Fidelius 对数据托管的支持

范学鹏

2022 年 8 月 15 日

1 需求分析

在已有的系统中，Fidelius 将系统抽象为了数据提供方与数据使用方，并要求数据提供方部署 Fidelius 计算节点（具备 TEE 环境）。然而，在很多场景下，数据提供方并不能部署 Fidelius 计算节点，这一方面是因为数据提供方没有部署环境或运维能力，另一方面是数据提供方的成本考虑。因此，为了使数据提供方在没有 Fidelius 计算节点的情况下提供数据服务能力，需要提供*数据托管能力*，即将数据托管到第三方进行计算，同时需要满足隐私计算的要求。

2 数据托管

我们首先在系统抽象中增加*算力提供方*，算力提供方不但提供算力，也提供必要的加密数据存储，以便在进行计算分析时，尽快获得数据，为了方便描述，下文不再赘述数据存储部分。在新的系统抽象中，算力提供方需要部署 Fidelius 隐私计算节点，因此需要相应的 TEE 环境，而数据提供方及数据使用方则不再需要部署隐私计算节点或 TEE 环境，仅需要基本的 Fidelius SDK 即可。当算力提供方与数据提供方为同一方时，系统抽象退化为之前的版本，即只有数据提供方和数据使用方。

在本文，数据托管指数据提供方将原始数据加密后，将加密数据转移到算力提供方，算力提供方根据数据提供方的授权提供数据服务。注意，将原始数据直接转移给算力提供方是危险的，这是因为算力提供方可以任意使用、转发相应的原始数据，从而造成数据泄露。因此，这在本文并不称为数据托管。

在安全性方面，Fidelius 假设算力提供方不会与数据提供方或数据使用方合谋。在这一假设下，算力提供方仅提供算力，不能获取被托管数据、计算参数、计算过程以及计算结果的任何信息；同时算力提供方仅能执行经过“授权”的计算任务，不能擅自使用托管的数据。

3 相关概念

为方便下文描述，本节首先罗列相关名词、概念以及符号。

- `keymgr` 是一个 Fidelius 中的命令行工具，用于创建、删除基于椭圆曲线的非对称密钥，以及查看相应的公钥。在 `keymgr` 中，私钥生成过程是在 TEE 内，且生成的私钥使用设备相关但不可见的密钥（如 TPM 中存储的密钥）加密后存储在本地；相应的公钥则可以公开查看。

- 典钥：是一对基于椭圆曲线生成的公私钥（典公钥 P^D，典私钥 S^D）。使用 `keymgr` 在 TEE 内生成，且私钥使用设备相关但不可见的密钥加密后存储到本地，因此，典钥是设备相关的，可以看做是一个 Fidelius 计算节点的标识。

- `yterminus` 是一个 Fidelius 中的命令行工具，用于在非可信环境下完成密码相关的操作。

- 枢钥：是一对基于椭圆曲线生成得公私钥（枢公钥 P^S，枢私钥 S^S）。使用 `yterminus` 在普通环境下生成，也可以使用相关的 javascript 库在网页生成，私钥直接明文存储。因此，枢钥是设备无关的。

- 智能合约：特指部署在区块链上的智能合约，用于完成必要的交互、验证操作。注意，Fidelius 本身不包括智能合约，此处的智能合约仅表示一个可信第三方，也可以使用中心化的服务器代替。根据上下文，本文亦会使用区块链，除非特别说明，同样指部署在区块链上的智能合约。类似的，上链指调用智能合约。

- 枢私钥转发：指将枢私钥通过典公钥转发给指定的算法，记为

  $$F(S^S, P^D, H_{\text{algo}}),$$

  即将枢私钥 S^S 通过典公钥 P^D 转发给哈希为 H_{algo} 的算法。此时，在 H_{algo} 对应的算法内，调用 `request_private_key_for_public_key`(P^S) 则返回 (S^S, P^D)，而在 $H'(H' \neq H_{\text{algo}})$ 对应的算法中以同样的参数调用同样的函数则返回 (nil, nil)，并抛出异常。函数

  `request_private_key_for_public_key`()

  通过加密的信道从 `keymgr` 获取枢公钥对应的枢私钥，以及加密枢私钥所使用的典公钥。

其他涉及到的概念还包括非对称加密、签名等，本文不再赘述。

4 系统原理

假设数据提供方为 p，数据使用方为 a，算力提供方为 c，数据提供方持有枢钥，P_p^S，S_p^S，数据使用方持有枢钥，P_a^S，S_a^S，算力提供方持有典钥 P_c^D，S_c^D。数据方持有的数据为 M。

数据托管的原理如图 1所示，

1. 加密本地数据：即 $\text{Enc}_{P_p^S}(M)$。

2. 发送加密后的数据：可以使用任意可信的文件共享服务，如 FTP，HTTP，IPFS 等。

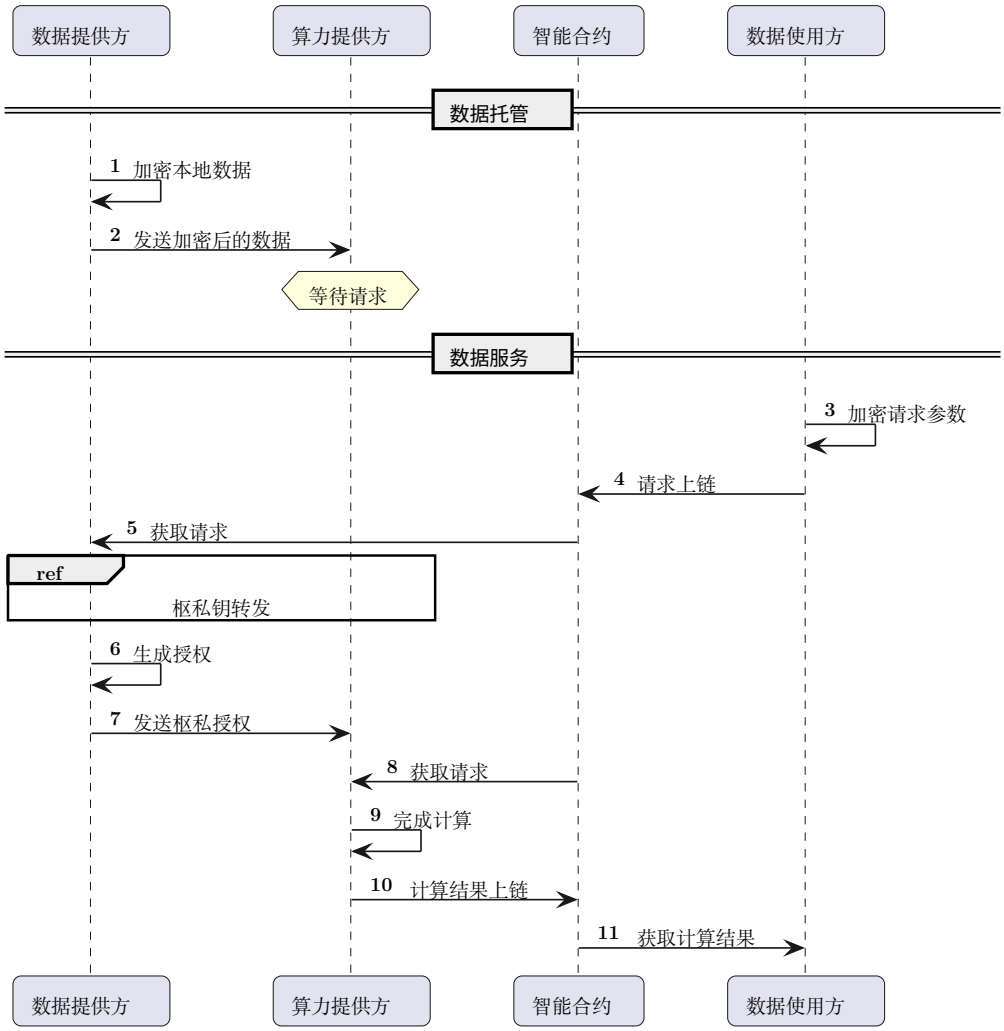

图 1: 数据托管模式下的数据服务流程

3. 加密请求参数：即 $\iota = \mathrm{Enc}_{P_a^S}(\mathrm{param})$，其中 param 为请求所使用的参数。

4. 请求上链：一个请求为一个多元组

$$\{\iota, H_{\mathrm{algo}}, H_{\mathrm{M}}, P_a^S, \ldots\},$$

其中 H_{algo} 为算法 (enclave) 的哈希，H_M 为数据哈希，$\ldots$ 为枢私钥转发需要的内容，此处省略。

5. 获取请求：通过监听区块链上的交易（或 event）获得请求的内容。

6. 生成授权，授权是一个签名，表示为一个三元组 $\{s, P_p^S, H_M\}$，其中

$$s = \mathrm{Sign}_{S_p^S}(H_\iota, H_{\mathrm{algo}}, P_c^D, H_M)$$

H_ι 为加密后请求参数的哈希。

7. 发送授权：注意此授权可以不使用智能合约进行转发。

8. 获取请求：通过监听区块链上的交易（或 event）获得请求的内容，注意，此步骤可以和步骤 5同步进行。

9. 完成计算：获得授权，读取数据，验证授权，并进行计算。

10. 计算结果上链，注意，此处的计算结果是加密的，且可能包含了其他与结果相关的内容或形式。

11. 获取计算结果：通过监听区块链上的交易（或 event）获得加密的计算结果。此处省略了后续对结果的解密操作。

枢私钥转发 注意到图 1中包含了枢私钥转发，为 $F(S_p^S, P_c^D, H_{\mathrm{algo}})$。

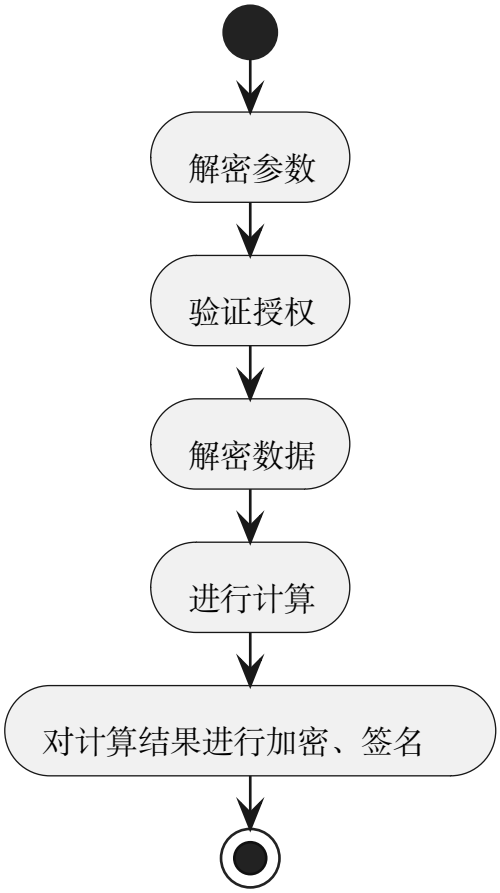

图 2: 验证授权在整个计算节点中的位置

验证授权 步骤 9中包含了验证授权的过程，图 2表示了验证授权在整个流程中的位置，图 2所示的流程在一个 enclave 内，因此不可篡改。算法 1描述了验证授权的过程，其中 ι 为加密后的参数，数据 d 为一个二元组 $\{P_p^S, H(M)\}$，授权 a 则为前述的三元组。

算法 1并未检查加密参数或数据的合法性，单独检查算法 1的话，容易认为可以通过使用历史上有效的参数或数据来绕开授权验证，但考虑到验证授权之前会对参数进行解密，验证之后会对数据进行解密，而且最后会对计算结果进行签名，因此，这些检查会在其他步骤进行，从而使得验证授权是有效的。

容易将算法 1扩展到多个数据、多个授权的情况，本文省略。

Algorithm 1 验证授权

procedure CHECK_ALLOWANCE(ι, d, a) ▷ ι 为加密后的参数，d 为数据，a 为授权

shu_skey, dian_pkey ← `request_private_key_for_public_key`($a.P_p^S$) ▷ 枢私钥转发

$m \leftarrow H_\iota + H_{\text{algo}} + \text{dian\_pkey} + a.H_M$ ▷ H_{algo} 为本地获取

$f \leftarrow$ `verify_signature`($a.P_p^S, m, a.s$)

$v \leftarrow \{\}$

if f is **false then** ▷ 无效授权

return false

else ▷ 签名有效

$v \leftarrow v \cup \{a.P_p^S + a.H_M\}$

end if

if $d.P_p^S + d.H(M) \in v$ **then** ▷ 授权有效

return true

else ▷ 使用的授权是其他数据的授权，无效

return false

end if

end procedure

算力提供方的性质 在本系统中，算力提供方有三个特性：

1. 保密性：算力提供方不能获取被托管数据、计算参数、计算过程以及计算结果的任何信息。

2. 可控性：算力提供方每次执行一个新的请求时，都需要得到数据提供方的授权，因此，算力提供方即使拥有（加密）数据，也有相应的请求，却不能使用数据进行计算。

3. 可审计：当数据提供方对某一此数据的使用有疑问或争议时，算力提供方可以出示相应的授权以供审计。

5 Fidelius 中的实现

在 Fidelius 中，授权（allowance）的定义在文件

`ypc/core/include/ypc/corecommon/nt_cols.h`

中。相关的处理逻辑在文件

`ypc/core/include/ypc/core_t/analyzer/interface/allowance_interface.h`

中。

此外，可以使用 `yterminus` 生成授权

```
yterminus --allowance --use-param PARAM_HASH
    --use-enclave-hash ENCLAVE_HASH
  --tee-pubkey DIAN_PKEY --dhash DATA_HASH
```

智能合约 本功能的实现不需要引入智能合约。

6 系统功能

本文仅描述了 Fidelius 部分的原理及实现，完整的数据托管功能还包括方便用户使用的工具及内容管理，此处列出一个可能的数据托管平台应该包括的功能：

1. 算力提供方注册、注销，查看已托管的数据，查看所有算力提供方。
2. 数据选择托管方，数据托管到多个算力提供方，查看某个数据是否是托管的。

3. 数据提供方查看请求，授权请求，查看授权情况，请求出现时的提醒。

4. 系统包括多个算力提供方，数据使用方可以将同一个数据托管到多个算力提供方。

5. 算力提供方按照比例抽取佣金，所抽取的比例可以由算力提供方自行设置 [1]。

额外的，数据提供方可以有自己的客户端或 Web 页面，基于数据提供方的枢私钥、按照规则进行自动、批量的授权。

7 未来工作

算力提供方提供了算力，因此应该获得一定的收益，按照计算过程中使用的资源进行收费，例如 CPU，磁盘、内存，为此，需要对资源进行监控、计算。

算力提供方执行的算法依赖于正确的实现，例如正确的对授权进行了检查，虽然 Fidelius 提供了相应的代码库，但是恶意的算法开发者仍有机会篡改其中的协议。

因此，为了解决以上两个问题，未来需要两个工作：

1. 寻求一种在 TEE 中对计算资源进行监控、计算的方法，同样的，该计算方法不能被恶意篡改;

2. 引入可信的编译器，在 TEE 中对代码进行编译，链接，签名，保证使用的代码库是未经篡改的。

[1] 未来，可以改为按照耗费的计算资源进行收费，但目前还存在技术难度，无法实现