Децентрализованый протокол доверенных вычислений основанный на blockchain

Глеб Русяев[1]
[1] *Летово*

Исходя из сложности анализа чипов концепции “root-of-trust”, созданых для противостояния аппартному реверс-инжинирингу, мы создали первый децентрализованый протокол доверенных вычислений основанный на Ethereum. Также, благодаря своим псевдо-гомоморфным свойствам, и используя концепцию ”оракулов“ - контейнеров с повышенными привилегиями, становится возможным эффективное решение различных проблем computer science. В качестве примера приводится решение усложненной версии проблемы Эндрю Яо – “Secure Multi-Party Computation”.

Ключевые слова— доверенные вычисления, blockchain, гомоморфное шифрование, SMPC

I. ВВЕДЕНИЕ

A. Гомоморфное шифрование

Вопрос о том, как производить доверенные вычисления в окружении, к которому злоумышленник может получить физический доступ является одним из основных вопросов современной криптографии. Решение этой проблемы позволит нам не только создавать более совершенные anti-tampering системы, но и проводить безопасные облачные вычисления.

Один из теоретических подходов к созданию подобных систем - гомоморфное шифрование, позволяющее производить вычисления над зашифрованными данными. Пусть $f\colon \mathbb{R}^n \to \mathbb{R}^k$ произвольная функция вычислимая для некоторого вектора $x \in \mathbb{R}^n$, так что:

$$f(x) = y$$

Тогда функция $\xi_p : \mathbb{R}^p \times K^c \to \mathbb{R}^p$, где K^c - пространство ключей, для которой справдедливо, что:

1. Если $\xi_p(a,k) = a_e$, тогда $(a,k) \mapsto a_e$
2. Существует функция $\overline{\xi_p}$, такая, что $\overline{\xi_p}(\xi_p(a,k),k) = a$, т.е. $\overline{\xi_p}$ дешифрует $\xi_p(a,k)$
3. $\overline{\xi_k}\left(f\left(\xi_{dim(a)}(a,k)\right),k\right) = f(a)$

является гомоморфным шифрованием относительно функции f. Полное же гомоморфное шифрование это такое ξ_p, которое сохраняет данные свойства для любой произвольной функции f. Последние открытия в области гомоморфного шифрования, такие как работа Крейга Гентри “Полностью гомоморфное шифрование с использованием идеальных решеток” [1] теоретически позволяли ответить на данный вопрос, однако страдали от некоторых недочетов.

Во-первых, гомоморфное шифрование предполагало существенное изменение временной сложности вычислений (т.е для функции f существует аналог f' поддерживающий гомоморфное шифрование, однако их временная сложность различается), что влечет к изменению представления о эффективных алгоритмах и препятствует портированию программ для подобных систем.

Во-вторых, большинство походов связанных с гомоморфным шифрованием предполагают использование функционального программирования, что тоже вносит большое влияние в архитектуру проектов и усложняет перенос уже существующих программ.

Таким образом задача практического решения подобной проблемы, а именно реализация гомоморфного шифрования для императивных программ с малым изменением вычислительной сложности до сих пор остается открытой. [2]

B. Необходимые определения

Для продолжения нам необходимо определение программы. Вектор $\mathfrak{P}^T \in \{0,1\}^n$, реализующий машину Тьюринга T называется программой. Очевидно, что код на любом языке программирования можно привести к такому виду, однако, не умоляя общности, чтобы избежать объяснения процесса компиляции, мы будем использовать несколько необычный процесс получения вектора $\mathfrak{P}^T$ из кода программы.

Лемма 1. *Лемма о сводимости кода к программе утверждает, что код на любом Тьюринг-полном языке программирования можно привести к виду* $\mathfrak{P}^T$
Доказательство:

1. *На основании Тьюринг-полноты выбранного языка программирования и языка Brainfuck, перепишем наш код на нем. Очевидно, что это возможно всегда.*
2. *Сопоставим нашему слову* $\mathfrak{B}$ *(коду на языке Brainfuck) алфавита* $\{+,-,<,>,[,]\}$ *вектор* $\mathfrak{P}^T$*, так, что* $\mathfrak{B}_i \mapsto (\mathfrak{P}^T_{3i}, \mathfrak{P}^T_{3i+1}, \mathfrak{P}^T_{3i+2})$*, где* $\mathfrak{P}^T_{3i+k}$ *—* $(k+1)$*-й разряд двоичного представления порядкового номера элемента* $\mathfrak{B}_i$ *в лексиграфически-отсортированном алфавите.*

Лемма 2. *Лемма о сводимости ввода к программе. Пусть ввод программы так-же представим ввиде вектора* $I \in \{0,1\}^p$*. Тогда если* $\mathfrak{P}^T$ *— программа ожидающая ввода размера* $P = inp\left(\mathfrak{P}^T\right)$*, утверждается, что существует такая функция* $i\colon \{0,1\}^p \times \{0,1\}^n \to$

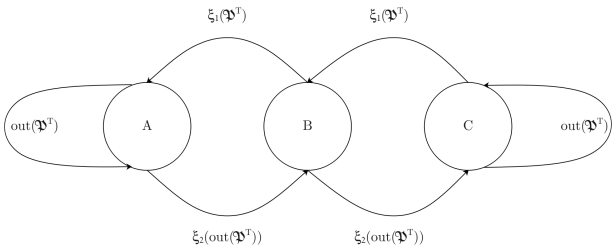

Рис. 1. Диаграмма системы.

$\{0,1\}^k$, *которая сопоставляет паре* $\left(\mathfrak{P}^T, I\right)$ *программу* $\mathfrak{P}^{T'}$, *ожидающую ввода размера* $\max(0, P-i)$.
Доказательство:

Модифицируем программу таким образом, что после каждого ввода от пользователя мы будем попорядку присваивать переменной получившей ввод некую константу принадлежащую коефициентам вектора I начиная с первого коефициента. Таким образом, значение первых $\min\left(dim\left(I\right), inp\left(\mathfrak{P}^{T'}\right)\right)$ *вводов от пользователя не важно.*

Это значит, что мы можем привести все нижеизложенные ситуации к виду программ.

1. Необходимо вычислить $\mathfrak{P}^T$ на полном вводе a, не предпологая ввода со стороны пользователя(т.е. $dim\left(a\right) = inp\left(\mathfrak{P}^T\right)$)

2. Необходимо вычислить $\mathfrak{P}^T$ на частичном вводе a (в том числе и $dim\left(a\right) = 0$), предпологая ввод со стороны пользователя(т.е. $dim\left(a\right) < inp\left(\mathfrak{P}^T\right)$)

Начиная с этого момента мы будем опрерировать только программами, подразумевая их связь с вводом и кодом на произвольном языке программирования.

C. Поход к решению задачи

Для решения поставленной вначале статьи задачи, нам необходимо создать систему подобную представленной на диаграме. На ней изображен процесс, в котором C запрашивает у B вывод программы $\mathfrak{P}^T$, причем он никогда не разглашает ни B, ни саму программу, ни её вывод. $A-$ в свою очередь является некоторым исполнителем, к вводу и выводу которого B имеет непосредственный доступ. Мы можем предположить, что A это процессор, который находтся в распоряжении B, однако в таком случае у него имеется доступ как к дебаггеру, так и просто возможность считывать значения шины A. Таким образом, A должно работать как процессор, однако не позволять анализировать внутреннее содержимое даже своему владельцу.

Моя работа предлагает именно такой подход, при котором, полагаясь на современные меры противодействия reverse-engineering, мы считаем A недоступным для анализа (и утечки $\mathfrak{P}^T$), а следовательно $out\left(\mathfrak{P}^T\right)$ тоже при необходимости сохраняет секретность, т.к. он может быть зашифрован внутри программы симметричным алгоритмом. Очевидно, что, вне абстракции, мы считаем A неким чипом плохо поддающемся обратной разработке. Концепция подобных чипов называется "root-of-trust" и она показала свою эффективность против утечек информации и пиратства в консолях XBOX и PlayStation поколения 2013 года.

Чип псевдо-гомоморфных вычислений **PHCM** – это такой чип, который можно подразделить на 2 части: LSL, HSL.

1. HSL подразумевает собой чип концепции "root-of-trust" состоящий из некоторого процессора способного к исполнению произвольной программы $\mathfrak{P}^T$ и дополнительному набору действий для работы с засекреченными данными (RSA, AES и т.д.), ROM/XOM и RAM памяти, доступ к которым осуществляется по-средствам Memory Management Units, а так-же watch-dog модуля для защиты от физических аттак, таких как fuzzing. Причем ROM/XOM память содержит приватные ключи α и γ алгоритма RSA, где α устанавливается на этапе производства, а γ генерируется на этапе производства (т.е. даже производитель чипа знает только γ-pub)

2. LSL представляет собой высокоуровневую надслойку над HSL и выполняет вспомогательные функции такие как хранение прошивки для HSL и программ в зашифрованном виде, чтобы предотвратить необязательные сеансы коммуникации с сервером и уменьшить нагрузку на сеть. LSL не обязана быть частью **PHCM**, она может существовать в форме ПО и не имеет особых требований по безопасности.

II. ПРОТОКОЛ

A. Основы

Для того, чтобы обеспечить работу **PHCM**, нам нужен унифицированный протокол П. Пусть $\Pi[i] \in \{0,1\}^6$, $\Pi[i] \equiv p, p \in \mathcal{P}\left(\{\alpha_d, \alpha_e, \beta_d, \beta_e, \gamma_d, \gamma_d\}\right)$, где i это обозначение субъекта протокола, а k_d, k_e означают секретный и открытый ключ k соответственно. Значит $\Pi[i]$ однозначно определяет ключи, доступные субъекту i в конце работы протокола П. Так-же обозначим $\Pi'[i]$ как ключи известные субъектом i до работы протокола.
Зададим Π':

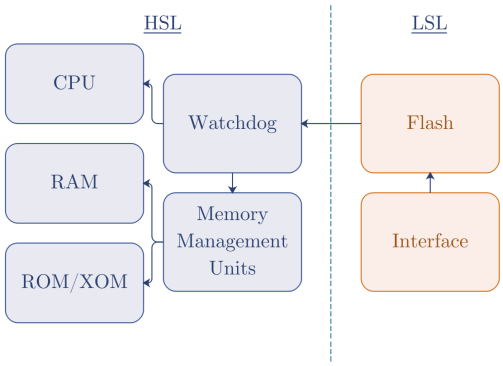

Рис. 2. Топология PHCM.

1. $\Pi'[HSL] \stackrel{\text{def}}{=} \Pi'[A] = \{\alpha_d, \alpha_e, \beta_e, \gamma_d, \gamma_e\}$
2. $\Pi'[LSL] \stackrel{\text{def}}{=} \Pi'[B] = \emptyset$
3. $\Pi'[C] = \{\alpha_e, \beta_d, \beta_e, \gamma_e\}$, причем C знает γ_e в отношении любого HSL.

Пусть $\mathfrak{C}_{I,J}(\phi, \eta)$ – защищенный туннель между субъектами I и J, основанный на алгоритмах RSA и AES с RSA-ключами ϕ, η, так что:

$$\{\phi_d, \eta_p\} \subseteq \Pi[I], \{\eta_d, \phi_p\} \subseteq \Pi[J]$$

Определим $\overline{\mathfrak{C}_{I,J}}$ как SSL/HTTPS туннель, ключи которого не представляют значения (к примеру сгенерированы случайно).

B. Handshake

Чтобы защитить протокол Π от прослушивания, создадим защищеный тунель $\overline{\mathfrak{C}_{HSL,C}}$. Далее необходимо провести так называемый handshake, который призван убедить всех участников в легитимности друг-друга, а именно:

1. C: убедиться, что HSL легитимен, т.е. располагает ключами α_d, γ_d.
2. HSL: убедиться, что C легитимен, т.е. располагает ключом β_d

Определим $rand()$ как случайное $r \in \mathbb{R}$, функцию $\mathfrak{s}(k, i_d)$ как электронную подпись значения k ключом алгоритма RSA i_d, а также пусть:

$$\mathfrak{s}'(k, i_{d1}, i_{d2}, \ldots, i_{dn}) = (k, \mathfrak{s}(k, i_{d1}), \mathfrak{s}(k, i_{d2}), \ldots, \mathfrak{s}(k, i_{dn}))$$

Тогда handshake проводится в 3 этапа:

1. $\overline{\mathfrak{C}_{HSL \to C}}$: $RSA_e(\mathfrak{s}'(\{\beta_e, \gamma_e, r\}, \alpha_d, \gamma_d), \beta_e)$, где $r = rand()$
2. C: проверяет корректность подписей $\mathfrak{s}'(\{\beta_e, \gamma_e, r\}, \alpha_d, \gamma_d)$, присутствие γ_e в базе данных пользователей и соответствие β_e. Затем генерирует сервер случайное значение $r_1 = rand()$ и создает сессию для ключа γ, с ключом сессии r_1. Если все прошло успешно, C должен быть уверен в легитимности HSL.
3. $\overline{\mathfrak{C}_{C \to HSL}}$: $RSA(\mathfrak{s}'(\{\gamma_e, r_1, r\}, \beta_d), \gamma_e)$
4. HSL: проверяет корректность подписей $\mathfrak{s}'(\{\gamma_e, r_1, r\}, \beta_d)$. Теперь и HSL убежден в C.

Лемма 3. *Лемма о легитимности. После успешно проведенного handshake внутри $\overline{\mathfrak{C}_{HSL,C}}$, все участники уверены в легитимности друг-друга.*
Доказательство:

1. *Пусть HSL нелегитимен, т.е. нерасполагает ключами α_d, γ_d, но на 2-м шаге C проверяет подлинность электронных подписей α_d, γ_d. Единственный возможный способ получения подобной подписи нелегитимным участником -- перехват или server impersonation. Перехват невозможен из-за того, что мы находимся внутри защищенного канала $\overline{\mathfrak{C}_{HSL,C}}$, поэтому рассмотрим сценарий impersonation.*

   Предположим, что злоумышленник выдал себя за сервер и принял handshake с легитимным HSL. В таком случае, он располагает информацией $RSA_e(\mathfrak{s}'(\{\beta_e, \gamma_e, r\}, \alpha_d, \gamma_d), \beta_e)$ и с её помощью может инициировать handshake с C. Хоть сервер и распознает его как HSL, злоучышленник все равно не сможет пользоваться своим статусом, так как не сможет расшифровать сообщение с шага 3, содержащее сессионный ключ.

2. *Пусть C нелегитимен, т.е. нерасполагает ключом β_a, тогда мы можем попробывать перенаправлять запросы на настоящий сервер и возвращать HSL его ответы. В этом случае мы будем просто Man In the Middle, в контексте защищенного канала и не сможем ни повлиять на пересылаемые данные, ни их прослушивать.*

Таким образом мы подтвердили устойчивость протокола к различным атакам на сетевом уровне.

C. Основные функции

Для того, чтобы реализовать основную функциональность **PHCM** нам необходимо два метода: “request” – метод для получения определенной программы с сервера и “execute” – для исполнения “instance” т.е. копии программы.

Обозначим $\aleph^T = (\xi(\mathfrak{P}^T), m)$ контейнером, причем $\xi(\mathfrak{P}^T)$ зашифрованная программа $\mathfrak{P}^T$, а m – метаданные содержащие id контейнера, все связанные с

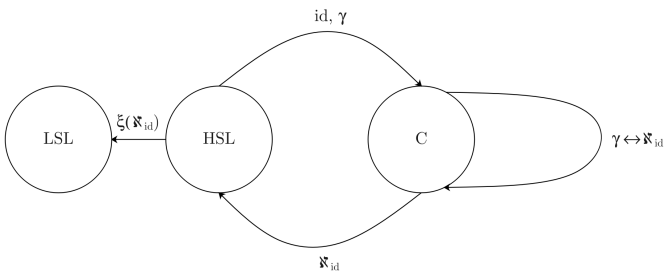

Рис. 3. Метод “request”.

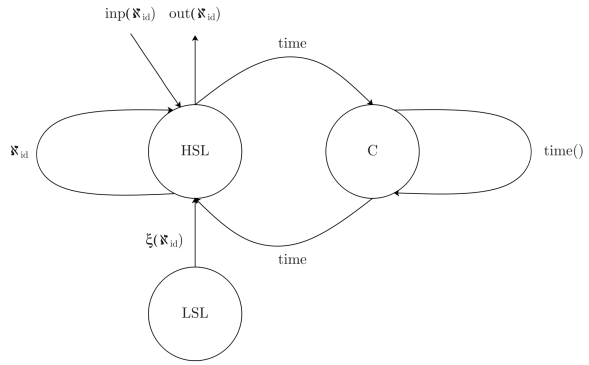

Рис. 4. Метод “execute”.

ним лицензии, необходимую прошивку и максимальную длинну ввода.

1. Метод “request”

На диаграмме 3 изображены коммуникации между LSL, HSL и C, при этом, очевидно, что трафик HSL проходит и через LSL, однако так как мы рассматриваем его в контексте защищенного канала $\overline{\mathfrak{C}_{HSL,C}}$, на диаграмме изображены стрелки идущие напрямую между HSL и сервером C.

Ниже приведен алгоритм описывающий работу сервера.

Algorithm 1 Request $\aleph^{TU}$

Require: r ключ сессии HSL, id уникальный ID запрашиваемого контейнера
  $s \leftarrow session(r)$
  $\gamma \leftarrow s_\gamma$
  $\aleph^T \leftarrow container(id)$
  $licenses \leftarrow \aleph^T.licenses$
  $userlicense \leftarrow \emptyset$
  for $i = 1 \dots |licenses|$ **do**
    if $licenses_{i\gamma} = \gamma$ **then**
      $userlicense \leftarrow licenses_i$
    end if
  end for
  if $userlicense.ts\_expiration > time()$ **then**
    $\aleph^{TU} \leftarrow \langle firmware, maxlen, \mathfrak{P}^T, userlicense.ts\_issued, userlicense.ts\_expiration\rangle$
    return $\mathfrak{s}'\left(\xi\left(\overline{\aleph^{TU}}, \gamma, \alpha\right)\right), \beta, \alpha\right)$
  else
    return 0
  end if

Таким образом сервер идентифицирует пользователя и отдает ему индивидуальный экземпляр контейнера $\aleph^T$, который называется $\aleph^{TU}$. После чего HSL должен передать его LSL на хранение, подписанный и зашифрованный ключом γ.

Лемма 4. *Лемма о форсированной лицензии $\aleph^{TU}$. Утверждается, что несуществует способа обойти лицензию экземпляра $\aleph^{TU}$. А именно:*

1. *Не существует способа использовать одну лицензию $\aleph^{TU}$ для двух пользователей*
2. *Не существует способа распространить $\aleph^T$ без лицензии*

Доказательство:
Для того, чтобы распространить контейнер в обход лицензии, необходимо получить его расшифрованную версию, однако так как ключи содержатся внутри HSL, это невозможно.

Единственный способ, которым можно использовать одну лицензию для двух пользователей, это подмена flash памяти LSL одного **PHCM** *на память другого (содержащего экземпляр полученый по лицензии). Именно для защиты от подобного рода ситуаций был введен ключ γ, ведь ключ γ в отличие от α не един для различных экземпляров* **PHCM**.

Здесь мы рассмотрели запрос сервера напрямую от центра сертификации C, однако возможен и другой способ получения $\aleph^{TU}$. Для того, чтобы позволить производителям сохранять независимость от центров сертификации, мы создадим блокчейн сеть о которой поговорим в последней секции данной главы.

2. Метод “execute”

На диаграмме 4 изображен процесс проверки временных лицензий, хотя конечно скорее всего лицензии будут выдаваться бессрочно, и в таком случае мы не будем связываться с сервером вообще.

Лемма 5. *Лемма о локально-форсированной лицензии $\aleph^{TU}$. Утверждается, что несуществует способа обойти лицензию экземпляра $\aleph^{TU}$. А именно:*

1. *Не существует способа обойти временную лицензию $\aleph^{TU}$*

Algorithm 2 Execute $\aleph^{TU}$

Require: *id* ID копии контейнера в LSL.
  $\aleph^{TU}_{enc} \leftarrow LCL.request\,(id)$
  if $not\,(check\_signatures\,(\aleph^{TU}_{enc}, \alpha, \gamma, \beta,))$ **then**
    return 0
  end if
  $\aleph^{TU} \leftarrow \xi_d\,(\aleph^{TU}_{enc}, \gamma, \alpha)$
  $expiration \leftarrow \aleph^{TU}.ts\_exp$
  $time \leftarrow C.request\_time\,()$
  if $expiration > time$ **then**
    $input \leftarrow inp\,()$
    if $|input| <= \aleph^{TU}.max\_len$ **then**
      return $exec\,(\aleph^{TU}, input)$
    else
      return 0
    end if
  end if

2. *Не существует способа распространить $\aleph^T$ без лицензии*

Доказательство:
Первый пункт подтверждается тем, что время запрошенное с серера (в формате UNIX timestamp) будет точным на этот момент. Для форсирования второго пункта нам необходимо запускать все контейнеры в изолированном окружении, так чтобы контейнер не смог повлиять на своих соседей и работу HSL вцелом.

D. Blockchain

Введем новый ключ RSA λ – индивидуальный для каждого пользователя, он будет использовать его (публичную часть) как свой логин, оформляя на него лицензию. Так-же зададим создателя ПО Ф, обладающего ключом ϕ. Итак пусть Ф сотрудничает с сертификаторами $P = \{\beta_1, \beta_2, \ldots, \beta_n\}$, причем каждый сертификатор однозначно характеризуется его ключом β.

Предположим пользователь обладающий ключом λ заключил сделку с Ф и стал обладателем его лицензии в k-экземплярах сообщив ему k пар вида (β, γ), причем $\beta \in P$. В таком случае далее представлен сценарий действий Ф:

1. Ф регистрирует лицензии своим ключом ϕ для всех k пар и запрашивает с каждого сертификатора данные $(\alpha_e, \beta_e, \gamma_e, \aleph^{TU}_{enc}) \subseteq L$, где L множество лицензий

2. Ф размещает torrent-раздачу с магнет-ссылкой m

3. Ф посылает в блокчейн запрос, подписанный своим ключом ϕ:
   `register {"user":` λ_e `, "licenses": {{` $\alpha_e, \beta_e, \gamma_e,$ `m}, ...}}`

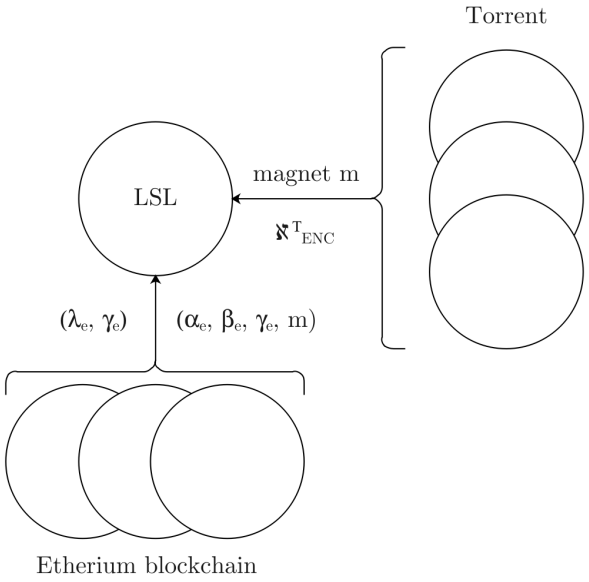

Рис. 5. Blockchain “request”.

Очевидно, что подобная система не будет работать без Proof-Of-Work алгоритма и блоков, которые в свою очередь будут майниться только за награду. Поэтому вместо написания блокчейна с нуля, мы будем использовать платформу Ethereum, которая обеспечит нам майнинг блоков.

На диаграмме 5 изображен метод получения экземпляра по лицензии (т.е. модифицированный ”request“)

Таким образом проткол становится децентрализованным.

III. ОРАКУЛЫ

A. Реализация

Оракул - это особый контейнер код которого публичен. Он отличается от всех остальных тем, что C хранит лист доверенных контейнеров в виде хешей и соответствующих им ключей. Когда пользователь получает **PHCM**, он делает запрос сертификатору C вернуть список оракулов и получает:
$\mathfrak{s}'$`({ "version": 1, "oracles": {"oracle": "1c5863cd55b5a4413fd59f054af57ba3c75c0698b3851d70f99b8de2d5c7338f "key":` $RSA(ok, \alpha)$`}, ... }` $, \beta_d)$

В таком виде список оракулов может быть отправлен в **PHCM**, причем любой другой предустановленный список оракулов будет заменен, в случае если “version” последнего превосходит предыдущий. Для запуска оракула приминяется следующий алгоритм:

Algorithm 3 Execute oracle **O** with hash h

Require: h, **O** (**O** - код оракла, полученный от пользователя)
  $\mathbf{O}_h \leftarrow sha256\,(\mathbf{O})$
  if $\mathbf{O}_h = h$ **then**
    $ok \leftarrow \emptyset$
    for $i = 1 \ldots |oracle\_list|$ **do**
      if $oracle\_list.oracle = h$ **then**
        $ok \leftarrow oracle\_list.key$
      end if
    end for
    if $ok = \emptyset$ **then**
      return 0
    end if
    ok $\leftarrow RSA_d\,(ok, \alpha_d)$
    $exec\_with\_input\,(\mathbf{O}, ok)$
  else
    return 0
  end if

B. Оракул-SMPC

Оракулы создают дополнительный уровень доверия и позволяют решать проблемы неразрешимые при обычной работе HSL. К примеру ниже приведено решение "Secure Multi-Party Computation Problem". Для начала мы продемонстрируем сам контейнер (алгоритм 4). Функция "out" подписывает вывод подписью оракула перед отправкой

Algorithm 4 Oracle $SMPC$: $O\,(n) + O\,(f)$

  $func \leftarrow inp\,()$
  $parties\_am \leftarrow inp\,()$
  $parties \leftarrow \emptyset$
  $ok \leftarrow request\_oracle\_key\,()$
  for $i = 1 \ldots parties\_am$ **do**
    $user\_pubkey \leftarrow inp\,()$
    $parties = parties \cup \{user\_pubkey\}$
  end for
  $out\,(ok_e, alpha_e, gamma_e, beta_e)$
  $data, proofkeys \leftarrow \emptyset$
  for $i = 1 \ldots |parties|$ **do**
    $T \leftarrow establish\,(\mathfrak{C}_{\mathbf{O},i}\,(ok, parties[i]))$
    $magic\_val \leftarrow T_{i\rightarrow\mathbf{O}}\,(inp\,())$
    $salt \leftarrow rand\,()$
    $out\,(T_{\mathbf{O}\rightarrow i}\,(func, magic\_val, parites[i], salt))$
    $salt', proofkey, value \leftarrow T_{i\rightarrow\mathbf{O}}\,(inp\,())$
    if $salt' \neq salt$ **then**
      return 0
    end if
    $proofkeys = proofkeys \cup \{proofkey\}$
    $data = data \cup \{val\}$
  end for
  $result \leftarrow exec\,(f, data[1], data[2], \ldots, data[n])$
  $output \leftarrow \mathfrak{s}'\,((result, func), proofkeys[0], \ldots, proofkeys[n])$

  $out\,(output)$

Теперь обсудим действия пользователей, которым небходимо вычислить данную функцию.

1. Они выбирают одного пользователя с $PHCM$ доверенного ими центра сертификации C, назовем его U.
2. U запускает оракул $SMPC$ для функции f, тогда вся коммуникация между пользователями и оракулом проходит через U.
3. Пользователи отправляют свои публичные ключи оракулу и получают информацию о оракуле взамен: $(ok_e, \alpha_e, \gamma_e, \beta_e)$. После проверки публичных ключей они приступают к следующему этапу
4. На основе отправленного пользователем ключа и ключа оракула строится защищенный канал T
5. Оракул генерирует случайное число "salt"
6. Через канал T каждый пользователь отправляет случайную константу и получает в ответ $(func, const, pubkey, salt)$, где pubkey его публичный ключ. Затем он проверяет полученные данные.
7. В случае достоверности, пользователь отправляет через канал T сообщение вида $(salt, proofkey, value)$, где "proofkey" уникальный сгенерированый секретный ключ который будет использоваться как гарантия использования данных $value$ при вычислении функции f.
8. Оракул подписывает результат вычисления в формате $(result, function)$ ключами всех пользователей и возвращает его U
9. U отправляет его остальным пользователям

Лемма 6. *Лемма о выполнении оракулом-SMPC условий SMPC. Она утверждает, что следуя вышеизложенному протоколу невозможно допустить нарушение условий SMPC. А именно: хотябы один из участников получает ненастоящий результат или хотябы один из участников узнает значения другого.*

Доказательство:

Предположим один из участников получил результат $(fakeres, f)$, причем функция f должна была вывести (res, f). Очевидно, что в таком случае значения остальных пользователей были модифицированы, однако тогда найдется такой пользователь, подпись которого незадействована при подписании результата.

Так-же предположим, что один участник узнал значения другого, это возможно в случае если первый участник - это U т.е. хост оракула, который загрузил в вредоносную функцию f. Однако на этапе 6 каждый пользователь получил взамен случайной константы (которая не является ценной информацией), функцию f. Из чего он может сделать вывод о нелегитимности U и прекратить дальнейшие взаимодействия с ним.

На этом возможности оракулов не заканчиваются, уровень доверия возникающий внутри PHCM позволяет эффективно реализовать как решения других задач имеющих решения с гомоморфным шифрованием (Zero Knowledge Proof [3], ...), так и стандартных задач крипто-процессоров, по сути делая его универсальной платформой для операций с секретной информацией.

IV. ПРИМИНЕНИЯ

В своем стандартном виде протокол находит свое приминение среди доверенных облачных вычислений, которые стали особенно актуальны после роста капитализации рынка облачных вычислений с 6 до 236 миллиардов долларов США [4]. На ряду с этим, благодоря децентрализации и прозрачности, PHCM может стать частью новых прозрачных anti-tampering систем в игровой индустрии, сочетая возможность модификации ПК с большей безопасностью для издателя, как от недобросовестного пользователя (что и стало причиной появления консолей), так и от недобросовестной площадки (используюя blockchain).

Более того, оракулы позволяют сделать PHCM универсальной платформой, способной как на решения теоретических задач криптографии, так и узкоспециализированных задач крипто-процессоров. В качестве демонстрации возможностей оракулов было приведено решение усложненной версии проблемы Эндрю Яо – “Secure Multi-Party Computation”.

V. РЕАЛИЗАЦИЯ

Демо версия данного концепта доступна по ссылке `https://github.com/rusyaew/PHCM_Salieri`, однако она пока еще находится в разработке и упускает часть функциональности.

[1] Craig Gentry. 2009. Fully homomorphic encryption using ideal lattices. In *Proceedings of the forty-first annual ACM symposium on Theory of computing* (*STOC '09*). Association for Computing Machinery, New York, NY, USA, 169–178. DOI:https://doi.org/10.1145/1536414.1536440

[2] Zihao Shan, Kui Ren, Marina Blanton, and Cong Wang. 2018. Practical Secure Computation Outsourcing: A Survey. *ACM Comput. Surv.* 51, 2, Article 31 (June 2018), 40 pages. DOI:https://doi.org/10.1145/3158363

[3] Damgård, Ivan Fazio, Nelly Nicolosi, Antonio. (2006). Non-interactive Zero-Knowledge from Homomorphic Encryption. 41-59. 10.1007/11681878_3.

[4] Holst, A. 2020. Global public cloud computing market 2008-2020. https://www.statista.com/statistics/510350/worldwide-public-cloud-computing/