Highlights

Salieri: transparent “root-of-trust” chips protocol with a fast pseudo-homomorphic general-sense platform for any trusted computing problems

Gleb Rusyaev

- Design of the protocol for more transparent and decentralized trusted computing
- Theoretical approach to trusted computing
- Discussing steganography data-leak prevention in handshakes

Salieri: transparent “root-of-trust” chips protocol with a fast pseudo-homomorphic general-sense platform for any trusted computing problems

Gleb Rusyaev[a]

[a]*Letovo School, Moscow, Russia*

ARTICLE INFO

Keywords:
trusted computing
open source
protocol design
cybersecurity

ABSTRACT

This paper proposes the unified protocol for “root-of-trust” chips. While allowing execution of secret programs with both secret and public inputs/outputs in isolated environments (called containers), it also proposes the concept of oracles. Oracle is a container with public source code and means to prove that result was obtained by the specific oracle that was executed on specific hardware (“root-of-trust” chip). These features allow oracles to be a fast pseudo-homomorphic general-sense platform for any trusted computing problems including but not limited to: secret smart contracts, transparent and isolated privacy servers, DRM solutions, and trusted cloud computing.

1. Introduction

The solution to the problem concerning trusted computations in a physically accessible environment is one of the general goals in modern cryptography and cybersecurity. All existing solutions to this problem can be split into two groups: mathematical and hardware ones.

While mathematical solutions, such as “Fully Homomorphic Encryption” [1] are impractical for most applications [2], hardware solutions lack trust, because “security through obscurity” design not only prevents a user from leaking data but also prevents the user from security analysis of these solutions.

In order to create “root-of-trust” chips without the previously mentioned problems, we proposed series of recommendations for such chips as well as unified protocol, allowing not only more transparent execution of secret programs with both secret and public inputs/outputs (containers), but also introducing the concept of oracles – containers with public source code and means to prove that result was obtained by the specific oracle that was executed on specific hardware (“root-of-trust” chip). In order to give an example of such an oracle, this paper contains the solution to the “Secure-Multi Party Computation” problem proposed by Andrew Yao.

Conventions:

container Combination of a binary representation/source code (if it’s an oracle) of a program and specific metadata, such as maximum input size, creation date, etc.

HSL Hardware Security Level is a main part of PHCM actually producing trusted computing.

instance While a container can be treated as a class in the OOP paradigm, an instance is its object. It consists of the container encrypted with HSL and platform keys, as well as container developer and platform signatures.

LSL Low Security Level is an appendix to HSL that stores encrypted instances. It can be emulated on user-side and doesn’t require advanced secure policies..

oracle Special kind of containers with public source code, support of source code validation, and the means to prove that result was obtained by executing specific oracle on specific hardware.

PHCM Pseudo-Homomorphic computation module is a “root-of-trust” chip with some restrictions and specific software set up to fulfill Salieri protocol and protect the user against security threats.

platform While sometimes we refer to a platform as a foundation for creating other software, we also may refer to it as an all “root-of-trust” chip produced by some party. The Exact definition should be clear from the context.

RSA_AES It’s the combination of RSA and AES algorithms: data AES-encrypted by random random key, then this key is encrypted using RSA.

secure channel A secure channel is a communication channel that uses a combination of RSA and AES algorithms to ensure data secrecy and safety.

2. Platform requirements

In order for any “root-of-trust” chip to become recognizable as a platform by Salieri users, producing party should ensure:

gleb@gleb.tk (G. Rusyaev)
ORCID(s): 0000-0002-4074-1116 (G. Rusyaev)

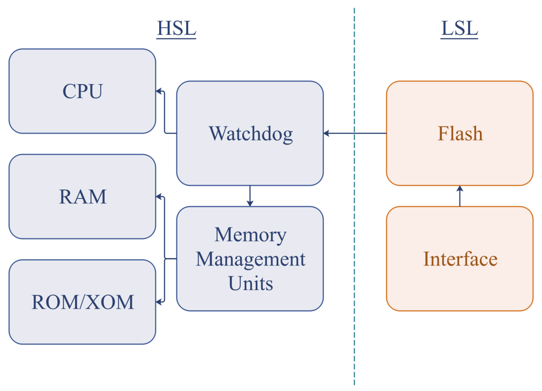

Figure 1: General topology of the "root-of-trust" platform. XOM/ROM is eXecution Only Memory/Read Only Memory. Memory Management Units are used to control memory access and imply permissions.

1. It doesn't require direct internet connection, disk access, nor have access to the main processor, so the user can be sure that it doesn't leak his data
2. It supports Salieri protocol
3. It properly isolates running instances

Of course, we need to consider a situation where the producing party is malicious. In order to prove previous requirements, we will be relying on reputation institute among Salieri users. (it is possible to proof that specific party is malicious)

Since our protocol unifies all platforms, we may treat any individual chip as an abstract party consisting of HSL. Its producer should have a private RSA key β, all platform chips should share the same key α and individual HSL should have unique random key γ generated on creation.

Salieri protocol can successfully function as long as HSL knows private keys γ, α and public key β, producer party knows all public γ keys of related HSL's, as well as public key α and private key β.

On the provided picture, we can see what's the example design of the platform.

3. Secret computation

In order to achieve computation that both secret from platform and user, we need to decrypt and execute instances inside of HSL. First, we need to check β signatures, then decrypt using α, γ, user keys. Since all keys required to produce such an outcome are only present inside of the "root-of-trust" chip, we will consider it secure (full security proof can be found further).

3.1. Receive method

Let $\aleph_{id}$ be desired instance of the container with specific id, we can just load it into LSL (and since LSL can be emulated by user, we may just download it on our disk) or add an additional encryption level in order to prevent PHCM hijacking, such as individual keys for each containers known only to user.

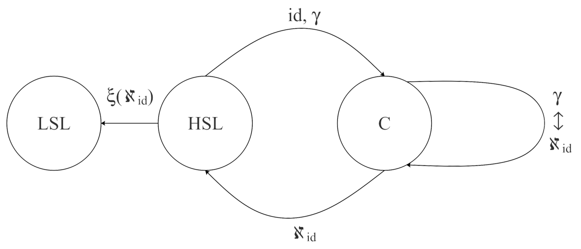

Figure 2: HSL receives container $\aleph_{id}$ from server C by id using it's key γ. Then it encrypts container with user-defined encryption key and transfers it into LSL.

Let's also define ξ as an function, such that:

$$\xi(a) = \begin{cases} \text{a} & : \text{if key isn't specified} \\ \texttt{AES(a, key)} & : \text{if key is specified} \end{cases}$$

Then, as long as we can get $\aleph_{id}$ from any source, we will be able to decrypt it inside of PHCM. Of course, since $\aleph_{id}$ aren't considered to be secret information, then attacker may receive $\aleph_{id}$ of another person.

Lemma 1. *$\aleph_{id}$ is secure, thus the information obtainable by knowing it doesn't allow third-party to read or alter source code.*

PROOF. Let's clarify what information are stored inside of $\aleph_{id}$:

1. `RSA_AES(container,` α, γ`)`: Public keys (2048-bit) can be theoretically used to recover the private key, yet this operation demands enormous computational resources. Besides, chosen cipher-text attacks are quite unlikely to succeed because in the terms of RSA algorithm – our cipher-text is a fixed-size AES key and we are using two keys instead of one.
2. `RSA_SIGN(...,` β`)`: RSA signature for previously encrypted container consists of hash and digital signature. Beta-key attacks are unlikely to succeed due to the computational complexity of such tasks. Collision-attacks will probably fail because any found collision must be correctly decrypted with AES, while the key for its decryption should correspond to the RSA decrypted session key.

Thus, information obtained from $\aleph_{id}$ is unlikely to give any valuable clues leading to reading or altering the source code.

3.2. Execute method

While we don't encourage unrestricted communication between HSL and the outside world and enforce it to pass through the user if necessary, there are still some risk factors present even if communication is transparent.

One of them is security handshakes requiring random data to be sent. The attacker may consider using One-Time-Pad cipher with predefined keys (same entropy) dependent

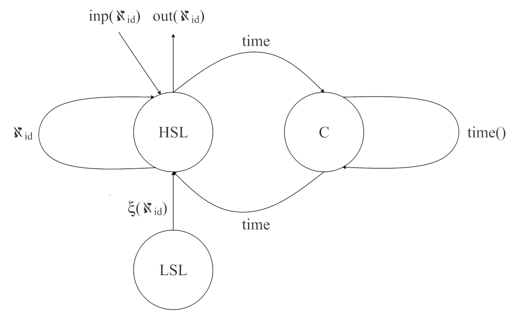

Figure 3: HSL receives user-encrypted container $\aleph_{id}$ from LSL, performs full decryption, then it request and checks timestamps provided by the server. If timestamps match the period of licence, HSL executes such container by creating two file descriptors: `inp`($\aleph_{id}$) (stdin), `out`($\aleph_{id}$) (stdout)

on some common variable available to both sides of communication to send hidden data. For example, timestamp value has 2^{64} possible values (allowing 2^{64} unique keys to be predefined). Thus, 8 bytes of the hidden information can be transferred in one message. Of course, by finding other common variables, we may transfer even more information, ultimately covering all 2^{8k} values (k - the number of bytes in a randomly generated handshake message). Therefore, we should select the smallest number k, while still preventing reuse attacks of any kind (if two messages are sent in the same second, they should have different numbers). In order for a user to have both security and privacy, we will consider using any $5 <= k$ (allowing 10^{12}+ simultaneous requests). Also, since k can vary it would be harder for the attacker to use predefined keys and transfer hidden information.

As in the previous example, the figure doesn't show the "bare" execute request, due to it being very straightforward. Yet, it rather reveals the more advanced version of such algorithm, acquiring instance from LSL, time from an authorized source, and executing only if it satisfies some logical expression.

4. Oracles

Oracle is a special public container, that is executed in special manner in order to ensure:

1. Oracle is open-source and determinable
2. Oracle output provides means to prove that result was obtained by the specific oracle that was executed on specific platform
3. Oracle is isolated

4.1. Proof of Source

This is achieved by explicitly giving to HSL, not the compiled binary, but the source code that would be compiled in place sent to the user, and then executed. In order to achieve transparency, memory safety, and protection against various exploits leaking oracle memory, we will be using Golang as an oracle language, instead of C++ or C. HSL returns compiled source to the user in order for him to check the HSL compiler (that should be also published) for any possible undocumented features and backdoors.

Thus, the user may notice backdoor directly in HSL compiler source code or find out about non-matching compiled bytecode on his side (with HSL compiler). If HSL uses both patched and unpatched compilers in order to deceive users, then we may look for indirect clues about oracle compilation such as non-matching data, too fast or too slow execution time, power consumption analysis, and so on.

4.2. Proof of Result

This proof is required in order to ensure that data were obtained by some pair of hardware identifier and oracle identifier. Let D be our potential output data, s – unique randomly-generated oracle session identifier, and t – current UNIX timestamp difference from the start of oracle execution (it should provide an order of outputs).

Thus, in order to prove given result, HSL calculates:

$$\texttt{RSA\_SIGN([D, s, t],}\ \alpha,\ \gamma)$$

4.3. Proof of Isolation

This proof comes from the definition of container instances running on some HSL. As long as we emulate a separate processor for any container, we are basically invincible to jail and container escape attacks. Yet of course, it may be rather inefficient, so every platform would decide for themselves how exactly isolation should be enforced.

4.4. Oracle bootstrapping algorithm

Algorithm 1 Execute oracle **O**

```
Require: O, m (O - oracle defined by it's source code, m - magic value required for handshake)
  $\mathbf{O}_h \leftarrow$ sha256 (O)
  $\mathbf{O}_c \leftarrow$ compile (O)
  session ← rand()
  out(RSA_SIGN([$\mathbf{O}_c$, $m$, session], $\gamma$, $\alpha$))
  instance ← exec_with_output($\mathbf{O}_c$)
  while true do
    if not(instance.online()) then
      return 0
    end if
    request ← instance.get_request()
    if request.type = INPUT then
      input ← ask_input()
      instance.input(input)
    else if request.type = OUTPUT then
      output ← request.output
      out(RSA_SIGN([output, session,
      instance.runtime], $\alpha$, $\gamma$))
    end if
  end while
  return 0
```

In this particular bootstrapping algorithm, m is present because of the need to make a handshake by the end of which, user can be sure that running oracle has specific bytecode and session number.

4.5. Oracle for sMPC

Oracles can be used as an general-sense platform for trusted computing tasks. In order to demonstrate oracle possibilities on the specific example, we proposed oracle-based solution to sMPC problem. In order to do so, we need to define $\mathfrak{C}_{I,J}(i,j)$ as the secure tunnel between subjects I and J, using keys i, j respectively.

Algorithm 2 Oracle sMPC $: O(n) + O(f)$

```
func ← inp()
parties_am ← inp()
parties ← $\emptyset$
ok ← genkey_via_handshake()
for $i = 1 \ldots$ parties_am do
    user_pubkey ← inp()
    parties = parties ∪ {user_pubkey}
end for
out (ok$_e$, alpha$_e$, gamma$_e$, beta$_e$)
data, proofkeys ← $\emptyset$
for $i = 1 \ldots$ |parties| do
    $T$ ← establish ($\mathfrak{C}_{\mathbf{O},i}$(ok, parties[i]))
    magic_val ← $T_{i\to\mathbf{O}}$(inp())
    salt ← rand()
    out ($T_{\mathbf{O}\to i}$(func, magic_val, parites[i], salt))
    salt', proofkey, value ← $T_{i\to\mathbf{O}}$(inp())
    if salt' $\neq$ salt then
        return 0
    end if
    proofkeys = proofkeys ∪ {proofkey}
    data = data ∪ {val}
end for
result ← exec ($f$, data[1], data[2], ..., data[n])
output ← $\mathfrak{s}'$((result, func), proofkeys[0], ...,
    proofkeys[n])
out (output)
```

Present algorithm controls what HSL do, yet we also need to ensure that all the users behave in a specific algorithm.

1. Users collectively select one person with trusted platform PHCM, let's denote that person by the letter U.
2. Selected user U launches SMPC oracle. All further communication between users and oracle would be passing through the user U.
3. All users receive oracle session, bytecode from oracle and (ok$_e$, alpha$_e$, gamma$_e$, beta$_e$). They proof this information and move to the next step.
4. After the handshake, using ok key, each user may establish secure channel
5. Salt is generated and sent to all the users
6. Each user sends randomly generated constant and receives (func, magic_val, parites[i], salt) from the oracle
7. If the user is satisfied with oracle's answer, then he sends (salt, proofkey, value), where the "proofkey" is the unique randomly-generated RSA, set up in order to achive proof of argument usage on the computation stage.
8. Oracle signs the computing outcome by all user's proofkeys and function binary representation

Lemma 2. *sMPC conditions are met while executing sMPC oracle, thus no party obtains secret values of other parties and no party receives the incorrect result.*

PROOF. Let the conditions of the protocol be not met, therefore exists some party that obtained secret values of other parties or received incorrect result.

Suppose that some party P received result (fakeres, f), but function f should have resulted in a different value (res, f). Apparently, the values of some users were modified, but then there is such a party, which signature was not applied to the final result.

Also, let's suppose that some participant P acknowledged the secret value of the other party, it's possible when P is U, i.e. host of the oracle, who sent malicious function f' into the oracle. However, on stage 6, all users received func from the oracle. From which party can conclude that U is illegitimate and stop further interactions with him (before sending a secret value).

The possibilities of oracles do not end there. The level of trust arising within the PHCM makes it possible to effectively implement solutions to other problems that can be solved with homomorphic encryption (Zero-Knowledge Proof [3], ...) and standard tasks of crypto processors, making it, in the matter, a general-sense platform for operations with the secret information.

5. Applications

In its basic state, the protocol can be applied to trusted cloud computing that became especially relevant after the growth of cloud computing market capitalization from 6 to 236 billion USD [4] and transparent DRM platforms.

Using oracles, we can perform any task achievable by the means of homomorphic encryption, because proof of source, proof of result and proof of isolation implies that this is indeed possible. For example, secret smart contracts, transparent and isolated privacy servers, etc.

While traditional "root-of-trust" protocols are proprietary and rely solely on the institute of reputation with a lack of means to analyze a protocol itself, Salieri criteria ensure that the trust necessary for the functioning system is established among users. Also, decentralization makes it easier for software developers and users to transfer from one platform to another and raises concurrency among Salieri platforms.

6. Implementation

Basic demo realization for Salieri protocol are available by the link, yet it sill may lack some advanced features and

intended only for demonstration reasons. PoC (model) implemented using Python programming language and libraries such as Flask and PyCryptoDome.
`https://github.com/rusyaew/PHCM_Salieri`

7. Further work

Some modifications of Salieri protocol may be beneficial for software marketplaces, for example we can use the combination of Blockchain and BitTorrent to store the encrypted containers, therefore removing human factor from the marketplace and providing means to the software developer to switch between various ones. This can be implemented using Ethereum Smart Contracts that store magnet links.

8. Related work

As for the related work, we may consider ”OpenTitan“ [5] project about open-hardware based RoT chips. While ”OpenTitan“ discusses only individual platform (in terms of Salieri Protocol), we design a protocol for any chips covering basic security requirements, thus allowing more diverse and personalized solutions.

References

[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. URL:https://www.statista.com/statistics/510350/worldwide-public-cloud-computing/

[5] "OpenTitan". OpenTitan. 2019. URL:https://opentitan.org/

Gleb Rusyaev is a sophomore year student at Letovo School in Russia.