CYBERNETICA

Zero-Knowledge Proofs for mDL Authentication

technical report

Version 1.0

December 27th, 2022

D-2-525

Annotation

Mailing address:
Cybernetica AS
Mäealuse 2/1
12618 Tallinn
Estonia

Table of Contents

1 Introduction

In the Digital Identity Technologies Department of Cybernetica, work has been done to accommodate the mobile driver's licence technology (mDL) [1] to other areas such as fishing [2]. The application code is based on the code of example applications provided by Google [3].

Together with the Information Security Research Institute of Cybernetica, an attempt is being made to join mDL technology with zero-knowledge proofs. The mDL standard enables a person to convince others in having certain rights, but for that, the relevant data must be echtly delivered to the mDL verifier. The aim of current work is to demonstrate that, using available tools for zero-knowledge proofs, it is possible to reorganize the authentication procedure in such a way that the only piece of information the verifier would learn is the person having or not having the claimed right. This would also enable people to prove more interesting facts than those allowed by the mDL standard.

This document describes the results of the first phase of the work in this direction. It covers the core of the zero-knowledge proof part of a prospective protocol, namely, proving that medical diagnoses a person has do not disqualify him from army service, without actually revealing any of these diagnoses to the verifier. The mDL standard is modified for asserting diagnoses and that these have been decided by certified doctors. No communication between mDL technology and zero-knowledge proof engines is developed yet, and adaptations of mDL to the new needs are preliminary and subject to change.

In Sect. 2, a brief introduction to zero-knowledge proofs is given and a domain-specific programming language ZK-SecreC [4] being developed by the PROVENANCE team in the Information Security Research Institute is introduced. In Sect. 3, the medical check scenario assumed by the ongoing work is described in more details. In Sections 4–5, the architecture of the zero-knowledge proof for this scenario is described along with its current limits. Section 6 contains some statistics of performance and the output size and Sect. 7 concludes with outlining the planned next steps. To understand this document, basic knowledge of the mDL standard and the CBOR standard for data serialization [5] would be helpful.

2 Zero-Knowledge Proofs

Zero-knowledge proofs (ZKP) are protocols that allow one party to convince another party in knowing/owning data with a certain property, in such a way that nothing else about the data is revealed. The former party is called Prover, the latter party is called Verifier. The property being proved is known to Verifier, but even a successful ZKP should not help Verifier to find/compute data with this property themselves.

Proof protocols that feature zero knowledge can be found in everyday life. For example, a person who knows the password of their user name can prove having this knowledge to others by inviting them to the login screen and entering the user name and password. A standard login screen shows the user name while replacing all characters in the password with bullets. Hence, if the login is successful, the others will be convinced that the person indeed knows the password of that particular user name while learning nothing that could help them to find out this password (assuming, of course, that the others could not see which keys were pressed on keyboard). For another example, a person with normal vision (Prover) can prove a colour-blind person (Verifier) the additional ability as follows: Let them take two balls that are of different colour but otherwise indistinguishable; While Prover closes eyes, Verifier either interchanges the balls or not; Prover, having opened eyes again, can tell if the balls were interchanged or not. A distrustful Verifier might require repeating the act but, sooner or later, all they will be convinced. Verifier will not become able to distinguish the balls themselves, no matter of how many rounds of proof have been performed.

ZKP algorithms for computers have been studied by cryptographers since 1985. ZKP in computers must usually rely on an abstract machine that is trusted by both parties and can thus compute with data known to Prover, as well as that known to Verifier. A standard choice is a *modular arithmetic circuit* which can do integer addition and multiplication modulo a fixed constant and test if a value is zero. The only outcome of a computation in a circuit is a bit that tells if all tests succeeded or not. A circuit reveals no information about the input values or intermediate results. Every circuit performs ZKP of a particular property. As the property itself, the corresponding circuit is known to both parties (unlike the data that the circuit computes with).

It would be extremely tedious and error-prone to encode all practical ZKP acts directly as arithmetic circuits. Therefore, researchers have seeked for ways to specify ZKP procedures in higher-level languages. The PROVENANCE team of the Information Security Research Institute is developing ZK-SecreC, a domain-specific programming language designed for specifying ZKP acts in the form of high-level program code. Its syntax resembles that of Rust [6] and its compiler translates the high-level code to SIEVE IR that is a standard under development as a joint work of several ZKP research teams of the world for expressing modular arithmetic circuits for ZKP in both text and binary format.

In ZK-SecreC, leaking of Prover's data to Verifier is excluded by its strong static type system. Such potentially dangerous leak would be discovered during compilation and the program would never be executed. ZK-SecreC distinguishes one more domain — Public (besides Prover and Verifier) — which contains constants whose values are available to the compiler. Values in neither of the other two domains are let to influence values in Public. In practice, constants in Public domain typically represent lengths of lists of values in other domains or their upper bounds. Branching constructs and loops are not available in circuits since they would alter the duration of computation and this could reveal information about the input. Hence loops performed in the

circuit must be unrolled by the compiler and, to be able to do it, the number of repetitions of the loop body must be in Public. It is up to the algorithm designer if the exact bounds of loops are assumed to be Public data or some upper bounds are used which would assure better secrecy while increasing complexity of the algorithm.

The type system of ZK-SecreC also distinguishes two *stages*, pre and post. Only the latter stage contains computations in circuit. In the former stage, the programmer can specify computations that the parties (Prover and Verifier) should perform in order to help the circuit in its task. (As the set of available operations in the circuit is very small, not all necessary checks can be done using circuit operations only. Therefore interaction between the circuit and the parties might be necessary during ZKP. Of course, all data provided by Prover must be eventually verified in the circuit.)

ZK-SecreC supports parametric polymorphism that allows programmers to avoid code duplication if the same functionality has to be implemented for all domains or both stages.

In a longer perspective, the aim of developing high-level tools for ZKP is to make the society able to bring ZKP into everyday interaction between people and authorities. This requires ability to build ZKP circuits for complicated relationships and running them for large data that occur in everyday life. Along with developing ZK-SecreC, the PROVENANCE team has tested its capability on several conceivable practical use cases such as financial auditing (the bank account owner wants to prove their income being in conformance with some requirements, without having to reveal all details of incomes), electronic vehicle subsidies (the electronic vehicle owner wants to prove being eligible for subsidies offered if the total driving distance within a certain country has exceeded a certain amount, without revealing the actual data), etc.

3 Medical Check Scenario

3.1 Informal Description

Medical check scenario is historically the first of the practical use cases the PROVENANCE team implemented in zero knowledge using ZK-SecreC. Now, it is being reworked for integration with mDL technology.

In this scenario, a military recruit must receive credentials from a certain list of doctors about his diseases, to be delivered to the Military Entrance Processing Station (MEPS). In order to make a positive decision, MEPS must check the following:

- Do all the applicant's diseases enable him to serve in the army or not;
- Are all the doctors who provided the documents recognized by authorities or not;
- Are all the documents valid or not.

For the first check, MEPS has an official set of disqualifying diagnoses. None of the diagnoses of the recruit may occur there. For the second check, MEPS uses a list of certified health care providers. All issuers of the delivered documents must occur in that list. For the third check to be successful, all documents must belong to the particular applicant and none of them may be out-of-date.

As the recruit does not want the MEPS officials to know his particular diagnoses, ZKP is used for interaction between the recruit and MEPS, so that MEPS can learn no information beyond that it needs for making the decision. In this ZKP, Verifier's input contains information known to MEPS (lists of disqualifying diagnoses and certified doctors) and Prover's input contains information that the recruit wants to hide (the data of credentials issued by doctors).

The particular solution reckons with a possibility that the MEPS officials can make a positive decision even in the presence of disqualifying diseases on a case-by-case basis. For applying to an exception, a recruit must reveal all his disqualifying diagnoses. The revealed diagnoses are added to the Verifier's input and removed from the automatic checklist during the ZKP.

The original solution also included checking of signatures of the doctors which required dealing with public and private key pairs in zero knowledge. In the solution designed for being integrated with mDL technology, signature checking is assumed to happen outside zero knowledge, whence signature checking is currently omitted.

Diagnoses, both in the disqualifying list and in the credentials, are assumed to contain up to 7 characters. The small length enables performing occurs checks more efficiently by interpreting strings as integers with the same bit representation, since such integers are bounded by 2^{56} and do not overflow in the case of standard circuit moduli. Diagnosis codes in the International Classification of Diseases (ICD) fit into this limit, hence this assumption is realistic.

Whenever a diagnosis code is a prefix of another diagnosis code, the latter one is assumed to elaborate on the former one. The set of disqualifying diagnoses is given in the form of a list of code prefixes; every item in this list forbids all diagnosis codes extending it. So the ZKP must check not just equality of codes but a code being a prefix of another.

The solution does not address the recruit's chance to cheat by not delivering the credentials of disqualifying diagnoses to MEPS or just missing a medical examination. This could be addressed by other means, e.g., by asking all recognized health care providers to deliver the list of recruits

whom they have issued credentials that are still valid (without actually sending the credentials to MEPS). By analyzing these data, MEPS can find out if some recruit has missed a required medical examination or denying its results.

3.2 Requirements arising from mDL standard

To move towards conforming to mDL standard, the current solution assumes that documents issued by health care providers consist of a subset of record fields occurring in the mDL data format (namespace `org.iso.18013.5.1`) and one additional field for diagnoses which does not occur in mDL. All fields that are irrelevant in the medical check scenario are currently omitted, even if they are mandatory in mDL. More precisely, the following fields are assumed in our solution:

Identifier	Value	CDDL type
`family_name`	Last name of the recruit	tstr
`given_name`	First name of the recruit	tstr
`birth_date`	Birth date of the recruit	tdate
`issue_date`	Date of issue of the document	tdate
`expiry_date`	Date of expiry of the document	tdate
`issuing_authority`	Name of the health care provider	tstr
`diagnoses`	List of diseases discovered	[*tstr]

The last column refers to CDDL types [7]. Recall from CDDL that “uint” denote the type of unsigned integers, “bstr” and “tstr” denote the byte string and text string types, respectively, “tdate” denotes the type of dates in the YYYY-MM-DD format, and “[*tstr]” denotes the type of arrays of text strings.

Health care providers are assumed to pack their credentials into a mobile security object (MSO). The current solution does not support the MSO field `deviceKey` required in the standard. So, MSO in the current solution is of the following form:

Identifier	Value	CDDL type
`digestAlgorithm`	The algorithm used for finding digests	tstr
`docType`	Document type	tstr
`validityInfo`	Information about validity of the document	ValidityInfo
`valueDigests`	Digests of mDL fields in the issued document	ValueDigests

In the current solution, SHA-256 is used as the algorithm for finding digests and `org.iso.18013.5.1.mDL` is used as document type. The table refers to CDDL types “ValidityInfo” and “ValueDigests” that match those defined in the mDL standard; the definitions are recalled below.

Firstly, “ValidityInfo” is the type of maps of the following form:

Identifier	Value	CDDL type
signed	Date of signing the document	tdate
validFrom	The first day of validity of the document	tdate
validUntil	The last day of validity of the document	tdate

The value of validFrom of "ValidityInfo" must not be earlier than the value of issue_date of mDL. Likewise, the value of validUntil of "ValidityInfo" must not be later than the value of expiry_date of mDL.

Secondly,

ValueDigests = { nameSpaces: { +(tstr ⇒ { +(uint ⇒ bstr) }) } }

In words, a value of type "ValueDigest" is a map that takes the key nameSpaces to a new non-empty map that takes keys of text string type to non-empty maps that, in turn, take digest identifiers in the form of integers to digests in the form of byte strings.

To compute digests of fields of an mDL document, each field is first converted to a separate object of type "IssuerSignedItem" of the following form:

Identifier	Value	CDDL type
digestID	Number of the current field	uint
random	Random value ensuring uniform distribution of digests	bstr
elementIdentifier	The key of the encoded field	tstr
elementValue	The value in the encoded field	any

This object is represented in CBOR and the resulting byte string is hashed using the SHA-256 algorithm to obtain the digest of the field. In examples provided in [1], digest identifiers are consecutive integers starting from 1. Our solution does not rely on this convention but a fixed set of digest identifiers associated to mDL fields is assumed however.

All objects are ultimately represented in CBOR. The representation of each field is chosen according to the specified CDDL type. Fields that contain dates are represented as CBOR data of major type 6 and tag 18013 (after the number of the mDL standard), whereby the tagged part is of major type 3 (text string).

MEPS will obtain neither the credentials in mDL format nor the mobile security objects. Nevertheless, each health care provider reveals the hash of the CBOR representation of the mobile security object. This enables anyone who has access to a real mobile security object to ascertain its validity by checking if it is the original of the hash.

4 Parsing CBOR

The solution under development assumes offline retrieval mode of mDL. In this mode, data are expressed in the CBOR format. In order to conform to this, parsing data in the CBOR format in zero knowledge was implemented as a significant part of the solution. The implementation recognizes data of both integral and string types, arrays and maps, and also tagged dates. Other tagged data, floating-point numbers and simple data types without content are currently not recognized. Support for these types will be added during later phases if necessary. All data are required to be in definite-length form.

Zero knowledge does not enable transforming an input in the CBOR format to some delimited representation reflecting its semantic structure since the sizes of subtrees and the number of them are not necessarily known. Parsing data in the CBOR format therefore means representing them internally in a way that enables the user to extract their meaningful contents. For example, given an array in the CBOR format, one can lookup its elements via indices; if the element is itself a compound object then one can next query its consitutents, etc. All direct and indirect constituents are accessible via pointers that refer to starting points of these constituents in the whole input. Operations are defined to be able to, given the pointer, read data from the respective constituent.

Data in the CBOR format are internally represented in the form of the following ZK-SecreC struct:

```
pub struct Cbor[N : Nat, $S, @D]
{ config: CborConfig
, arrays: CborArrays[N, $S, @D]
, stores: CborStores[N, $S, @D]
}
```

Here `N`, `$S` and `@D` are type level parameters, denoting the modulus of the circuit (a natural number), a stage, and a domain (a stage always starts with dollar and a domain always starts with at). The value of the `config` field is a record in the following form:

```
pub struct CborConfig
{ total_len: uint $pre @public // the number of bytes
, val_len: uint $pre @public // the number of values encoded (nested included)
, max_strlen: uint $pre @public // maximal byte or text string length
, max_sublen: uint $pre @public // maximum number of items in array or pairs in map
, max_dep: uint $pre @public // maximal depth of nesting
}
```

The meaning of every field is described in code comments. The field `val_len` counts all values occurring in the CBOR representation, which coincides the number of initial bytes occurring throughout the CBOR representation. For example, consider a map consisting of 3 key-value pairs where keys are text strings “A”, “B”, “C” and the corresponding values are an array of integers 1, 2, 3, 4, 5, an array of integers 6, 7, 8, 9, 10 and an array of integers 11, 12, 13, 14, 15. The CBOR encoding is

```
A3 61 41 85 01 02 03 04 05 61 42 85 06 07 08 09 0A 61 43 85 0B 0C 0D 0E 0F
```

which contains 25 bytes, hence the value of `total_len` is 25. The value of `val_len` is 22 since the whole text encompasses 1 map, 3 strings, 3 arrays and 15 integers and $1 + 3 + 3 + 15 = 22$.

The value of the field `arrays` in struct `Cbor` has the form of the following ZK-SecreC struct:

```
struct CborArrays[N : Nat, $S, @D]
{ raw_data: list[uint[N] $S @D] // The original text followed by placeholder bytes
, val_ptrs: list[uint[N] $S @D] // Indices of initial bytes in the original text
, typs: list[uint[N] $S @D] // Types of data items
, vals: list[uint[N] $S @D] // Values in data items
, val_wids: list[uint[N] $S @D] // Byte numbers in values (initial byte included)
, up_ptrs: list[uint[N] $S @D] // Indices (in val_ptrs) of immediate parents
, arg_lsts: list[uint[N] $S @D] // Indices (in val_ptrs) of immediate children,
                          // grouped by the parent
, arg_bgns: list[uint[N] $S @D] // Indices (in arg_lsts) of the first children
, key_inds: list[uint[N] $S @D] // Flags (0/1) indicating if the data item is a key
, ..........................
}
```

(Dots stand for two additional fields that are not discussed in this document.) The values of all fields in this record are arrays of integers. The length of each array equals the value of the field `val_len` in the configuration record, except for fields `raw_data` and `arg_lsts`. The length of the array in the field `arg_lsts` is 1 less since the whole text is not a child of any other data item. Elements of the array in the field `raw_data` are bytes of the text under consideration, converted to integers. The length of this array is larger than the value of the field `total_len` in the configuration record because placeholder bytes are added to the end of the original text for achieving robustness of loops with an imprecise limit.

Elements in `val_ptrs` are ordered increasingly. For instance, in the case of the CBOR representation considered above, `val_ptrs` reads as

$$0, 1, 3, 4, 5, 6, 7, 8, 9, 11, 12, 13, 14, 15, 16, 17, 19, 20, 21, 22, 23, 24.$$

Numbers 2, 10 and 18 are skipped because bytes at these positions are not initial bytes of any data item (they are string bytes 'A', 'B', 'C'). Elements in array `typs` are CBOR types in the form of integers between 0 and 6 (7 is not allowed as data of that type are not supported). In the example considered, `typs` reads as

$$5, 3, 4, 0, 0, 0, 0, 0, 3, 4, 0, 0, 0, 0, 0, 3, 4, 0, 0, 0, 0, 0.$$

Elements in array `vals` are CBOR values occurring in the corresponding data item. In the case of types 0 and 1 (integers), the value coincides with the corresponding data item; in the case of types 2 and 3 (strings), the value equals the length of the string; and in the case of types 4 and 5 (compound structures), the value equals the length of the array or the map whose serialization starts at that point. In the same example as before, `vals` reads as

$$3, 1, 5, 1, 2, 3, 4, 5, 1, 5, 6, 7, 8, 9, 10, 1, 5, 11, 12, 13, 14, 15.$$

Elements in array `val_wids` store widths of the values, i.e., the number of bytes each value occupies. Only the value parts of data items are taken into account; in the case of strings for instance, bytes of the string are not counted. In our example case, `val_wids` reads as

$$1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1.$$

In the array of parent pointers, `up_ptrs`, parent of the root points to the root itself. In our example case, `up_ptrs` reads as

$$0, 0, 0, 2, 2, 2, 2, 2, 0, 0, 9, 9, 9, 9, 9, 0, 0, 16, 16, 16, 16, 16.$$

The array `arg_lsts` in the example case reads as

$$1, 2, 8, 9, 15, 16, 3, 4, 5, 6, 7, 10, 11, 12, 13, 14, 17, 18, 19, 20, 21,$$

i.e., the children of 0 are listed in increasing order, followed by children of 2 in increasing order, followed by children of 9 in increasing order, followed by children of 16 in increasing order. Other data items do not have children. The array `arg_bgns` consisting of the starting positions of corresponding children segments in `arg_lsts` reads as

$$0, 6, 6, 11, 11, 11, 11, 11, 11, 11, 16, 16, 16, 16, 16, 16, 16, 21, 21, 21, 21, 21.$$

That is, the children segment of a data item starts immediately after the previous segment ends, regardless of the data item having children or not. Finally, `key_inds` in the example case reads as

$$0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0.$$

The value of the last field `stores` in struct `Cbor` contains a similar set of fields with the same content as in `arrays` but in the form of stores. Stores emulate list-like data structures with access by index within circuit computations. Duplication of the data structures is useful for reducing time complexity of operations since reading arrays works faster than loading from stores but arrays can be accessed only if the index is in pre stage.

During initialization, the configuration record, as well as arrays and stores, are filled with correct data. Conformance of the input text to the CBOR standard and the extra restrictions imposed by the current solution is also checked during this phase. Tasks that required most of programming effort are the following:

- *Computing the contents of* `up_ptrs`. For that, relative depth of every data item is computed at first in pre stage using a straightforward recursive algorithm. Relative depth measures depth w.r.t. the previous data item; it can be 1 (if the current data item is the first in an array or a map or is a tagged item), 0 (if the current data item is a sibling of the previous one in the parse tree) or negative (if the previous data item was the last in an array or a map or a tagged item). Note that 1 is the only possible positive relative depth since every compound data structure starts with an initial byte marking down the base depth level of that data structure, while negative relative depths of any magnitude can in principle occur since no end markers are used. Secondly, entries of `up_ptrs` are computed in the circuit in the left-to-right order by firstly computing a chain of length `max_dep` of parent pointers starting from the current data item (the parent of 0 is defined to be 0) and then looking up the item at the relative depth in the chain. Thirdly, the entries of `up_ptrs` are checked in the circuit. This is needed since computing the entries make use of relative depths computed outside the circuit. It can be proven mathematically that, due to the way the entries are computed, it is enough to check correctness of the number of occurrences of every value as a parent pointer. The correct numbers of occurrences are available in the circuit as they are encoded in the CBOR input as values associated to data items of types 4 and 5 (in the case of supported tagged items, the number of occurrences must always be 1). Equality of the expected number of occurrences and their actual number is checked by introducing two polynomials whose roots are the parent pointers, taking into account their multiple occurrences (expected resp. actual), and checking that values that these polynomials produce equal at sufficiently many points to conclude that these polynomials are equal. The argument points are generated randomly in order to reduce the number of checks needed.
- *Computing the contents of* `arg_lsts`. The entries of `arg_lsts` are also computed in pre stage. The circuit then computes the array of parents of the corresponding data items using

`up_ptrs`. Finally, correctness of entries of both arrays are checked in the circuit. For that, the circuit checks that the entries in the array of parents are in non-decreasing order and, within any segment of equal parents, the corresponding entries of `arg_lsts` occur in strictly increasing order. This establishes non-repetition of child-parent pairs. Finally, it is checked that the first entry of `arg_lsts` is non-zero. Altogether, success of these checks implies that all the correct child-parent pairs and only these are present.

- *Checking uniqueness of keys of every map.* For that, the list of triples consisting of the being-a-key flag, the parent pointer and the value corresponding of the same data item is sorted lexicographically, using standard sorting techniques of zero knowledge. Thereby, the value of every data item of string type is replaced with the polynomial value at a fixed argument using bytes in the string as coefficients. Then the circuit checks that, for all data items that are keys of some map, it differs from the next in the lexicographic order item by either the parent pointer or the value. The argument of the polynomial is asked from Prover who has to choose it in such a way that all strings that occur as keys of the same map will be distinguished. In addition, it is checked that every map uses only keys of the same type. Therefore cases with a key of integral type and another key of string type of the same map having equal values are not allowed.

The library of CBOR parsing provides functions which, given a `Cbor` struct along with an index (of the array `val_ptrs`), can find the type, the value or the string contents of the data item at the given position of the given CBOR input. Other library functions can find the index (in `val_ptr`) of an element of an array or that of a value associated to a given key in a map at a given position of a given CBOR input. The element of the array is referred to by its serial number whereas keys of maps are referred to by their value (in the case of integral keys) or content (in the case of strings). Finally, there are functions that, given a `Cbor` struct and its index (in `val_ptr`), extracts the date that is encoded in the tagged item if the given index is that of a tagged date. Most of these functions are easy to define, only map lookups are somewhat complicated since the right value must be found out via multiplexing over all data items, out of which all except the one corresponding to the given key in the particular map must be eliminated. (The word 'multiplexing' refers to the way of performing an integer array lookup in circuit by creating another array of the same length containing 1 at position where the element in the original array must be read and 0 in every other position and then computing the scalar product of the original and the new array. In the case of map lookup, things are more complicated because of other maps with possibly the same keys occurring in the input text, keys being strings rather than integers etc.)

Most library functions have two versions, one that performs checks that the arguments form a well-defined query (e.g., that the given index indeed points to a data item of expected type) and another without these checks. Some functions also have different versions for being applied with arguments in different stages.

Here are the signatures of some of the CBOR library functions:

```
pub fn cbor_get_typ[N : Nat, $S, @D](ref cbor : Cbor[N, $S, @D], ptr : uint[N] $S @D) ->
    uint[N] $S @D where Finite[N]

pub fn cbor_get_val[N : Nat, $S, @D](ref cbor : Cbor[N, $S, @D], ptr : uint[N] $S @D) ->
    uint[N] $S @D where Finite[N]

pub fn cbor_get_str_pre[N : Nat, $S, @D](ref cbor : Cbor[N, $S, @D], idx : uint $pre
    @public) -> String[$S, @D, N] where Finite[N]

pub fn cbor_get_str[N : Nat, $S, @D](ref cbor : Cbor[N, $S, @D], ptr : uint[N] $S @D) ->
    String[$S, @D, N] where Finite[N]
```

```
8
9  pub fn cbor_lookup_array[N : Nat, $S, @D](ref cbor : Cbor[N, $S, @D], ptr : uint[N] $S
       @D, idx : uint[N] $S @D) -> uint[N] $S @D where Finite[N]
10
11 pub fn cbor_lookup_map_strkey[N : Nat, $S, @D](ref cbor : Cbor[N, $S, @D], ptr : uint[N]
       $S @D, str : list[uint[N] $S @D]) -> uint[N] $S @D where Finite[N]
12
13 pub fn cbor_lookup_map_intkey[N : Nat, $S, @D](ref cbor : Cbor[N, $S, @D], ptr : uint[N]
       $S @D, key : uint[N] $S @D) -> uint[N] $S @D where Finite[N]
14
15 pub fn cbor_lookup_tagged[N : Nat, $S, @D](ref cbor : Cbor[N, $S, @D], ptr : uint[N] $S
       @D) -> uint[N] $S @D where Finite[N]
16
17 pub fn cbor_get_date_with_check[N : Nat, $S, @D](ref cbor : Cbor[N, $S, @D], ptr :
       uint[N] $S @D) -> Date[$S, @D, N] where Finite[N]
```

5 Implementing Medical Check

5.1 Input data

ZK-SecreC assumes a separate input in every domain. In the ZKP terminology, Prover's and Verifier's inputs are called *witness* and *instance*, respectively. Input in the public domain, also allowed in ZK-SecreC, enables one to define constants that the ZK-SecreC compiler can use when translating the source code to circuit (instead of defining them directly in the ZK-SecreC source code).

Input for each domain must be located in a different file. The file names can be passed to the compiler as command line arguments. Each input file, regardless of the domain, must be in JSON format and consist of exactly one JSON object, values of which are strings or arrays of strings or arrays of arrays of strings etc. All strings occurring in values (unlike keys) must represent integers (keys can be any strings).

To conform to these restrictions, any integer input must be encoded as a string whose content is the integer, whereas string inputs must be encoded as lists of bytes, each byte being represented as a string whose content is the numeric value of the byte.

In the following, we specify CDDL types of inputs of all parties like we did in Subsect. 3.2 but note that now all data are ultimately represented in JSON (following the conventions described in the previous paragraphs) rather than CBOR.

In the current solution, Prover's input contains the credentials obtained by the recruit from health care providers and also their mobile security objects that contain fieldwise hashes of the credentials. More precisely, Prover's input contains the following two fields:

Identifier	Value	CDDL type
`hrs`	List of health reports	[+[+(bstr .cbor IssuerSignedItem)]]
`msos`	List of MSOs	[+(bstr .cbor MobileSecurityObject)]

The value of `hrs` is in the form of array of arrays. Each inner array contains credentials issued by one health care provider. Each element of an inner array is a CBOR encoding of an "IssuerSignedItem" object. The value of `msos` is an array of CBOR encodings of "MobileSecurityObject" objects, each corresponding to one health care provider. The arrays in fields `hrs` and `msos` contain the same number of elements, which equals the number of health care providers that have issued documents to the recruit, and the elements at the same position correspond to the same health care provider.

Verifier's input and public input do not use CBOR encoding. Verifier's input contains data known to MEPS, i.e., the list of disqualifying diagnoses, the list of recognized health care providers and the hashes of mobile security objects obtained from the health care providers that have issued credentials to the recruit. Verifier's input also includes data revealed by the recruit, i.e., the list of existing disqualifying diagnoses. More precisely, Verifier's input contains the following four fields:

Identifier	Value	CDDL type
bad_diagnoses	List of disqualifying diagnoses	[+tstr]
good_provider_npis	List of recognized health care providers	[+tstr]
mso_hashes	List of hashes of MSOs	[+bstr]
revealed_diagnoses	List of diagnoses revealed by the recruit	[+tstr]

Finally, public input contains technical parameters and upper bounds necessary for computation. The complete list follows:

Identifier	Value	CDDL type
dob	Date of birth of the recruit	tstr
family_name	Last name of the recruit	tstr
given_name	First name of the recruit	tstr
lens_msos	List of lengths of MSOs	[+uint]
lenss_hr_fields	List of lists of lengths of health report fields	[+[+uint]]
maxlen_diagnosis	Maximum length of a diagnosis	uint
maxlen_npi	Maximum length of name of health care provider	uint
num_bad_diagnoses	Number of disqualifying diagnoses	uint
num_good_providers	Number of recognized health care providers	uint
num_hr_fields	Number of health report fields	uint
num_hrs	Number of health reports the recruit has	uint
num_isi_fields	Number of fields in “IssuerSignedItem”	uint
num_mso_fields	Number of fields in “MobileSecurityObject”	uint
num_revealed_diagnoses	Number of diagnoses revealed by the recruit	uint
nums_diagnoses	List of numbers of diagnoses issuerwise	[+uint]
today	Today’s date	tstr

Patient’s date of birth and name are part of public input as MEPS has to know whose record they are considering. Alternatively, these data could be part of the instance (as it is not necessary that personal data were available to the compiler) but then the lengths of the first and last names would still have to be encoded as public input.

The value of `lens_msos` contains the lengths of CBOR representations of MSOs in Prover’s input field `msos`. Similarly, the value of `lenss_hr_fields` contains the lengths of CBOR representations of health report fields in Prover’s input field `hrs`, in the corresponding order and grouped issuerwise.

5.2 Computation

The following ZK-SecreC struct types are used for internal representation of the input data:

```
struct Person[$S, @D]
{ family_name: String[$S, @public, N61]
, given_name: String[$S, @public, N61]
, birth_date: Date[$S, @public, N61]
, revealed_diagnoses: list[String[$S, @D, N61]]
}

struct Restrictions[$S, @D]
{ bad_diagnoses: list[String[$S, @D, N61]]
, good_provider_npis: list[String[$S, @D, N61]]
, maxlen_diagnosis : uint $pre @public
, maxlen_npi : uint $pre @public
}

struct MDL[$S, @D]
{ family_name: String[$S, @D, N61]
, given_name: String[$S, @D, N61]
, birth_date: Date[$S, @D, N61]
, issue_date: Date[$S, @D, N61]
, expiry_date: Date[$S, @D, N61]
, issuing_authority: String[$S, @D, N61]
, diagnoses: list[String[$S, @D, N61]]
}

struct MSO[$S, @D]
{ signed: Date[$S, @D, N61]
, valid_from: Date[$S, @D, N61]
, valid_until: Date[$S, @D, N61]
, value_digests: list[list[u61 $S @D]]
}
```

Here, `String` and `Date` are struct types defined in ZK-SecreC standard library. Structs of type `String` consist of a byte array and string length, whereby the string length, as well as the bytes in the array, can be values in the circuit. The length of the byte array must be `$pre @public` which means that, in general, the number of bytes in the array is greater than the string length. In this case, bytes beyond the string length are considered as not belonging to the string. Dates represented as structs of type `Date` support comparison in the chronological order. Types `N61`, `u61` and `b61` are defined by

```
type N61 : Nat = 0x1FFFFFFFFFFFFFFF;
type u61 : Unqualified = uint[N61];
type b61 : Unqualified = bool[N61];
```

That means, `N61` is the Mersenne prime $2^{61} - 1$ and `u61` and `b61` are the integer and boolean types in circuit that computes modulo this prime.

Extracting data from public and Verifier's input is straightforward. Extracting data from Prover's input is done by functions `extract_mDLs` and `extract_msos` which must exploit the CBOR parsing library introduced in Sect. 4 and are therefore more complicated (especially `extract_mDLs` because values of different fields are located in different "IssuerSignedItems"). As the details are uninteresting, we present the signatures only:

```
fn extract_mDLs(input : list[list[list[u61 $post @prover]]], person : Person[$post,
    @verifier], nums_diagnoses : list[uint $pre @public], maxlen_npi : uint $pre @public,
    digest_ids : list[u61 $post @public]) -> list[MDL[$post, @prover]]
```

```
fn extract_msos(input : list[list[u61 $post @prover]], digest_ids : list[u61 $post
    @public], num_mso_fields : uint $pre @public, len_hash: uint $pre @public) ->
    list[MSO[$post, @prover]]
```

Three kinds of checks are performed. Firstly, it is checked that the credentials are issued to the right person and currently valid:

```
fn check_names_dates(person : Person[$post, @verifier], mDLs : list[MDL[$post, @prover]],
    mobile_security_objects : list[MSO[$post, @prover]], today : Date[$post, @public,
    N61])
{ let today_post = date_to_prover(today)
; for i in 0 .. length(mDLs)
  { string_assert_eq(string_to_prover(person.family_name), mDLs[i].family_name,
      person.family_name.len as $pre as uint)
  ; string_assert_eq(string_to_prover(person.given_name), mDLs[i].given_name,
      person.given_name.len as $pre as uint)
  ; date_assert_eq(date_to_prover(person.birth_date), mDLs[i].birth_date)
  ; assert(date_le(mDLs[i].issue_date, mobile_security_objects[i].valid_from))
  ; assert(date_le(mobile_security_objects[i].valid_from, today_post))
  ; assert(date_le(today_post, mobile_security_objects[i].valid_until))
  ; assert(date_le(mobile_security_objects[i].valid_until, mDLs[i].expiry_date))
  }
;
}
```

The code is mostly self-explaining. The library function `date_le` returns a boolean telling if the first argument date is before or equal to the second argument date.

Secondly, it is checked that all issuers of credentials are recognized health care providers and every disease discovered by any of the issuers is either confessed by the recruit or not disqualifying:

```
fn check_business_logic(person : Person[$post, @verifier], restrictions :
    Restrictions[$post, @verifier], mDLs : list[MDL[$post, @prover]], maxlen_diagnosis :
    uint $pre @public)
{ let issuing_authorities = for i in 0 .. length(mDLs) { mDLs[i].issuing_authority }
; let diagnoses = concat_non_rectangle(for i in 0 .. length(mDLs) { mDLs[i].diagnoses })
[...] // complicated computation details omitted from the report
; let is_bad_diagnosis : list[b61 $post @prover] = [...] // computation details omitted
; let is_revealed_diagnosis : list[b61 $post @prover] = [...] // computation details
    omitted
; let provider_validities : list[b61 $post @prover] = [...] // computation details omitted
; assert_true_cnt(is_revealed_diagnosis, length(person.revealed_diagnoses) * 2)
; for i in 0 .. length(diagnoses)
  { let is_revealed = is_revealed_diagnosis[length(person.revealed_diagnoses) + i]
  ; let is_bad = is_bad_diagnosis[length(restrictions.bad_diagnoses) + i]
  ; let is_valid = provider_validities[i]
  ; assert(is_valid)
  ; assert(is_revealed | !is_bad)
  }
;
}
```

The ZK-SecreC library function `concat_non_rectangle` concatenates a list of list, so `diagnoses` is a single list containing all diagnoses insisted by all health care providers. The lists `is_bad_diagnosis` and `is_revealed_diagnosis` of Booleans are computed as characteristic vectors that enable one to check by lookup if a certain diagnosis is bad or revealed or not. Due to the way these lists are

computed, they contain also values corresponding to the list of reference (i.e., the list whose membership is the criterion for deciding between true and false) in the beginning (so they start with a certain number of trues). The list `provider_validities` is similarly a characteristic vector but without the extra trues. The library function `assert_true_cnt` asserts that the number of true values in the given list equals the given number. In our case, success of this assertion implies that all diseases confessed by the recruit are indeed discovered by the health care providers (assuming that the list of diagnoses is without repetitions). Finally, the for loop computes, for every diagnosis, if it is revealed, if it is bad and if its issuer is valid, and asserts the required statements.

As a technical subtlety, the truth values in the characteristic vectors reflect *containing a prefix* in the reference list in the case of disqualifying diagnoses, in order to allow MEPS to specify whole families of diagnoses as disqualifying via including their common prefix in the list. In the case of revealed diagnoses and valid providers, characteristic vectors reflect *equality to an element* of the reference list.

Thirdly, it is checked that all hashes of "IssuerSignedItems" occurring in MSOs originate from the separate "IssuerSignedItems" in Prover's input and that the hashes of MSOs in Verifier's input originates from the MSOs in Prover's input. The following function is used for this task:

```
fn check_hash[@D](str : list[u61 $post @prover], hsh : list[u61 $post @D])
{ let n = length(str)
; let m = length(hsh)
; let bitss = bitextract_array(str, 8)
; let rev_bitss = for i in 0 .. n { reverse(bitss[i]) }
; let inter = sha256(concat(rev_bitss))
; let grouped_inter = group(inter, 8)
; let computed_hash = for i in 0 .. m { bits_to_uint(reverse(grouped_inter[i])) }
; for i in 0 .. m { assert_zero(computed_hash[i] - hsh[i] as @prover) }
;
}
```

The library function `bitextract_array` represents each element of an array in binary, in the form of list of bits. As the library function produces little-endian representations but the SHA256 algorithm assumes big-endian input, the order of bits in every byte must be reversed. The result of hashing is then grouped into bytes of 8 bits, the order is reversed in every byte and the library function `bits_to_uint` is applied in order to obtain integers. Finally, the result is compared to the expected hash value.

6 Performance

A performance test family was created with three varying parameters: the number of health reports, the number of diagnoses in each report, and the number of disqualifying diagnoses and recognized health care providers. The number of disqualifying diagnoses and recognized health care providers are not related in the problem statement but they were chosen to be equal in all our tests. Likewise, each health report contained the same number of diagnoses in our tests although the implemented solution works well also if the number of diagnoses in health reports varies. These decisions were made in order to reduce the number of variables.

A Haskell program was written that automatically generates necessary input files for any given triple of parameter values. Among other features, the program supports converting data structures to CBOR and to the restricted JSON with conventions described in Subsect. 5.1. For computing hashes, the program uses the non-standard `cryptohash-sha256` package available in Hackage.

The compiler was run on tests with 36 different triples of parameter values. In the following, we present the total numbers of gates in the resulting circuits. Different 3×3 tables correspond to different numbers of health reports which was 1 and 2 in the topmost row and 4 and 8 in the bottommost row. In each table, lines correspond to numbers of diagnoses in each health report (2, 4 and 8, resp.) while columns correspond to numbers of disqualifying diagnoses and recognized health care providers (10, 100 and 1000, resp.).

1	10	100	1000
2	4.48M	4.89M	9.01M
4	4.52M	4.93M	9.04M
8	4.60M	5.01M	9.27M

2	10	100	1000
2	8.92M	9.34M	13.5M
4	8.99M	9.40M	13.5M
8	9.29M	9.84M	13.7M

4	10	100	1000
2	17.8M	18.2M	22.3M
4	17.9M	18.3M	22.5M
8	18.7M	18.9M	22.9M

8	10	100	1000
2	35.6M	36.0M	40.1M
4	35.8M	36.2M	40.4M
8	37.2M	37.6M	41.7M

The table shows that the number of health reports has the largest among the parameters impact to the performance, whereby the number of gates depends nearly linarly on it. Both other parameters have a relatively small impact, although the number of disqualifying diagnoses and recognized health care providers has much larger variation within our tests.

The running time was moderate (approximately 17 sec. in the case of the test with the largest parameter values) whence we omit the exact figures.

7 Future Work

In the current state of the art, the solution described in this document enables one to test the ZKP part of a prospective software tool that joins mDL technology with zero knowledge. To use the current solution, input ready-to-use by ZK-SecreC compiler has to be prepared separately and the output must be interpreted using other means.

In the future, this part of the work must be integrated into real software that works with mDL technology, so that the ZKP part gets its input in real time and the output of ZK-SecreC compiler would be possibly executed by other technologies developed (not in Cybernetica) for running ZKPs, such as the EMP toolkit. It is likely that also the ZKP part will have to be enhanced considerably in order to conform to all specific requirements of standards that are used by mDL technology. Some problems are concerning the capacity of the tools for performing ZKPs that currently exist in the world. Standardization of the SIEVE IR language in which the modular arithmetic circuit output by the ZK-SecreC compiler is represented is a work in progress yet. The current version of its standard does not support Verifier challenges that is a very useful technique to create ZKPs and used also in our solution. This implies that running our solution with other ZKP tools may require Verifier challenges to be replaced with more inefficient alternatives.

The next step towards the aims would be creating a simple prototype that, at least to some extent, could run ZKP within the protocol used in mDL technology for authentication.

Bibliography

[1] ISO/IEC DIS 18013-5. *Personal identification — ISO-compliant driving licence — Part 5: Mobile driving licence (mDL) application*. 2021.

[2] *Tõend kui platvorm. Näidisrakenduse kirjeldus*. Cybernetica Dok: D-25-1. 2022.

[3] Google. *Identity Credential (github repository)*. url: https://github.com/google/mdl-ref-apps.

[4] D. Bogdanov et al. "ZK-SecreC: a Domain-Specific Language for Zero Knowledge Proofs". In: *CoRR* abs/2203.15448 (2022). doi: 10.48550/arXiv.2203.15448. arXiv: 2203.15448. url: https://doi.org/10.48550/arXiv.2203.15448.

[5] C. Bormann and P. Hoffmann. *Concise Binary Object Representation (CBOR)*. 2013. url: https://www.rfc-editor.org/rfc/rfc7049.

[6] S. Klabnik and C. Nichols. *The Rust Programming Language*. url: https://doc.rust-lang.org/book/.

[7] H. Birkholz, C. Vigano, and C. Bormann. *Concise Data Definition Language (CDDL): A notational convention to express Concise Binary Object Representations (CBOR) and JSON data structures*. 2019. url: https://www.rfc-editor.org/rfc/rfc8610.