2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER) | 978-1-6654-3786-8/22/$31.00 ©2022 IEEE | DOI: 10.1109/SANER53432.2022.00141

Stone: A Privacy Policy Enforcement System for Smart Contracts

Jihyeon Kim
Dept. of Computer Sci. and Eng.
Chungnam National Univ.
Dajeon, Korea
kim.jihyeon078@gmail.com

Dahyeon Jeong
Dept. of Computer Sci. and Eng.
Chungnam National Univ.
Dajeon, Korea
jeong960622@naver.com

Jisoo Kim
Dept. of Computer Sci. and Eng.
Chungnam National Univ.
Dajeon, Korea
cycyou78@gmail.com

Eun-Sun Cho
Dept. of Computer Sci. and Eng.
Chungnam National Univ.
Dajeon, Korea
eschough@cnu.ac.kr

***Abstract*—Smart contracts running on blockchain potentially disclose all data to the participants of the chain. Therefore, because privacy is important in many areas, smart contracts may not be considered a good option. To overcome this limitation, this paper introduces *Stone*, a privacy preservation system for smart contracts. With *Stone*, an arbitrary Solidity smart contract can be combined with a separate privacy policy in JSON, which prevents the storage data in the contract from being publicised. Because this approach is convenient for policy developers as well as smart contract programmers, we envision that this approach will be practically acceptable for real-world applications.**

Keywords—smart contracts, program analyses, privacy policies

I. Introduction

A smart contract is a self-executing contract in which the promise between a buyer and seller is written directly into lines of code. Because they are currently distributed and decentralized over the blockchain network in most cases, smart contracts and transactions made through the contracts cannot be traced or reversed.

In many cases, smart contracts deal with monetary data, where privacy is essential. However, running on the blockchain, smart contracts disclose all data to the participants of the chain. Thus, privacy concerns of smart contracts prevent the widespread usage of smart contracts in real-world applications.

This paper proposes *Stone*, through which an arbitrary Solidity smart contract can be combined with a separate privacy policy in JSON, which prevents the storage data in the contract from being publicised. This approach is convenient for both policy developers and smart contract programmers, and we envision that it will become practically applicable.

The remainder of this paper is organised as follows. The next section describes related studies. Section III overviews *Stone*, and Sections IV and V introduce the frontend and backend of the proposed system, respectively. Section VI presents the experimental results, and Section VII provides some concluding remarks regarding this research.

II. Related Work

Because a smart contract is a 'program', privacy protection technologies for programs will be effective for smart contracts. Traditionally, they protect variables in programs by assigning 'owners' and enforcing strict access control and delegation policies. Thus, tracing the usage of variables and identifying cases of information exposure are major efforts for privacy preservation for programs [1][2]. One of the most popular methods is using extended type systems to trace dataflows throughout the use of a program. If an illegal flow is detected, it throws a type error. However, the burden placed on programmers to learn a new type system or a new language is not inconsequential when the method is considered for real-world applications. In addition, reusing existing programs also becomes more complicated because the code becomes less understandable when merged with privacy policies. Moreover, one can easily imagine that privacy policies will vary more frequently than the programs themselves owing to outside circumstances, such as organisation principles and official rules. This implies that the lifetime of the type system differs from that of the program, which will burden programmers and reduce the productivity.

By contrast, privacy preservation efforts for blockchain data have also been made by researchers and practitioners. Zcash [3] (also known as Zero-Cash) allows encrypted versions of financial data on the chain. Although promising, they do not broaden the data domain for protection beyond limited monetary data. Owing to the overhead of encryption/decryption on blockchains, this is clearly understandable. However, this approach is inapplicable to smart contracts, which often handle various types of data.

A non-interactive zero-knowledge (NIZK) proof [4] based privacy preservation scheme has recently been applied to smart contracts. Zokrates [5] supports an off-chain execution facility for the part of smart contract logic that engages private data. For instance, when a smart contract checks if a consumer is over 19 years old, the consumer evaluates the query off-chain and then discloses the answer (i.e., true or false) on the chain. Verification of the correctness of the off-chain calculation is made from a zero-knowledge proof using zksnark [4]. Zokates provides a new programming language that is easier to learn than the zksnark interface. However, the burden of developers having to learn a new language and concept remains.

Merging privacy specifications into Solidity programs as a type system, zkay [6] has recently emerged. Through this approach, contract developers can avoid being bothered with

978-1-6654-3786-8/22/$31.00 ©2022 IEEE
DOI 10.1109/SANER53432.2022.00141

Authorized licensed use limited to: University of North Texas. Downloaded on January 19,2024 at 23:43:12 UTC from IEEE Xplore. Restrictions apply.

the boilerplate of a zero knowledge proof creation process through the well-designed complier provided by zkay.

This is extremely similar to a traditional scheme used to protect private variables in program languages, which makes zkay suffer from the same shortage of traditional schemes, as mentioned previously. Developers still need to learn the new type extensions of Solidity to use the zkay compiler, as well as consider the different lifetimes between the smart contracts and the privacy policy described in their types.

In this paper, to overcome these shortcomings, we propose *Stone*, which will enable developers to focus on creating smart contracts and write separate privacy policies in JSON format.

III. *STONE* OVERVIEW

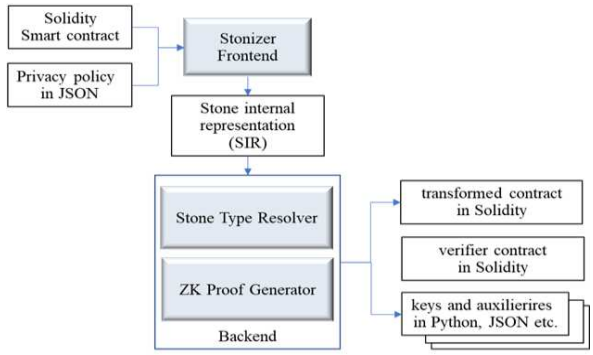

Figure 1. Overall process of *Stone*

As shown in Figure 1, *Stone* receives as input an arbitrary Solidity contract and a separate privacy policy in JSON format. First, the *Stone* frontend *Stonizer* combines the smart contract and privacy policy into a single smart contract in the *Stone Intermediate Representation (SIR)*. In addition to having a privacy-aware type system, the semantics of the *SIR* code are similar to that of Solidity. The *SIR* code is then delivered to the backend to yield a set of Solidity contracts with zero-knowledge proofs, keys for privacy preservation, an API operation code, and a metadata file.

To this end, the *Stone Type Resolver* then examines the *SIR* code to check whether private information has been leaked. Once it detects a violation of the given policy, a type error is issued, and the *Stone* backend process is terminated. Otherwise, the *Stone* backend confirms that the *SIR* code protects private information and delivers it to the *ZK Proof Generator*, which generates the expected results including proofs and other by-products. We use zkay v0.2 [7] as the proof generator, which implies that *Stone* can support a variety of zero-knowledge tools, such as jsnark and zksnark. (Note that the previous zkay was based on Zokrates.)

IV. FRONTEND OF *STONE*

A. JSON policy file

The *Stone* privacy policy in JSON has entries for the fields, constructors, and functions. For each field, each argument of the constructors, and each argument and variable of the functions, the JSON policy provides the attributes 'name', 'type', and other necessities. A type includes the basic Solidity types (e.g. `bool` or `uint`), the owner of the data, and the delegation information. A field may have a complex type mapping, which requires a bounded type variable led by a '`!`' symbol.

```
contract MedStats {
  final address hospital;
  uint count;
  mapping(address => bool) risk;
  constructor() {
     hospital = msg.sender;
     count = 0;
  }
  function record(address pat, bool r) public {
     require (hospital == msg.sender);
     risk[pat] = r;
     count = count + (r? 1:0);
  }
  function check(bool r) public {
     require(r == risk[msg.sender]);
} }
```

Listing 1 Example Solidity contract [1]

For instance, let us assume that a plain Solidity contract in Listing 1 is given to *Stone*, which was originally introduced in [6]. This is assumed to be controlled by a hospital. Information regarding the health of a blood donor is stored in the mapping `risk`, and the hospital should invoke the function `record`, allowing the risk status of the donor to be manually typed in and saved to the smart contract storage. As a result, it is stored in `risk[donor]`. Donors may call the `check` function to ensure that their health risk (e.g. `r`) that they are informed of is the same as the stored risk (e.g. `risk[msg.sender]`). The principle of information access in this example is as follows:

P1. Only the donors can access their own risk information.

P2. The initial input of risk information is made at the hospital because it is derived after an examination of the health information of the hospital donor.

P3. The risk level stored in `risk` by the hospital can only be viewed by the donor. No other player, even the hospital, can access the risk data of an individual.

Listing 2 shows P1 expressed as a JSON file with the entry of the mapping `risk`. The key to the mapping `risk` is the type `address` and is referred to as type variable `x`. The value of the mapping risk is a `bool` type, and the type variable `x`, that is, its own key, is designated as the owner.

```
"name" : "risk",
"type": {
    "k_type": {
        "t_type":"address",
        "quantifier": "!x"
    },
    "v_type": {
        "t_type":"bool",
        "owner": "@x"
    }
}
```

Listing 2. JSON policy for the mapping '`risk`'

The JSON policy for the function `record` is listed in Listing 3. The signature field of the JSON file serves as a unique ID representing each function. The second argument `r` has `me` as its owner. Here, `me` is a special keyword, meaning the caller of the function (that is, `msg.sender`), which should be the hospital in this case. In addition, `pat` in the `delegate_to` field implies that the access right will be delegated to `pat`, which is the first argument of this function. Note that the owner of `r` is a hospital (based on P2), and after it is stored in `risk`, it can be read by no other one than the donor who owns the information (based on P3); thus, an explicit delegation of the access rights is required.

Authorized licensed use limited to: University of North Texas. Downloaded on January 19,2024 at 23:43:12 UTC from IEEE Xplore. Restrictions apply.

```
"functions":[
    {
        "signature" : "6f0d85bb",
        "args" : [
            {
                "type": "address",
                "name": "pat"
            },
            {
                "type": "bool",
                "owner": "me",
                "delegate_to": "pat",
                "name": "r"
            }
```

Listing 3. JSON policy for the 'record' function

B. SIR and Stonizer

Such a JSON privacy policy is converted into a *SIR* along with the Solidity file, as shown in Fig. 1, and *Stone* checks to ensure that the conversion results do not violate the given privacy policy.

Because a *SIR* code is merged with a Solidity contract and JSON privacy policy, most of its syntax (shown in Fig. 2) and semantics are borrowed from the extended Solidity provided by zkay [6]. However, *SIR* is not an end-user language but an intermediate representation, and thus the complexity of *SIR* does not bother developers.

```
L ::= id | L[e] (Location) α ::= me | all | id
e ::= c | me | L | ⊖e | e1 ⊕ e2 | e1? e2 : e3 | pk(e) | ···
τ ::= bool | uint | address | bin
        | mapping(τ1 => τ2@α2) | mapping(address!id => τ @α)
P ::= skip | τ @α id | L = e | P1; P2 | require(e) | if e {P1 } else {P2 }
        | while e {P } | verifyφ (e0, e1, . . . , en )
F ::= function f (τ1@α1(>@β1)? id1, . . . , τn@αn (>@βn)? idn ) returns τ @α
        {P; return e; }
C ::= contract id{(final)? τ1@α1 id1; . . . (final)? τn@αn idn; F1 . . . Fm }
```

Figure 2. *SIR* grammar

Privacy policies are directly embedded in the program as types with an '@' mark, and during the type checking will block any access by others than the assigned owner. For instance, `uint@hospital count;` is the same as the declaration of the `uint`-typed variable `count` in the plain Solidity program, but indicates that the owner is a `hospital`. The part of the *SIR* code below shows that the second argument `r` of the function record in Listing 1 is owned by the caller but can be delegated to the first argument `pat` during the function execution. This delegation notation makes the biggest difference between SIR syntax and zkay syntax.

```
function record(address pat, bool@me>@pat r) public {…
```

Similarly, `mapping(address!x => bool@x) risk`, which is translated from `mapping(address=>bool) risk` in Listing 1 together with the policy Listing 2, denotes that the key is the owner of the value of each entry of this mapping.

Transforming JSON privacy policies into type notations is a major task of the *Stonizer*. It entails parsing the Solidity program and comparing the contents with the JSON policy. First, the Solidity functions, whose type and name match the JSON policy, are examined in turn. Then, ownership and delegation information are inserted into to the *SIR*. For instance, the `owner` field is encoded into the *SIR* with a type prefixed with an '@'. Delegation specifications are inserted by attaching a '>' symbol after the ownership specification. A straightforward redundancy elimination is also made during this process. For instance, *Stonizer* ignores the `delegate_to` field that has the same value as the `owner`.

C. Stone Type Resolver

The *Stone Type Resolver* considers all complex expressions, assignment statements, and data flows between variables to see if there are any policy violations at the compilation time. Listing 4 shows the type system of *SIR*, which is mainly borrowed from zkay [6]. For instance, if an expression, e.g. `r+1`, is determined based on type @x and will be delegated to @y, it will be described as `r+1: unit@x>@y`.

$$\text{(i)}\ \frac{x : \tau@\alpha \in \Gamma}{\Gamma \vdash x : \tau@\alpha} \qquad \text{(ii)}\ \frac{\Gamma \vdash L\text{: } \mathbf{mapping}(\tau \Rightarrow \tau'@\alpha\,)@\mathbf{all}\ \ \Gamma \vdash e\text{:}\tau@\mathbf{all}}{\Gamma \vdash L[e] : \tau'@\alpha}$$

$$\text{(iii)}\ \frac{\Gamma \vdash L\text{: } \mathbf{mapping}(\mathbf{address}!\mathrm{id} \Rightarrow \tau@\alpha\,)@\mathbf{all} \quad \mathrm{id} \notin \Gamma \quad \Gamma \vdash e\text{: } \mathbf{address}@\mathbf{all} \quad (e = \mathrm{id}' \vee e = \mathbf{me})}{\Gamma \vdash L[e] : \tau'@\alpha\,[\mathrm{id} \to e]}$$

$$\text{(iv)}\ \frac{\Gamma \vdash e : \tau@\alpha,\ L\text{:}\tau@\alpha,\ \Gamma \vdash e\text{:}\tau@\alpha'\ (\alpha = \alpha' \vee \alpha' = \mathbf{all})}{\Gamma \xrightarrow{L=e} \Gamma} \qquad \text{(v)}\ \frac{\Gamma \vdash L\text{: } \tau@\alpha\ \ \Gamma \vdash e\text{:}\tau@\mathbf{me} > @\alpha}{\Gamma \xrightarrow{L=e} \Gamma}$$

Listing 4 Solidity typing rules

Note that the rule (v) deals with delegation, which does not exist in zkay. *Stone* makes a `delegate-to` description based on JSON policies and types such that any violation of the delegation policy will incur an error, whereas zkay does not. In addition, we assume that such a delegation is always made from `@me`, which is the case in most of the examples we examined.

This type of judgment is followed by a long process of the analysis of variable declarations, expressions, statements, and others. Because the soundness and completeness of the zkay type system are proven, we can concentrate on the rule (v).

Theorem 1 The *SIR* type system is sound and complete when the delegation is made from the caller.

Proof) Assume that a *Stone* policy states that the owner of `r` is `@x` and that the access right will be delegated to `@y` such that the *SIR* type system has `r`: `@me>@y` in the declaration of r with caller `x`. If `@x` is `@all`, `r` is not private, and no delegation is required. If `@y` is the same as `@x`, again, no delegation is required. In other cases, when `r` is accessed by `@y`, which is formulated by `L = r;`, where `L : t@y`, and when `t` is an arbitrary value type, the access should be allowed according to the policy. By the rule (v) of Listing 4, we can ensure that no blocking occurs.

By contrast, assume that the owner of `r` is `@x` (where `@x ≠ @all`), and there is no delegation description in the *SIR* types. When `L = r;` is met, where `L: t@y` and `@y ≠ @x`, an error will occur as expected, because there is no rule matching for this case (i.e. the rule (v) does not match owing to the lack of a delegation description, and the rule (iv) is also not a candidate because `@me ≠ @y` and `@x ≠ @all`.) □

V. BACKEND OF *STONE*

Currently, the *Stone* prototype uses zkay as the backend. Thus, *SIR* programs should be manipulated to utilise the facilities of zkay. Although high-level structures of *SIR* and zkay are similar, and thus the transformation is a little easier, some analysis [8] and adaptation are needed. We call this module a zkay *Adaptor*.

As mentioned previously, in zkay, a delegation relationship is not described in a type, but is expressed as a special 'reveal' expression, which is completely different from that of *Stone*. For instance, if the expression `s*2` should be disclosed to `a` along with the current owner, zkay uses the expression `reveal(s*2, a)`.

Thus, after the *Stone Type Resolver* ensures that the *SIR* is well typed, which implies that there is no policy violation, the zkay *Adaptor* converts a delegation type specification into appropriate `reveal` expressions. By analysing the function body, it locates the program points to insert the `reveal`

Authorized licensed use limited to: University of North Texas. Downloaded on January 19,2024 at 23:43:12 UTC from IEEE Xplore. Restrictions apply.

expressions with the corresponding parameters. Then, a conversion function ρ inserts an appropriate reveal expression to the right-hand side of a given assignment statement if all type conditions are satisfied, according to the following rule:

$$\frac{\Gamma \vdash e{:}\tau 1@x{>}@y,\ w{:}\ \tau 2@y}{\Gamma \vdash \rho\,[\text{“w = e”}] = \text{“w = reveal(e, y)”}} \ (\rho\text{-Transformation})$$

For instance, consider the following function `f` in the *SIR*:

```
address a; uint@a y;
function f(uint@me>@a t) public { y = t*3; }
```

After analysing the body of function `f`, our *Adaptor* finds the assignment as the program point that should be converted using `reveal`. Thus, it injects a reveal expression to use the zkay facilities as follows:

```
address a; uint@a y;
function f(uint@me s) public {y = reveal(t*2, a);}
```

VI. Experiment

A. Functionality test

Figure 3 shows the results when we run a Solidity file and the JSON policy mentioned in the previous sections. (the messages and warnings are omitted due to the lack of space.)

```
escho@ubuntu:~/stone-main/stone-main/nytest$ zkay stonecheck Hospital.stn
//Using solc version v0.6.12
Stone Type checking file Hospital.stn:
Parsing... done
Type checking with solc...
  ***
done
Finished successfully
```

Figure 3 Example: A successful case of the *Stone* process

However, when we run Listing 1 with Listing 3 after removing the `delegate_to` field of the function `record`, *Stone* raises an error in that messages that the `r` should be delegated to `pat` as follows (i.e. access is blocked without proper access rights).

```
escho@ubuntu:~/stone-main/stone-main/nytest$ zkay stonecheck Hospital.stn
//Using solc version v0.6.12
Stone Type checking file Hospital.stn:
Parsing... done
Type checking with solc...

WARNING:
At line: 18;5
    function check(bool@me>@all r) public {
----/

Function state mutability can be restricted to view


done
Preprocessing AST... done
Zkay type checking...

COMPILER ERROR:
At line: 14;21, in function 'record' of contract 'MedStats'
        risk[pat] = r;
--------------------/

Expected type bool@x but got bool@me
```

Figure 4 Example: Failed case of the *Stone* process

B. Test against actual contracts

Stone was tested against 50 smart contracts used to verify zkay [6]. Because all of them are written in zkay, we first modify the syntax and manually separate Solidity and JSON to place them into *Stone*. All 50 contracts were successfully executed.

We collected 10 smart contracts from Etherscan [9] as a test for real-world contracts. We first select the contracts in Solidity 6.0, because this version is compatible with the zkay facilities that we currently use as the backend. The list of smart contracts is `ERC20`, `AdminAuth`, `BotRegistry`, `ContextUpgradeSafe`, `Pausable`, `CM`, `Ownable`, `OwnableUpgradeSafe`, and `CMErc20MintBurnSnap`, some of which are well-known. We created simple policies for these contracts and applied them to *Stone*, obtaining successful results, as expected.

VII. Conclusions

Although smart contracts run on blockchains, this potentially harms the data from a privacy preservation perspective. Blockchains disclose all data to the participants of the chain. Although existing research on the support of private variables used in programs and the encryption/decryption protocols applied in blockchain has shown that these could be options to overcome this limitation, they do not sufficiently consider the characteristics of smart contracts: traditional type-based privacy preservation might confuse developers and reduce their productivity, and protocol-based support for encrypted data is quite dedicated to restrict the types of private data owing to the overhead involved. Although recent studies on off-chain computing with zero-knowledge proofs seem promising, their usage might also bother contract developers for reasons similar to those of previous approaches.

We propose *Stone* as a method to overcome the limitations of previous methods. By attaching a separate JSON policy to a plain Solidity contract, the storage data of the contract will be protected from being divulged to the chain. All boilerplates of the type systems and zero-knowledge-proving mechanisms are hidden under the intermediate *SIR* representation. We elaborated on the automatic transformation from high-level Solidity programs and JSON policies to low-level proving mechanisms, which ensures privacy preservation both statically (through privacy-aware type checks) and dynamically (by generating proofs). We believe that *Stone* will be preferable to developers owing to its practicality and expected enhancement of productivity. Currently, we are planning to support more adaptors for various proof-generating backends.

Acknowledgment

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (2020R1A2C2008864, Study on Smart Contract Framework for Zero Knowledge Proof).

References

[1] J. Ligatti et. al, “Edit automata: Enforcement mechanisms for run-time security policies”, *International Journal of Information Security*, vol. 4, pp. 2–16, 2005

[2] A. Sabelfeld and A. C. Myers, “Language-based information-flow security”, in *IEEE Journal on Selected Areas in Communications*, vol. 21, no. 1, pp. 5–19, Jan. 2003, doi: 10.1109/JSAC.2002.806121.

[3] E.B. Sasson, et. al. “Zerocash: De-centralized anonymous payments from bitcoin”, in Proc. of IEEE Symposium on Security and Privacy (S&P). pp. 459–474, 2014

[4] E. Ben-Sasson, A. Chiesa, E. Tromer, M. Virza, “Scalable zero knowledge via cycles of elliptic curves”, *Advances in Cryptology – CRYPTO 2014 Lecture Notes in Computer Science*, vol. 8617. Springer, Berlin. https://doi.org/10.1007/978-3-662-44381-1_16, 2014

[5] J. Eberhardt et. al, “ZoKrates–Scalable privacy-preserving off-chain computations”, in *Proc. of the Workshop on Scalable and Resilient Infrastructures for Distributed Ledgers (SERIAL)*, pp. 7–12, 2018

[6] S. Steffen et. al, “zkay: Specifying and enforcing data privacy in smart contracts”, in *Proc. of the ACM conf. on Computer and Communications Security (CCS)*, pp. 1759–1776, 2019

[7] N. Baumann et. al, “zkay v0.2: Practical data privacy for smart contracts”, *arXiv:2009.01020v2*, Sept. 2020

[8] A. Aho, M. Lam, R. Sethi, J. Ullman, *Compilers: Principles, Techniques, and Tools (2nd Ed.)*, Addison Wesley, ISBN: 978–0321486813, Aug. 2006

[9] Etherscan, The Ethereum Blockchain Explorer, https://etherscan.io

Authorized licensed use limited to: University of North Texas. Downloaded on January 19,2024 at 23:43:12 UTC from IEEE Xplore. Restrictions apply.