical code constructs that are known to be working are full of checks (assert calls) to insure the reliability of the calling code. For example, typical arithmetic operations need to be implemented in Solidity as:

```
function times(uint a, uint b) constant private returns (uint) {
  uint c = a * b;
  assert(a == 0 || c / a == b);
  return c;
}
function minus(uint a, uint b) constant private returns (uint) {
  assert(b <= a);
  return a – b;
}
function plus(uint a, uint b) constant private returns (uint) {
  uint c = a + b;
  assert(c>=a);
  return c;
}
```

It must be also noted that the EVM allows calling external contracts that can take over the control flow and make changes to data that the calling function wasn't expecting. This class of bug can take many forms and all of major bugs that led to the DAO's collapse (O'Hara, 2017) were bugs of this sort.

Despite the shortcomings, nowadays Solidity is widely used: it is the most used "blockchain language" supporting "smart-contracts" in the world.

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)." (Momot et al., 2016)

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.

Threats when developing a language

As one engages the task of developing a language there are four main threats to be identified, well described in LangSec literature:

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. For example, checking 'benignness' of arbitrary Javascript or even an HTML5+CSS page is a losing proposition.

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 (Momot et al., 2016). 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 (Freeman, 2013). 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 self-contained and clearly versioned and hashed and its hash verified before every computation.

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.

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.

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). Implementation details refer only to Zencode and not to how Zenroom is implemented, since the latter is already covered in other documents.

The implementation section contains three parts explaining:

- the language model inherited by Behaviour Driven Development
- the data validation model based on Schema Validation
- the implementation of implicit certificates

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 (Costin, 2017). 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 C or Python as a host language.

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: that of adopting BDD for develing Zencode and implement a human-friendly language to face the challenges posed by the pilots of the DECODE project. Among the diverse challenges we faced the recurrent need of implementing advanced and complex cryptographic schemes to be executed in a distributed manner across different nodes. For a large project involving different patterns in the technological delivery, a prominent difficulty was that of sharing components behaving in a consistent and deterministic way across different platforms and to share simple knowledge on how to include these components in different applications as well how to update their cryptographic operations.

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 (Wynne, 2012), 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 extendable initialisation of states "Given (and)"
2. followed by an extendable transformation of states "When (and)"
3. concluded by returning the 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, memory wiping etc.) here below the scheme of allowed state changes:

```
{ name = 'enter_given',   from = { 'scenario' },     to = 'given' },
{ name = 'enter_when',    from =   'given',          to = 'when' },
{ name = 'enter_then',    from = { 'given', 'when' }, to = 'then' },
{ name = 'enter_and',     from =   'given',          to = 'given' },
{ name = 'enter_and',     from =   'when',           to = 'when' },
{ name = 'enter_and',     from =   'then',           to = 'then' }
```

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.

Later, during the execution phase, Zenroom will take the collected pointers and execute them calling the functions and providing them as many arguments as the variables parsed. As a result, one (or more, synonyms are supported) non-repeating line of parsed Zencode utterance corresponds to a declared function allowing the execution of Lua commands inside the Zenroom VM.

Brief examples of this implementation follow:

```
Given("I introduce myself as ''", function(name) ACK[whoami] = name end)
Given("I am known as ''",         function(name) ACK[whoami] = name end)
```

The above definition of two lines possibly occurring within the utterances in Zencode are demonstrating how one can declare their own name by using one of the two different phrases, leading to a simple assignment of the variable *whoami* which will be available to the subsequent *When* prefixed Zencode block.

This simple demonstration is a hint to the fact that multiple patterns can be defined also in different ways, making the Zencode DSL implementation very easy to translate across different spoken languages as well contextualised within specific idiolects adopted by humans.

Furthermore, another example of implementation:

```
Given("that '' declares to be ''",function(who, decl)
        -- declaration
        if not declared then declared = decl
        else declared = declared .." and ".. decl end
        whois = who
end)
Given("declares also to be ''", function(decl)
        ZEN.assert(who ~= "", "The subject making the declaration is unknown")
        -- declaration
        if not declared then declared = decl
        else declared = declared .." and ".. decl end
end)
```

Shows how is possible to accept multiple variables and process them through more complex transformations that also contemplate the concatenation of contents to previous states. States are in fact permanent within the scope of the execution of a single utterance and will be modified in the same deterministic order by which they are expressed across lines. What is also visible within this example implementation, which we intend to facilitate by customization made by people who have a simple knowledge of Zenroom's API and LUA scripting, is that the 'ZEN.' namespace makes available a number of utility functions to easily check states (asserts) and propagate meaningful error messages that are then part of a backtrace output given to the calling application (host) on occurrence of an error.

The full implementation of Zencode available at the time of publishing this document is inside the source-code files 'zenroom/src/lua/zencode_*' and is relatively easy to maintain for the pilots analyzed in our project, as well easy to extend to more use-cases. The current implementation addresses specific schemes that useful to the pilots in DECODE, while contemplating future extension:

- Simple symmetric encryption of cipher-text by means of a PIN and KDF transformations (pilot: Amsterdam Register)
- Diffie-Hellman asymmetric key encryption (AES-GCM) (pilot: Making Sense IoT)
- Blind-sign credentials for unlinkable selective attribute revelations[1] (pilot: DECIDIM and Gebied Online)
- In addition there is also the implementation of an "implicit certificate" crypto scheme (Qu-Vanstone, ECQV) that is limited to first order curve transformations, which may apply to pilots requiring simple certification schemes[2].

All the implementations are illustrated in more detail in the following sections.

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 encoding

Every data structure processed in Zencode enters as a JSON string input (IN), it is decoded and parsed, then checked for cryptographic validity (for instance checking point-on-curve) and stored in its

[1] This implementation refers to work on the Coconut credential system (Sonnino et. al, 2018) designed after specific needs in DECODE's pilots. It does not implement, however, the threshold issuance part, which is only required in the scenario of a fully open blockchain implementation, which is still work in progress.

[2] It is important to note that while the ECQV scheme was not examined by other partners in our project, it has been choosen for its stable role in the industry and for its augmented complexity within an approachable implementation, complexity which could better inform the Zencode implementation.

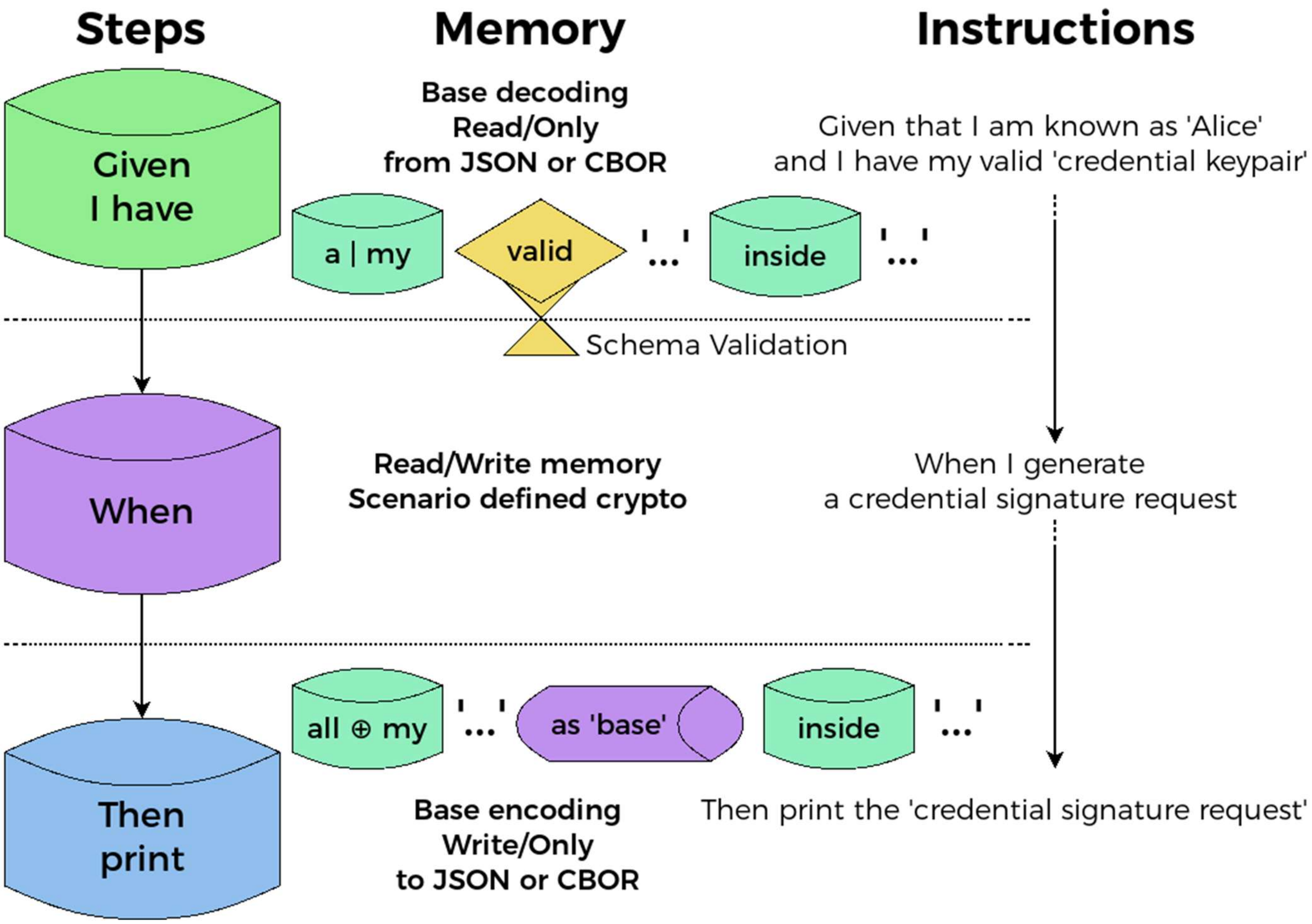
validated data type (ACK) and at last is encoded back from defined data types to JSON output string using encoding methods (OUT). This creates three cascading sections in the HEAP of Zenroom:

1. IN
2. ACK
3. OUT

Each of these sections correspond 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. Future plans include the lock-down of this flow with checks operated by Zenroom to insure that different areas of the HEAP are not accessed by the wrong section of Zencode scenario implementations.


Steps
Memory
Instructions
Base decoding
Read/Only
from JSON or CBOR
Given that I am known as 'Alice'
and I have my valid 'credential keypair'
Given
I have
a | my
valid
'...'
inside
'...'
Schema Validation
When
Read/Write memory
Scenario defined crypto
When I generate
a credential signature request
Then
print
all ⊕ my
'...'
as 'base'
inside
'...'
Base encoding
Write/Only
to JSON or CBOR
Then print the 'credential signature request'

Work In Progress

This page is left intentionally blank.

More details on the Zenroom implementation will be published in subsequent iterations of this whitepaper.

Bibliographic references

Aron, J., 2012. BitCoin software finds new life. New Sci. 213, 20.

Ascott, R., 1990. Is There Love in the Telematic Embrace? Art J. 49, 241.

Back, A., Corallo, M., Dashjr, L., Friedenbach, M., Maxwell, G., Miller, A., Poelstra, A., Timón, J., Wuille, P., 2014. Enabling blockchain innovations with pegged sidechains. URL Httpwww Opensciencereview Compapers123enablingblockchain-Innov.--Pegged-Sidechains.

Bocek, T., Stiller, B., 2018. Smart Contracts--Blockchains in the Wings, in: Digital Marketplaces Unleashed. Springer, pp. 169–184.

Costin, A., 2017. Lua code: security overview and practical approaches to static analysis.

Diakopoulos, N., 2016. Accountability in algorithmic decision making. Commun ACM 59, 56–62.

Fowler, M., 2010. Domain-specific languages. Pearson Education.

Freeman, J., 2013. Exploit & Fix Android˙Master Key"; Android Bug Superior to Master Key; Yet Another Android Master Key Bug.

McCartney, J., 2002. Rethinking the computer music language: SuperCollider. Comput. Music J. 26, 61–68.

Momot, F., Bratus, S., Hallberg, S.M., Patterson, M.L., 2016. The Seven Turrets of Babel: A Taxonomy of LangSec Errors and How to Expunge Them, in: IEEE Cybersecurity Development, SecDev 2016, Boston, MA, USA, November 3-4, 2016. pp. 45–52. https://doi.org/10.1109/SecDev.2016.019

Monico, F., 2014. Premesse per una costituzione ibrida.: la macchina, la bambina automatica e il bosco. AutAut Condizione Postumana.

Nakamoto, S., 2008. Bitcoin: A peer-to-peer electronic cash system. Consulted 1, 2012.

Nevejan, C.I.M., others, 2007. Presence and the Design of Trust.

O'Hara, K., 2017. Smart Contracts-Dumb Idea. IEEE Internet Comput. 21, 97–101.

Pelizza, A., Kuhlmann, S., 2017. Mining Governance Mechanisms. Innovation policy, practice and theory facing algorithmic decision-making. Handb. Cyber-Dev. Cyber-Democr. Cyber-Def.

Pizka, M., Rehn, C., 2002. Heaps and Stacks in Distributed Shared Memory, in: 16th International Parallel and Distributed Processing Symposium (IPDPS 2002), 15-19 April 2002, Fort Lauderdale, FL, USA, CD-ROM/Abstracts Proceedings. https://doi.org/10.1109/IPDPS.2002.1016494

Roio, D., Sachy, M., Lucarelli, S., Lietaer, B., Bria, F., 2015. Design of Social Digital Currency. -CENT.

Sassen, S., 1996. Losing Control? Sovereignty in an Age of Globalization. Columbia University Press.

Sward, A., OP_0, V., Stonedahl, F., 2017. Data Insertion in Bitcoin's Blockchain.

Wegner, P., Eberbach, E., Burgin, M., 2012. Computational Completeness of Interaction Machines and Turing Machines, in: Turing-100 - The Alan Turing Centenary, Manchester, UK, June 22-25, 2012. pp. 405–414.

Willett, J.R., 2013. MasterCoin Complete Specification.

Wood, G., 2014. Ethereum: A secure decentralised generalised transaction ledger. Ethereum Proj. Yellow Pap.

Wynne, A., 2012. The Cucumber Book: Behavior-Driven Development for Testers and Developers.

Acknowledgements

Denis Roio, also known as Jaromil, is co~founder of the Dyne.org think &do tank. Established in 1999 and based in Amsterdam since 2005, Dyne.org is a software house promoting the works of ethical hackers and free software developers. Dyne.org works to promote its 3 constitutional pillars: software freedom, environmental sustainability and interdisciplinary methodologies.

Jaromil received the Vilém Flusser Award at Transmediale (Berlin, 2009) while leading for 6 years the R&D department of the Netherlands Media art Institute (Montevideo/TBA), is an European Young Leader for the "40 under 40" program by FriendsOfEurope (2012) and is listed in the "Purpose Economy" list of top 100 social enterpreneurs in Europe (2014). Among recent projects he has been involved are: Bitcoin core, D-CENT (FP7/CAPS nr.610349), DECODE (H2020/ICT12 nr.732546), Devuan (GNU+Linux distribution) and Dowse (ISOC NL Innovation prize 2016 special mention).

(portrait courtesy of Robert Lloyd)

Zenroom and Zencode development has been funded by the European Union's Horizon 2020 Programme, under grant agreement number 7325346 (DECODE).

Dyne.org Foundation, Haparandadam 7-A1, 1013AK, Amsterdam
Chamber of Commerce registration number 34237525
For contact and enquiries please write to <info@dyne.org>