zkGen: Policy-to-Circuit Transpiler

***Abstract*—Modern privacy-enhancing technologies reach new forms of computation and data privacy according to a statement (e.g. private value > 100). However, beyond the statement expression, the description of a private computation circuit requires knowledge of security algorithms protecting private values.**

To keep the description of private and compliant computation circuits as close to the statement expression as possible, we introduce a new composable policy language called zkPolicy. Further, we introduce a policy transpiler, called zkGen, to decouple the complexity expressed via zkPolicy from the complexity of the underlying security algorithms. Our results show that, with zkPolicy, the description of compliant data provenance circuits can be reduced from 957 to 22 lines of code. And, with zkGen, we automate the generation and composition of private computation circuits to a minimum effort of configuring a zkPolicy.

***Index Terms*—Transpiler Software, Policy Language, Verifiable Policy-compliant Computation, Zero-knowledge Proofs**

I. Introduction

Through Privacy-enhancing Technologies (PETs) such as Zero-knowledge Proof (ZKP) systems, data and computational privacy achieve new forms of compliance, where compliance holds against a public statement. The statement typically expresses single or multiple relations that hold on the arguments which are kept private during the privacy-preserving computation. Even though statements can be as simple as a comparison of two variables (e.g. bank balance > 10.000$), the security protocols keeping variables private might rely on complex cryptographic suites and many algorithms. For instance, to prove the data provenance of web traffic, current ZKP circuits prove unambiguous mappings between an Application Programming Interface (API) values and parameters of a Transport Layer Security (TLS) session and, with that, reach constraints in the range of millions [1]–[3]. Thus, creators of privacy-preserving computation circuits need domain-specific knowledge to link security algorithms to private variables and cannot solely specify circuits via a public statement.

With upcoming proofs of web interactions [4], configurable network policies [2], and verifiable credential applications [5], not only the efficiency requirements of PETs grow. More importantly, users become exposed to selecting or configuring public statements to express the desired interaction. As such, users implicitly become the creators of circuits. The custom selection and combination of statements introduces a new dynamic, which is currently not covered by frameworks that implement PET systems. Instead, current frameworks provide privacy-preserving subcircuits which are referred to as gadgets. Gadgets are, by default, not connected to any statements such that every circuit with compliance against a statement depends on a manual composition of gadgets to asserts the statement.

As a remedy, our work introduces a transpiler architecture to automatically compose statement-compliant circuits of privacy-preserving computations. Our transpiler, called *zkGen*, relies on a domain-specific policy to parse the desired public statements. For instance, to generate ZKP circuits, *zkGen* takes in a ZKP-specific policy written in a new *zkPolicy* language. With the *zkPolicy* language, we focus on a higher-level description of ZKP circuits at a statement level and rely on an abstraction layer, the gadget library, to connect statement arguments to the variables of security algorithms. Thus, taking as input a *zkPolicy* and the gadget library, *zkGen* outputs a constraint-based circuit description written in a Domain Specific Language (DSL) (cf. Figure 1). With that, any general-purpose ZKP framework supporting the DSL is able to process the constraint-based circuit and, if applicable, generate blockchain verification code (e.g. Solidity code for the Ethereum Virtual Machine (EVM)).

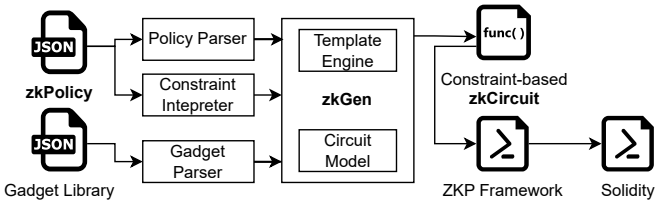

Fig. 1. Overview of the *zkGen* software architecture which, based on a gadget library, transpiles a policy-specific circuit description (e.g. *zkPolicy*) into a constraint-based representation of a ZKP circuit.

Hence, with *zkGen*, the automated and composable generation of compliant private computations takes a next step towards a simple description with less lines of code than constraint-based circuit descriptions. Further, circuits with outdated or insecure gadgets can be easily reconfigured by updating a single keyword in a *zkPolicy*. In short,

- We propose a policy language called *zkPolicy*, which expresses ZKP circuits via statements and gadget identifiers.
- We introduce a transpiler architecture to automatically generate and compose ZKP circuits according to a policy statement defined by the *zkPolicy* language.
- We open source the *zkGen* toolkit[1] with our gadget library and evaluate the utility of *zkGen* (cf. Section V).

II. Background

The background outlines the preliminaries of security algorithms, zero-knowledge proofs, and transpilers.

A. Zero-knowledge Proofs (ZKPs)

ZKPs became prominent by securing cryptocurrencies to remain private but publicly verifiable [6]. ZKP technology gives

[1]https://github.com/anonsubsub/zkGen

a prover the opportunity to convince a verifier of knowing a private input to a ZKP computation. The ZKP computation validates the private input against a public statement. Upon receiving a proof of a ZKP computation from the prover, the verifier validates the proof and learns nothing except, whether or not, the private input complies with the public statement. The field of verifiable credentials uses ZKPs to prove knowledge of signatures on data which is further asserted against public statements (e.g. age verification) [7]. Network policies rely on ZKPs to validate traffic compliance against block or allow lists [1], [2]. Data provenance protocols use ZKPs to validate integrity and non-ambiguity of TLS data [3], [8].

ZKP systems rely on separate data representations to achieve the desired functionality. For instance, ZKP systems translate a constraint-based description of a computation into a provable arithmetic encoding via a frontend stack [9]. The arithmetic representation can be evaluated by the respective backend stack of the ZKP system. As of today, different DSLs exist to describe the frontend, backend, or intermediate encodings of a computation circuit. Additional tooling exists to compile regular programs into ZKP circuits [10] or to provide interoperability between different ZKP encodings [11]. However, no tooling generates ZKP circuits at the abstraction level of policy compliance, which our work addresses.

B. Security Algorithms

In this work, we primarily focus on security algorithms which, expressed via ZKP circuits and combined with data assertions, achieve a compliance notion of confidential data. As such, we apply encryption algorithms, cryptographic commitments, digital signatures, or secure channel protocols to construct compliant and secure ZKP circuits. For instance, cryptographic commitments as a tuple of algorithms (*commit*, *open*) protect a data and witness pair under a commitment string with the *commit* algorithm. The *open* algorithm succeeds only if a non-ambiguous data and witness pair is validated against a commitment string. If the *open* algorithm is executed in a ZKP circuit and the witness and data pair are used as private variables, then a verifier, which evaluates a proof against a public commitment string, does not learn anything beyond the commitment validity.

C. Transpiler

Transpilers are source-to-source compilers which translate languages with comparable abstraction levels. The *zkGen* transpiler of this work translates a ZKP-specific policy language, called *zkPolicy*, into a constraint-based DSL of a ZKP system. With the ZKP system, the transpiler continues to compile and generate ZKP-specific parameters and code which can be used in the context of open decentralized systems.

III. zkPolicy Language

The *zkPolicy* language gives the opportunity to describe a ZKP circuit by connecting policy-relevant variables in a new data model (cf. Section III-A). The data model builds upon a *gadget library* abstraction layer which groups ZKP subcircuits according to security algorithms (cf. Section III-B).

```
{"name": "zkPolicy_tls13openSessionCommit",
  "relations":[{
    "value": {
      "type": "s-string",
      "size": 5,
      "protection": {
        "algorithm": "secure_channels:
            openRecord_TLS13AES128GCMSHA256",
        "parameter": "value"
      }
    },
    "number": {"type": "p-integer"}
  }],
  "constraints": [
    "0:value->-0:number",
    "0:value:protection:algorithm:key-=-
        commitments:mimc:message"
  ]
}
```

Fig. 2. JSON file expressing a ZKP circuit with the zkPolicy language.

A. Data Model

The *zkPolicy* language allows configurations based on the following data model, where

Relations are objects connecting two *argument*.
Arguments are key-value pairs, where keys express the name of an *argument*. Argument keys are unique in the scope of a relation. Argument values are objects with three key-value pairs; a *type*, a *size*, and a *protection*.
Types specify if the *argument* counts as secret or public by concatenating a "s-" or "p-" with an argument type t. Currently, *zkGen* supports two argument types with $t \in \{\text{string}, \text{integer}\}$.
Sizes specify the number of characters in an *argument* of type t=string and do not exist for *arguments* of type t=integer.
Protections use objects as values and rely on two key-value pairs to connect an *argument* to a security algorithm. To do so, *protections* point to a security algorithm of the *gadget library* via a string identifier "algorithm_type:algorithm_id" (cf. Figure 2). Further, *protections* map an *argument* to a security algorithm parameter via the "parameter" key.
Constraints are strings expressing assertions between elements such as *arguments*, *protections*, or both. Assertions rely on comparison operators (e.g. <,>,$\in$, etc) between two dashes. Multiple dashes express a logical OR between appended elements and the first element. Multiple constraints on the same element express a logical AND. The key words *if* and *for* follow a transpiler-specific syntax and are used to express conditions and loops. If *constraints* reference *protection* parameters, the *protection* string identifier is combined with the parameter name. If *constraints* reference *arguments*, the *relation* index is combined with the argument key or argument path towards a protection parameter.
zkObjects represent a complete *zkGen* policy configuration using the *zkPolicy* language (cf. Figure 2). The *zkObject* comprises three key-value pairs which define a list of *relations*, a list of *constraints*, and a name.

```
{"commitments": {
  "mimc": {
    "commit_string":
      {"type": "p-string", "size": 32 },
    "witness":
      {"type": "s-string", "size": -1},
    "message":
      {"type": "s-string", "size": -1}
}}}
```

Fig. 3. Simplified entry of the *gadget library* that defines the parameters of an mimc hash computation. The presented algorithm_type falls to the cryptographic commitments category and the algorithm_id is the keyword "mimc". Parameters of arbitrary size are indicated with a size=-1.

B. Gadget Library

The *gadget library* abstracts security algorithms to public and private parameters and uniquely defines algorithmic abstractions by grouping identifying names per algorithm under algorithm types (cf. Figure 3). Notice that the *zkGen* transpiler requires an implementation of every gadget in the DSL of the respective PET system. The *gadget library* currently supports different abstractions, where

Cryptographic commitments depend on a commitment string as the public parameter and a witness and a message as the private parameters (cf. Figure 3).

Encryption algorithms currently exist in two forms, where symmetric, asymmetric, and the one-time-pad encryption algorithms depend on a plaintext and a key as the private parameters and a ciphertext as the public parameter. *Encryption algorithms* in the counter mode require additional public parameters with an initialization vector and a block index.

Digital signatures are represented via two private parameters with the message and the signature value itself and a public key as the public parameter. In the context of anonymous credential systems [12], [13], proving knowledge of signatures is combined with a statement evaluation of the signed data.

Secure channels are constructed by the composition of an encryption gadget and a single or multiple commitment gadgets. In the context of data provenance proofs [1], [3], [8], [14], ZKP circuits allow users to validate web traffic secured by TLS against a statement (e.g. bank balance > 100$).

C. Compositions

The *zkPolicy* together with the *gadget library* allow the composition of multiple relations in a single policy, where relations are linked together by constraints. For example, credential chaining as introduced in the work [12] can be achieved by using two relations r_1, r_2 with arguments that refer to the respective credential values via the constraints:

1) "0:age->-0:number"
2) "1:age-<-1:number"
3) "0:age:p:a:commit_string-=-1:age:p:a:commit_string"
4) "0:age:p:a:witness-=-1:age:p:a:witness"

Here, the characters "p:a" shorten the "protection" and "algorithm" path keywords of the *zkPolicy* language and the indices 0,1 refer to the relation indices. Credential chaining is used to bind proofs towards multiple credentials.

IV. zkGen Transpiler

The following subsections introduce the components and capabilities of the *zkGen* transpiler as a command line toolkit.

A. Gadget Parser

Before a policy is parsed, the transpiler reads in the *gadget library* in form of a *library.json* file. In the next stage, the transpiler iterates over all security algorithms of the *gadget library* and searches for every gadget implementation. To successfully parse a gadget implementation, gadgets must be written under certain rules. Independent of the DSL a gadget is written in, *zkGen* relies on specific comments to facilitate the gadget parsing. Since gadget implementations can grow into many files and folders, a typical comment location is before the beginning and end of a circuit definition. If the gadget exists as a module, then comments must indicate the module name to enable seamless imports of other gadgets.

B. Policy Parser

The policy parser reads in policies as JavaScript Object Notation (JSON) files and iterates over the *relation* arguments to identify a list of *protection* algorithms per *argument*. The sequence of *protection* algorithms is arranged by the constraint interpreter (cf. Section IV-C) and used to identify gadget implementations for the composition of a circuit model (cf. Section IV-D). If an *argument* is of type string, the policy parser extends the list of *protection* algorithms at an *argument* with an extra gadget that converts the input argument to an aggregated integer representation. We present the string-to-integer conversion logic in the Formula 1, where the function $\texttt{ascii}(\text{str}_i)$ returns the American Standard Code for Information Interchange (ASCII) number of a string character and $len(\text{str})$ indicates the length of the string str.

$$a^{\text{aggr}} = \sum_{i=0}^{len(\text{str})} 10^i \cdot (48 - \texttt{ascii}(\text{str}_{len(\text{str})-i})) \qquad (1)$$

In the next stage, the parser module reads in and shares the list of *constraints* with the constraint interpreter module.

C. Constraint Interpreter

The constraint interpreter processes the list of *protection* algorithms and *constraints* and creates data attributes for the template engine. By iterating over the lists of the policy parser, the constraint interpreter removes duplicate references of the same security algorithm such that the circuit template instantiates every required security algorithm once. Any constraint duplicates per argument are removed as well. Next, the constraint interpreter builds a list of data attributes to encode the string-to-integer conversion per input argument. The second list of data attributes is used to instantiate and sequentially apply *protection* algorithms to arguments. The last list of data attributes encodes constraint checks between public and private arguments. The constraint interpreter throws errors

if private arguments potentially leak information (e.g. equality check of private argument against public argument).

D. Circuit Models & Template Engine

The output of the transpiler is generated based on a circuit model which *zkGen* maintains via a template engine. To start the composition of the circuit model, *zkGen* uses the template engine to add all string-to-integer conversion gadgets per input argument. Next, the required *protection* algorithms per argument augment the circuit model. To add the protection algorithms to the circuit model, *zkGen* relies on the parsed gadget implementations and the second list of data attributes of the constraint interpreter. If a security algorithm is called multiple times, then the circuit model resets all initialization variables of the algorithm. Last, the template engine adds individual constraints per argument to the circuit model.

In the end, *zkGen* generates circuits by executing the template engine on a circuit model. The outputs of the template engine are strings which are stored in unique output folders. Before a string is stored, *zkGen* determines the filename extension of output files based on the DSL the circuit is written in. The *zkGen* transpiler supports the generation of testing suites. However, to generate a test, the values of private and public arguments must be provided in a test configuration. The execution of a generated test suite depends on an installation of the desired ZKP system. If the ZKP system is installed, *zkGen* supports the generation of blockchain code (e.g. Solidity) that verifies the generated circuit.

V. Evaluation & Discussion

The evaluation summarizes the transpilation results and optimizations and compares *zkGen* against related works.

A. Transpilation Results

Our Golang implementation[2] of *zkGen* transpiles a *zkPolicy* to a policy-compliant ZKP circuit in 1.2 milliseconds. We rely on the Golang *text/template* package as the template engine and use the *gnark* ZKP framework [15] to (i) generate ZKP verification code in Solidity and to (ii) compute, verify, and test ZKPs. The *zkPolicy* language reduces the description complexity of a policy-compliant ZKP circuit to the definition of *relations* and *constraints*, and remains independent from algorithmic parameters. As such, the *zkPolicy* language reduces the description of a TLS oracle ZKP circuit from 957 to 22 Lines of Code (LOC). The description of a circuit to prove a privacy-preserving credential reduces from 85 to 20 LOC.

B. Related Works

The work *zkDocs* [5] introduces a transpiler toolkit to generate customizable privacy documents based on readable JSON schemas. Schemas of *zkDocs* let users define lists of fields, constraints, and trusted institutions, where fields are protected by commitments and asserted against defined constraints. Every constraint follows the syntax

$$<f_A><\text{SUB/ADD}><f_B><\text{LT/GT}><\text{const}/f_{Comp}> \quad (2)$$

[2]https://github.com/anonsubsub/zkGen

with fields f, the subtract and addition operators as SUB, ADD, and the comparison constraints larger than or greater then as LT, GT. The list of trusted institutions is used as an access control list to protect attestations on commitments. The generated circuit computes a fixed commitment check for each schema field and adds constraints, as additional assertions, on fields. In contrast to *zkDocs*, the *zkGen* can generate circuits with multiple distinct security algorithms and introduces a flexible *zkPolicy* language with enhanced expressiveness.

The work *DataCapsule* [16] introduces a flexible policy language called *PrivPolicy* with complete expressiveness. *PrivPolicy* extends the *Legalease* policy language [17] by using residual policies to describe the minimum and maximum privacy restrictions of processed data. With residual policies, *PrivPolicy* is applicable to programs which protect information leakage of private data processing through the concept of differential privacy [18]. In contrast to *zkGen*, *DataCapsule* statically analyzes programs to guarantee policy-compliant data processing but does not generate policy-compliant programs. However, since ZKP circuits do not support differential privacy [19], we deem *DataCapsule* as an interesting related work to investigate if differential privacy compliance could be added to the generation of ZKP circuits.

C. Limitations & Future Work

The first limitation of the *zkGen* toolkit is that it supports the generation of ZKP circuits only. With Multi-party Computation (MPC) or Homomorphic Encryption (HE) as other PETs, we deem the generation compliant MPC or HE circuits as future work. Secondly, as the *zkPolicy* language lacks a formal definition and is limited to a specific PET domain, we see the formal definition of the *zkPolicy* language and the definition of an domain-independent PET policy language as future work. The future goal of formalizing the *zkPolicy* language is to attribute generated PET circuits with sound privacy guarantees.

VI. Conclusion

This work automates the policy-compliant and composable generation of computational privacy via a transpiler architecture. Instead of manually configuring computational assertions with regard to a policy, we introduce a new policy language to describe and connect private computation variables to policy statements. The policy language delivers a concise description of compliant and private computation circuits with a fraction of code lines compared to constraint-based circuit descriptions. With a policy and a gadget library, the transpiler composes and executes circuit models via a template engine.

References

[1] P. Grubbs, A. Arun, Y. Zhang, J. Bonneau, and M. Walfish, "{Zero-Knowledge} middleboxes," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 4255–4272.

[2] C. Zhang, Z. DeStefano, A. Arun, J. Bonneau, P. Grubbs, and M. Walfish, "Zombie: Middleboxes that don't snoop," *Cryptology ePrint Archive*, 2023.

[3] J. Lauinger, J. Ernstberger, A. Finkenzeller, and S. Steinhorst, "Janus: Fast privacy-preserving data provenance for tls 1.3," *Cryptology ePrint Archive*, 2023.

[4] Chainlink, "Unlocking web3 identity: Blockchains, credentials, and oracles," https://blog.chain.link/web3-identity-blockchains-credentials-oracles/, November 2022.

[5] a16z crypto, "zkdocs: schema to snark circuit transpilation," https://github.com/a16z/zkdocs/tree/main/zkdocs-backend, December 2023.

[6] C. Baum, J. H.-y. Chiang, B. David, and T. K. Frederiksen, "Sok: Privacy-enhancing technologies in finance," *Cryptology ePrint Archive*, 2023.

[7] W. W. W. Consortium, "Verifiable credentials data model v2.0," https://www.w3.org/TR/vc-data-model-2.0/, 2023.

[8] F. Zhang, D. Maram, H. Malvai, S. Goldfeder, and A. Juels, "Deco: Liberating web data using decentralized oracles for tls," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 1919–1938.

[9] M. Bellés-Muñoz, M. Isabel, J. L. Muñoz-Tapia, A. Rubio, and J. Baylina, "Circom: A circuit description language for building zero-knowledge applications," *IEEE Transactions on Dependable and Secure Computing*, 2022.

[10] A. Ozdemir, F. Brown, and R. S. Wahby, "Circ: Compiler infrastructure for proof systems, software verification, and more," in *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2022, pp. 2248–2266.

[11] D. Benarroch, K. Gurkan, R. Kahat, A. Nicolas, and E. Tromer, "zkinterface, a standard tool for zero-knowledge interoperability," in *2nd ZKProof Workshop. https://docs. zkproof. org/pages/standards/acceptedworkshop2/proposal–zk-interop-zkinterface. pdf*, 2019.

[12] M. Rosenberg, J. White, C. Garman, and I. Miers, "zk-creds: Flexible anonymous credentials from zksnarks and existing identity infrastructure," in *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2023, pp. 790–808.

[13] D. Rathee, G. V. Policharla, T. Xie, R. Cottone, and D. Song, "Zebra: Anonymous credentials with practical on-chain verification and applications to kyc in defi," *Cryptology ePrint Archive*, 2022.

[14] "Pagesigner: One-click website auditing." https://old.tlsnotary.org/how_it_works, 2023.

[15] G. Botrel, T. Piellard, Y. E. Housni, I. Kubjas, and A. Tabaie, "Consensys/gnark: v0.9.0," Feb. 2023. [Online]. Available: https://doi.org/10.5281/zenodo.5819104

[16] L. Wang, J. P. Near, N. Somani, P. Gao, A. Low, D. Dao, and D. Song, "Data capsule: A new paradigm for automatic compliance with data privacy regulations," in *Heterogeneous Data Management, Polystores, and Analytics for Healthcare: VLDB 2019 Workshops, Poly and DMAH, Los Angeles, CA, USA, August 30, 2019, Revised Selected Papers 5*. Springer, 2019, pp. 3–23.

[17] S. Sen, S. Guha, A. Datta, S. K. Rajamani, J. Tsai, and J. M. Wing, "Bootstrapping privacy compliance in big data systems," in *2014 IEEE Symposium on Security and Privacy*. IEEE, 2014, pp. 327–342.

[18] F. McSherry and K. Talwar, "Mechanism design via differential privacy," in *48th Annual IEEE Symposium on Foundations of Computer Science (FOCS'07)*. IEEE, 2007, pp. 94–103.

[19] A. Narayan, A. Feldman, A. Papadimitriou, and A. Haeberlen, "Verifiable differential privacy," in *Proceedings of the Tenth European Conference on Computer Systems*, 2015, pp. 1–14.