E2EE Metaverse

Omar Huseynov
saccharineboi@gmail.com

Abstract: Ever since its emergence the Internet has revolutionized the way people interact and share information with each other. However certain limitations in the capabilities of current protocols hinder the development of more immersive and private experiences. The Internet in its current form is lacking mainly in two respects: immersion and privacy. By lack of immersion I mean the continuing use of 2D interfaces in place of 3D virtual environments that offer more detail and better user experience. By lack of privacy I refer to the at best incomplete and at worst nonexistent safeguards to protect user privacy. In order to solve these problems I propose *Hades*[1] - a Metaverse protocol designed to enable richer experiences and better privacy for its users. Hades employs a modified version of the glTF standard for efficient 3D file exchange and strong cryptography as a means to better privacy. I call the virtual worlds that make use of the Hades protocol *Hadean Worlds*, which together form the *Hadean Metaverse*.

Keywords: Metaverse, Privacy, Cryptography

Introduction

Although the term Metaverse originates from the science fiction novel "Snow Crash" (Stephenson, 1992), the core idea that it represents – transcendental worlds inhabited by agentic beings – is much more ancient. Sundry items found in burials in the Upper Paleolithic indicate belief in the afterlife (Petru, 2019: 6-13) – an ancestral precursor to the modern notion of Metaverse. Pantheons and spirit worlds in ancient mythologies are also early examples (Black et al., 1992; Fry, 2017). Modern religious belief in the existence of supernatural worlds has persisted in part due to the continuing influence of the Abrahamic doctrines (Somov, 2017; Davidson, 1973: 33). Altered states of consciousness induced by psychoactive substances are additional means of interacting with hallucinated beings in otherworldly environments (Clottes et al., 1998)[2]. We are interested in artificially reproducing these experiences with the aid of computers[3].

Dreams are naturally occurring states of consciousness in which the user is usually unaware that they are dreaming (Lévy-Bruhl, 2020). Such unawareness can be undesirable within the Metaverse. Henceforth I define *Redfield's first problem*[4] to describe a case in which the user is unable to distinguish between the Metaverse and the real world[5]. I define *Zhuangzi's first problem*[6] to describe a case in which the user is unable to distinguish between the different identities they own or represent themselves as in the Metaverse and the real world. Furthermore I define a stronger variant of Redfield's first problem to refer to a case where the user is unable to distinguish between the different virtual worlds within the Metaverse. Note that the user may be unable to distinguish between the real world and the Metaverse, or between different virtual worlds within the Metaverse due to psychosis (Arciniegas, 2015) or other naturally-occurring mental disturbances, but in this paper I assume that the user is lucid. A stronger variant of

[1] Hades protocol is named after the ancient Greek God of the Underworld, see *Mythos* (Fry, 2017).
[2] c.f. Shamanism (Diószegi et al., 2024) and animism (Park, 2023).
[3] Modern Metaverse is a form of *computational shamanism* – a retrogression toward an ancestral environment where agentic "spirits" embodied by avatars inhabit transcendental worlds.
[4] Redfield's first problem is named after Cooper Redfield (played by Wyatt Russell) from Black Mirror episode "Playtest" (Netflix, 2016).
[5] The reality may be a simulation (Bostrom, 2003), but this ambiguity has no effect on the protocol. See Appendix B for a discussion on the consequences *of* the Hades protocol *for* the simulation hypothesis.
[6] Zhuangzi's first problem is named after the author of "The Butterfly Dream" (庄子, 476–221 BC).

Zhuangzi's first problem is a case where the user is unable to distinguish between the different identities they own or represent themselves as within the Metaverse. Note that the user may not be able to distinguish between the different identities they own or represent themselves as in the Metaverse and the real world, or within different virtual worlds in the Metaverse due to dissociative identity disorder (Peters et al., 2017) or other naturally-occurring mental disturbances. But in this paper I assume that the user is undisturbed. I define third-person variants of these two problems as secondary problems. Thus *Redfield's secondary problem* describes a case in which a third-party is able to find out whether the user is in the real world or the Metaverse. A stronger variant of Redfield's secondary problem refers to a case when a third-party is able to find out which virtual world the user currently resides in within the Metaverse. *Zhuangzi's secondary problem* has to do with the case where a third-party is able to distinguish between the different identities the user owns or represents themselves as in the Metaverse and the real world. A stronger variant of Zhuangzi's secondary problem refers to a case where a third-party can distinguish between the different identities the user owns or represents themselves as within the Metaverse. First and secondary problems together with their stronger variants form the *Redfield-Zhuangzi Matrix*:

	No First Problem (I)	**Redfield's First Problem (II)**	**Stronger Variant of Redfield's First Problem (III)**	**Zhuangzi's First Problem (IV)**	**Stronger Variant of Zhuangzi's First Problem (V)**
No Secondary Problem (A)	A-I	A-II	A-III	A-IV	A-V
Redfield's Secondary Problem (B)	B-I	B-II	B-III	B-IV	B-V
Stronger Variant of Redfield's Secondary Problem (C)	C-I	C-II	C-III	C-IV	C-V
Zhuangzi's Secondary Problem (D)	D-I	D-II	D-III	D-IV	D-V
Stronger Variant of Zhuangzi's Secondary Problem (E)	E-I	E-II	E-III	E-IV	E-V

Figure 1: Redfield-Zhuangzi Matrix

The matrix above is useful in classifying different kinds of Metaverse applications. The red columns (IV and V) are associated with Metaverse applications where users are artificially dissociated from their previous and/or real selves. The fictional video game "Roy: A Life Well Lived" from "Rick and Morty" (Adult Swim, 2015) is an example[7]. The yellow columns (II and III) refer to Metaverse applications in which the users are either unaware that they are within the Metaverse, or they cannot distinguish between the different virtual worlds within the Metaverse. The horror game from Black Mirror's "Playtest" episode (Netflix, 2016) is an example. The blue column (I) contains Metaverse applications where the users are sovereign[8]. The Metaverse of the "Ready Player One" (Cline, 2017) is in the I column.

While columns represent levels of awareness, rows represent levels of privacy. Rows D and E represent Metaverse applications where users can be easily identified. This happens when lack of encryption allows a third party to infer a user's identity based on their digital footprint. Rows B and C represent Metaverse applications where the users can be easily located. This happens when the user is forced to share their identity across distinct virtual worlds. Row A represents Metaverse applications where the users can selectively reveal their location and identity to those that they trust.

[7] Interestingly Rick seems to be unaffected by the device. Also, cf. episode "Puhoy" of "Adventure Time" (Cartoon Network, 2013).

[8] Sovereignty holds in cases where users are manipulated by other agents. Unawareness and dissociation must be induced by the application.

Traditional Metaverse applications employ a mode of operation where users are authenticated by trusted servers where:

1. The user shares their identity across virtual worlds.
2. No encryption is provided to remedy the privacy implications of computing on unencrypted data.

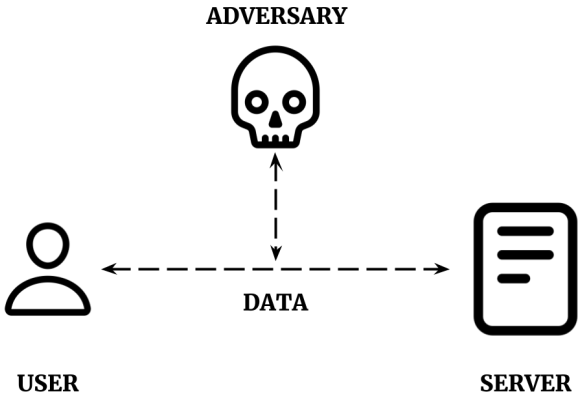

Figure 2: Traditional Metaverse Applications

In Figure 2, there is a bi-directional transfer of **DATA** between the **USER** and the **SERVER** over an insecure channel. **DATA** is encrypted during transit, and upon its arrival to the **SERVER**, it is decrypted and computed upon. The goal is to prevent the **ADVERSARY** from reading and writing to **DATA**. This is achieved by securing the connection between the **USER** and the **SERVER** via a cryptographic protocol like TLS (Rescorla, 2018). Each **SERVER** gains its competitive advantage by the large amounts of **DATA** it collects from each **USER** and its ability to process it.

Hades protocol drops the distinction between the **SERVER** and the **ADVERSARY**:

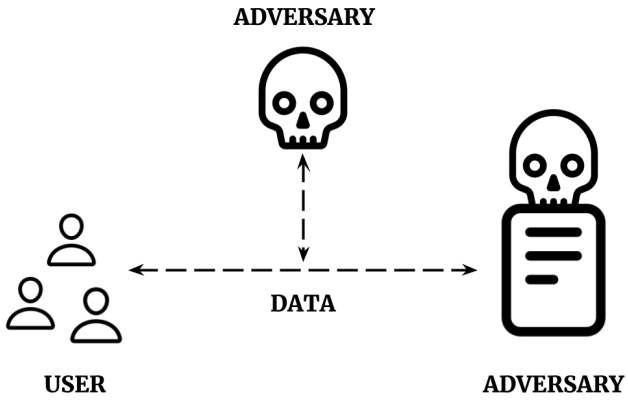

Figure 3: Hades protocol

In Figure 3, there is a bi-directional transfer of **DATA** between one of the virtual identities owned by the **USER** and the **ADVERSARY** over an insecure channel. **DATA** is encrypted during transit, and upon its arrival to the **ADVERSARY**, it *remains* encrypted. An authenticated **ADVERSARY** is prevented from reading *from* **DATA**, while an unauthenticated **ADVERSARY** is also prevented from writing *to* **DATA**[9]. Each **ADVERSARY** gains its competitive advantage by its ability to process varying types of encrypted **DATA**.

The goal of the Hades protocol is to enable *adversarial* machines to become arbiters *ignorant* of that which they *arbitrate*. I call this *Programmable Blind Arbitration*, or **PBA** for short.

[9] Unless otherwise specified, in this paper all mentions of **ADVERSARY** refer to the authenticated **ADVERSARY**. Hadean adversaries are honest-but-curious (Paverd et al., 2014), sometimes also called semi-honest.

Hades solves Redfield's first problem and its stronger variant by assigning a universally unique identifier (UUID) to each virtual world, by explicitly notifying the user when there is a transition between two distinct virtual worlds[10], or between a virtual world and the real world[11], and by disabling blending[12]. Together they form the first three epistemological guarantees of the Hades protocol:

1. Each virtual world is unique and different from each other and the **NULL WORLD** *(uniqueness)*.
2. Each transition between two distinct virtual worlds or between a virtual world and the **NULL WORLD** is made explicit *(explicitness)*.
3. No blending is allowed between a virtual world and the **NULL WORLD**, or between two distinct virtual worlds *(separateness)*[13].

Hades solves Zhuangzi's first problem and its stronger variant by assigning a UUID to each virtual identity, by explicitly notifying the user when there is a transition between their two distinct virtual identities[14], or between their virtual identity and real identity[15], and by disabling blending[16]. Together they form the next three epistemological guarantees of the Hades protocol:

4. Each virtual identity is unique and different from each other and the **NULL IDENTITY** *(uniqueness)*.
5. Each transition between two distinct virtual identities or between a virtual identity and the **NULL IDENTITY** is made explicit *(explicitness)*.
6. No blending is allowed between a virtual identity and the **NULL IDENTITY**, or between two distinct virtual identities *(separateness)*[17].

Hades solves Redfield's secondary problem and its stronger variant by *decoupling* the **USER** from the virtual identities that they own. There is no relationship between distinct virtual identities other than the fact that they may belong to the same **USER**. The real identity of the owner of a given virtual identity, i.e. **USER**, is withheld from the **ADVERSARY**[18].

Hades solves Zhuangzi's secondary problem and its stronger variant by its use of end-to-end encryption. E2EE enables each **USER** to share their **DATA** only and only with those that they trust. This prevents the **ADVERSARY** from inferring the identity of the **USER** based on their **DATA**. Hades protocol achieves this by use of *Local* and *Shared Programmable States*, which we define next:

Let $\Omega_1, \Omega_2, \ldots, \Omega_n$ be virtual identities and let Θ be a *Shared Programmable State* (**SPS**) which is a set of some arbitrary number of states $\Phi_1, \Phi_2, \ldots, \Phi_n$ such that each state Φ_i denotes some state variable relevant to the **SPS**, e.g. position vector of the virtual identity Ω_i at time t. Let $Z_1, Z_2, \ldots, Z_n$ be a list of rules that Θ must obey, e.g. y value of the position vector cannot be negative. Let $\mathcal{H}$ be a function that, given $\Omega_1, \Omega_2, \ldots, \Omega_n$, can compute the given state transition ($\Theta_i \rightarrow \Theta_k$) valid under the rules $Z_1, Z_2, \ldots, Z_n$ while having access to neither Θ_i nor Θ_k. I call $\mathcal{H}$ the *Hadean function*. $\mathcal{H}$ allows the **ADVERSARY** to compute the state transition ($\Theta_i \rightarrow \Theta_k$) while knowing neither Θ_i nor Θ_k. This is how the Hades protocol achieves **PBA**[19]. $\mathcal{H}$ function is defined as:

$$\mathcal{H}_{\mathbf{SK}} : Enc(\Theta_i) \rightarrow Enc(\Theta_k)$$

where *Enc* is a function that encrypts the given Θ under the secret key **SK**. I assume that there is a corresponding *Dec* function such that:

[10] The UUIDs of virtual worlds are constant – one can only create or destroy virtual worlds.
[11] The real world is assigned the UUID of 0, i.e. NULL. In this paper the real world may also be referred to as the **NULL WORLD**. Updates to the real world don't change its UUID.
[12] This has the effect of disabling augmented reality (AR).
[13] This refers to augmented reality (AR) where the elements of some virtual world or worlds are overlaid on top of or mixed with a different virtual world or the real world.
[14] The UUIDs of virtual identities are constant – one can only create or destroy virtual identities.
[15] The real identity is assigned the UUID of 0, i.e. NULL. In this paper the real identity may also be referred to as the **NULL IDENTITY**, or simply the "user identity". Updates to the real identity don't change its UUID.
[16] This has the effect of disabling augmented identity (AI).
[17] This refers to augmented identity (AI) where the elements of some virtual identity or identities are overlaid on top of or mixed with a different virtual identity or the real identity.
[18] See Appendix A to see cases where this can be bypassed.
[19] When Z = ∅, we get a **NULL SPS**, which is useful in cases where no moderation is necessary.

$$\Theta \cong Dec(Enc(\Theta))^{20}$$

Suppose Ω_1 = Ἀχιλλεύς is playing chess with Ω_2 = Πάτροκλος. Chess is a turn-based game with a predefined list of rules (Z_1 = "bishops can only move diagonally", Z_2 = "pawns can only move forward", etc.) and win/lose conditions[21]. Whenever Ἀχιλλεύς or Πάτροκλος makes a move, there is a transition in the state of the game. The role of the $\mathcal{H}$ function is to compute a valid state transition ($\Theta_i \rightarrow \Theta_k$) while knowing neither Θ_i nor Θ_k. $\mathcal{H}$ allows Ἀχιλλεύς and Πάτροκλος to play a game of chess without revealing the positions of the pieces nor the result of the move to the **ADVERSARY** in whose machine it is executed.

In order for virtual identities to own and control avatars and other interactable and non-interactable objects and share them with each other the Hades protocol offers *Local Programmable States* (**LPS**).

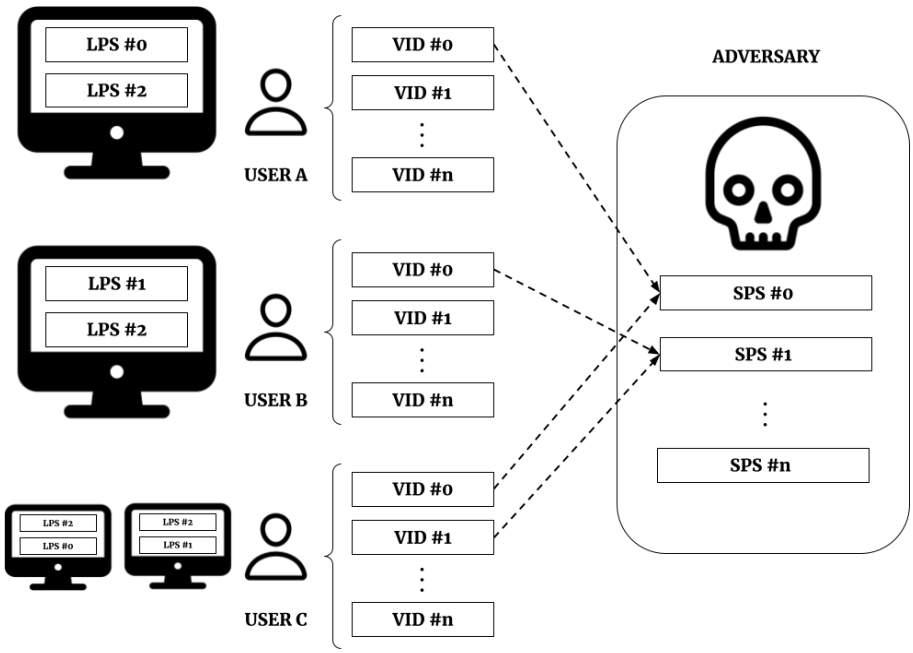

Figure 4: Local Programmable States

In Figure 4, **LPS #0** is the avatar of **USER A**, **LPS #1** is the avatar of **USER B**, and **LPS #2** is the avatar of **USER C**. Naturally **LPSs** contain not only the data required to render the avatars, but also scripts to add programmable functionality. In the case of an avatar, the script may listen for user input and run an animation. The scripts are run in the owner's machine. The data contained within **LPSs** are used by the client to render the model, play sounds, etc. An **LPS** is always owned by a VID. There are no orphan **LPSs**. Each **LPS** contains a ***state*** that is updated by the owner of the **LPS** and shared with others in the same **SPS** session. The **ADVERSARY** holds a table linking each **LPS** to a corresponding VID, but the **ADVERSARY** doesn't know what's *inside* the **LPS** – it simply listens for ***state*** changes from its owner and then broadcasts the new ***state*** to everyone else within the same **SPS** session. The contents of **LPSs** are hidden from the **ADVERSARY**. Only the owner of an **LPS** is allowed to write to its ***state***, but anyone within the same **SPS** session can read it. Each VID is allowed to own more than one **LPS**, but each VID must own at least one **LPS**.

Note that in Figure 4 **USER C** is running two clients simultaneously. If these **SPS** sessions are chess games, then **USER C** is playing two E2EE chess games with users **A** and **B** at the same time with different VIDs. Even if **USER C** uses the same **LPS** for each of these games, because they belong to different sessions, the **ADVERSARY** is unable to link them together[22].

[20] For this paper it is sufficient that the decryption of *Enc(Θ)* is *approximately* equal to *Θ*.
[21] For the sake of simplicity I will ignore other possible outcomes like draws and stalemates.
[22] One could argue that the **ADVERSARY** may link the two different VIDs of **USER C** based on the IP address that they most likely share, but IP addresses alone are insufficient to resolve the ambiguity. See Appendix A for further discussion.

I categorize Metaverse applications with sovereign users as *X-I* where *X* corresponds to one of the rows { *A, B, C, D, E* } and *I* refers to the leftmost column of the RZ matrix. Thus the Hadean Metaverse is a *Class X-I Metaverse*.

Parameters

Implementations must decide on the following parameters:

Name	Definition
info	An ASCII string identifier for the application
hash	A cryptographically-secure hash function, e.g. BLAKE2b (Aumasson et al., 2013)
kdf	A key derivation function that generates a key from a given password and random nonce, e.g. Argon2id13[23] (Biryukov et al., 2015)
hkdf	HMAC-based extract-and-expand key derivation function, e.g. HKDF-SHA256 (Krawczyk et al., 2010)
pqkem	A post-quantum key encapsulation mechanism, e.g. Crystals-Kyber-1024 (Bos et al., 2017)
pqdss	A post-quantum digital signature scheme, e.g. Crystals-Dilithium-5 (Ducas et al., 2017)
aead	A scheme for authenticated encryption with associated data, e.g. ChaCha20-Poly1305[24] (Nir et al., 2018)
kex	An Elliptic Curve based key exchange mechanism, e.g. X25519 (Josefsson et al., 2018)
dss	An Elliptic Curve based digital signature scheme, e.g. Ed25519 (Ibid.)
fhe	A fully-homomorphic encryption scheme, e.g. CKKS (Cheon et al., 2017)[25]

I have programmed two separate implementations: one for the client and one for the server, which I've named Charon[26] and Minos[27], respectively.

[23] Charon supports both Argon2i13 and Argon2id13, but uses Argon2id13 by default.
[24] Both Charon and Minos use the XChaCha20-Poly1305 variant (Arciszewski, 2020).
[25] Both Charon and Minos use the full RNS variant (Cheon et al., 2018).
[26] Charon is named after Χάρων – the psychopomp and the ferryman of the ancient Greek Underworld.
[27] Minos is named after Μίνως – king of Crete, son of Zeus and Europa.

Cryptographic Notation

Name	Definition
X **||** ***Y***	Concatenation of two byte sequences ***X*** and ***Y***
(Y) = HASH(X)	Output ***Y*** of the chosen cryptographically-secure hash function where ***X*** is an arbitrary-long sequence of bytes
(SK) = KDF(P, S)	Output of the chosen key derivation function where ***P*** is the password, ***S*** is the salt, and ***SK*** is the secret key
(SK) = HKDF(IKM, S, CTX)	Output of the chosen HMAC-based extract-and-expand key derivation function where ***IKM*** is the input key material, ***S*** is the salt, and ***CTX*** is the ASCII string description of the secret key ***SK***
(CT, SS) = PQKEM-ENC(PK)	Output of the chosen post-quantum key encapsulation function, where ***SS*** is the shared secret encapsulated by the ciphertext ***CT*** using the public key ***PK***
(SS) = PQKEM-DEC(SK, CT)	Output of the chosen post-quantum key decapsulation function, where ***SS*** is the shared secret encapsulated by the ciphertext ***CT***, and ***SK*** is the secret key
(SIG) = PQDSS-SIGN(M, SK)	Signature ***SIG*** produced by the chosen post-quantum digital signature scheme, given plaintext ***M*** and secret key ***SK***
PQDSS-VERIFY(SIG, M, PK)	Verification (**TRUE** or **FALSE**) by the chosen post-quantum digital signature scheme, given the signature ***SIG***, the plaintext ***M***, and the public key ***PK***
(CT) = AEAD-ENC(SS, M, N)	Output of the encryption function of the chosen AEAD scheme, where ***SS*** is the shared secret, ***M*** is the plaintext, ***N*** is the nonce, and ***CT*** is the ciphertext containing an authentication tag
(M) = AEAD-DEC(SS, CT, N)	Output of the decryption function of the chosen AEAD scheme, where ***SS*** is the shared secret, ***CT*** is the ciphertext containing an authentication tag, ***N*** is the nonce, and ***M*** is the plaintext
(SIG) = DSS-SIGN(M, SK)	Signature ***SIG*** produced by the chosen digital signature scheme, where ***M*** is the plaintext, and ***SK*** is the secret key
DSS-VERIFY(SIG, M, PK)	Verification (**TRUE** or **FALSE**) by the chosen digital signature scheme, where ***SIG*** is the signature, ***M*** is the plaintext, and ***PK*** is the public key
(SS) = DH(PK1, PK2)	Output of the Elliptic Curve Diffie-Hellman function where ***SS*** is the shared secret, and ***PK{1,2}*** are the public keys representing the corresponding key-pairs
(CT) = FHE-ENC(M, PK)	Output of the encryption function of the chosen FHE scheme, where ***M*** is the plaintext, ***PK*** is the public key, and ***CT*** is the ciphertext
(M) = FHE-DEC(CT, SK)	Output of the decryption function of the chosen FHE scheme, where ***CT*** is the ciphertext, ***SK*** is the secret key, and ***M*** is the plaintext
(CT2) = FHE-EVAL(M, CT1, F, HEPUB{1..N})	Encrypted output of the function ***F*** on the plaintext ***M*** and the ciphertext ***CT1***, using public parameters ***HEPUB{1..N}***, resulting in another ciphertext ***CT2***
(CT3) = FHE-EVAL(CT1, CT2, F, HEPUB{1..N})	Encrypted output of the function ***F*** on two ciphertexts ***CT{1,2}***, using public parameters ***HEPUB{1..N}***, resulting in another ciphertext ***CT3***

User Identity

Hadean clients maintain a *User.json*[28] file that contains the following fields:

Name	Definition
version	Version of the protocol encoded as a uint32_t[29] value[30]
kdf	ID of the **KDF** encoded as a uint32_t value[31]
cpu	CPU intensity of the **KDF** encoded as uint64_t
memory	Memory intensity of the **KDF** encoded as uint64_t
salt	Salt value encoded in padded URL-safe base64 string[32]
vids	Array of virtual identities
extensions	Array of extensions used by the client[33]

User.json makes it easy for users to migrate from one client to another. Access to *User.json* is regulated by the OS, where each user can parse only the *User.json* that they own[34].

[28] All human-readable text files are stored in the JSON format (Ecma International, 2017).
[29] The referenced integral data types are those defined in the stdint.h header (part of the C library).
[30] Hades protocol uses semantic versioning as defined in (semver.org) where MAJOR is specified in the most significant 8 bits, PATCH in the least significant 16 bits, and MINOR in the remaining 8 bits.
[31] 0 is Argon2I13 and 1 is Argon2ID13.
[32] This corresponds to the sodium_base64_VARIANT_URLSAFE enum from the libsodium library (libsodium.org).
[33] Each client is free to define its own extensions. For instance, Charon defines *creation_date*, *update_date*, and a bunch more to store parameters such as usage statistics, graphical options, language, security preferences, etc. The client may also offer an option to encrypt such information under a different key to make it inaccessible to other clients.
[34] It is recommended that clients change the permissions of the User.json so that only its owner can read/write it. Clients that run on UNIX should follow the XDG Base Directory Specification (Bastian et al., 2021) and place the User.json file in $XDG_CONFIG_HOME/$CLIENT_NAME.

Virtual Identities

Before users can connect with each other they need to generate their virtual identities. Each virtual identity has the following fields:

Name	Definition
nickname	Human-readable nickname for the VID encoded as a UTF-8 string (Yergeau, 2003)
pqdss_pub	Public key associated with **PQDSS** encoded in padded URL-safe base64 string
pqdss_pri	Encrypted private key associated with **PQDSS** encoded in padded URL-safe base64 string
pqdss_nonce	Nonce value used for the encryption of ***pqdss_pri*** encoded in padded URL-safe base64 string
dss_pub	Public key associated with **DSS** encoded in padded URL-safe base64 string
dss_pri	Encrypted private key associated with **DSS** encoded in padded URL-safe base64 string
dss_nonce	Nonce value used for the encryption of ***dss_pri*** encoded in padded URL-safe base64 string
friends	Array of friends that belong to this virtual identity where each element is a public identity

Public Identities

Users connect with each other by sharing their public identities[35]. Each public identity is generated from a given virtual identity, and its fields are defined below:

Name	Definition
version	Version of the protocol encoded as a uint32_t value
pqdss_pub	Public key associated with **PQDSS** encoded in padded URL-safe base64 string
dss_pub	Public key associated with **DSS** encoded in padded URL-safe base64 string

[35] Public identities can be shared in person, or via an authenticated channel.

Hadean Transmission Format

In order to facilitate efficient exchange of data between users I introduce a new format based on some modifications of the glTF™ 2.0 specification[36] (The Khronos® 3D Formats Working Group[37], 2021), which I have named the *Hadean Transmission Format*, or HTF for short. Below is shown the relations between top-level objects in an HTF asset:

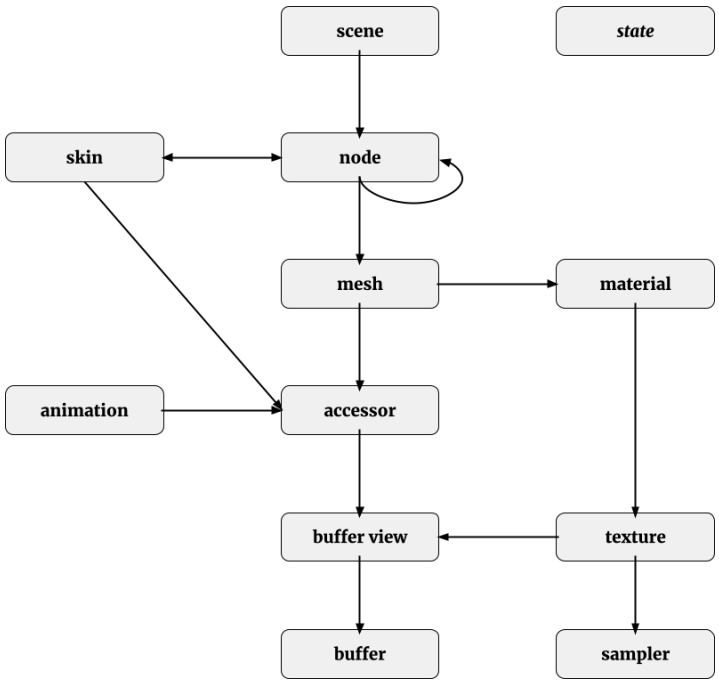

Figure 5: HTF Object Hierarchy

Note that some glTF constructs like cameras and images are removed. Below is a non-exhaustive list of differences between glTF and HTF:

- File extension is changed to .htfl for *Local Programmable States* and .htfs for *Shared Programmable States.*
- Only one scene is allowed. The "scenes" array is removed and the "scene" property contains all the nodes. "scene" property is mandatory.
- "uri" property is removed. Data cannot be embedded inside the JSON. External references are disallowed.
- "source" property in the texture object is replaced with the "bufferView" property.
- "mimeType" property is removed from the texture objects.
- All textures must be stored in the KTX™ 2.0 format[38] (The Khronos® Group Inc., 2024)[39].
- "byteLength" is uint64_t.
- Wherever possible, all OpenGL[40] constants are replaced with their Vulkan[41] equivalents.
- ***state*** construct is added to support *Local* and *Shared Programmable States.*

[36] glTF and the glTF logo are trademarks of the Khronos Group Inc.
[37] Khronos and the Khronos Group logo are registered trademarks of the Khronos Group Inc.
[38] KTX and the KTX logo are trademarks of the Khronos Group Inc.
[39] HTF implicitly requires the "KHR_texture_basisu" extension.
[40] OpenGL® and the oval logo are trademarks or registered trademarks of Hewlett Packard Enterprise in the United States and/or other countries worldwide.
[41] Vulkan is a registered trademark and the Vulkan SC logo is a trademark of the Khronos Group Inc.

All HTF files are binary and are structured as shown below:

magic *(uint32_t)*	**version** *(uint32_t)*	**length** *(uint64_t)*	**script** *(uint8_t[])*	**data** *(uint8_t[])*

Figure 6: HTF File Layout[42]

- **magic**: must be 0x4C465448 for .htfl files and 0x53465448 for .htfs files.
- **version**: this indicates the version of the HTF file format[43].
- **length**: size of the file in bytes (including the header).
- **script**: a block that contains the Lua script (lua.org).
- **data**: a block that contains the actual payload.

script block contains the Lua source file that adds programmable functionality and contains the following fields:

source_length *(uint64_t)*	**source** *(uint8_t[])*

Figure 7: Script Block

- **source_length**: size of the source block in bytes (including the header).
- **source**: Lua source code.

data block contains the actual payload:

json_length *(uint64_t)*	**json** *(uint8_t[])*	**bin_length** *(uint64_t)*	**bin** *(uint8_t[])*

Figure 8: Data Block

- **json_length**: size of the json block in bytes.
- **json**: represents the JSON part of the HTF file.
- **bin_length**: size of the raw data block in bytes.
- **bin**: represents the raw data part of the HTF file.

In addition to the changes specified above, HTF also introduces the ***state*** construct into the glTF specification, the role of which depends on whether the given HTF represents an **SPS** or an **LPS**.

[42] Each HTF file has a 16-byte header.
[43] HTF files use semantic versioning as defined in (semver.org) where MAJOR is specified in the most significant 8 bits, PATCH in the least significant 16 bits, and MINOR in the remaining 8 bits.

Local Programmable States

To support programmable behavior the Hades protocol makes use of *Local Programmable States*, shortened to **LPS**, which are identified by the 0x4C465448 magic number in the first 4 bytes of the header. Additionally, the "`id`" field of the ***state*** object is 0. The goal of the ***state*** object is two-fold:

1. Establish *semantic* relationships between the ***state*** variables and the corresponding properties of the HTF object[44],
2. Send the values of the ***state*** variables from their owners to the other VIDs.

By properties of the HTF object I mean the properties of its children, e.g. its transform, materials, samplers, animations, etc. By ***state*** variables I mean the data that is used by the script to modify the aforementioned properties. The script can influence the properties of the HTF object only and only through these ***state*** variables. Furthermore, the values of the ***state*** variables are sent via the **ADVERSARY** to the other participants of the **SPS** session. Only the owner of an **LPS** is allowed to update its ***state*** variables, everyone else can read them.

The **ADVERSARY** pre-allocates blocks of memory to store the encrypted values of the ***state*** variables[45]. It listens for changes in the ***state*** variables from the owner of the **LPS**, and then propagates them to the other participants of the **SPS** session. The **ADVERSARY** is incapable of *reading* the ***state*** variables, it can only *transfer* them.

State variables are declared within the ***state*** object, where each variable defines the top-level *category* of the object being read/modified, its *index*, and *type*:

```
{
    "state": {
        "id": 0,                    /* Always 0 for LPSs */
        "character_position": {     /* Name of the state variable */
            "category": NODE,       /* Name of the top-level category */
            "index": 10,            /* Its index in the array */
             "type": TRANSLATION    /* Its type */
        },
        "helmet_material": {
            "category": MATERIAL,
            "index": 2,
            "type": PBR_METALLIC_ROUGHNESS
        },
        /* etc. */
    },
    /* other properties here */
}
```

The ***state*** object declares variables that are to be accessible to the script[46]. For each variable we name its *category* which is the name of the top-level object in an HTF file. Then we specify its *index* in the corresponding array, and finally declare its *type*. The type of the ***state*** variable can either be *explicit*, i.e. they can be found by reading the JSON part of the HTF file, or they can be *implicit*, in which case they reference properties that, while not specified in the HTF file, are still assumed to exist:

---

[44] Semantic relationships allow Hadean clients to interpret a ***state*** variable based on its *type* property. For example, if the type is specified as TRANSLATION, then the client knows that it must modify a 3-element array containing floating-point values, if the type is PBR_METALLIC_ROUGHNESS, then it must modify a struct, etc. In memory these properties live inside the instances of the various components attached to the **LPS**, e.g. transform component, render component, particle component, etc. Changing the value of the property thus changes the value in memory, not the actual HTF file.

[45] Hadean adversaries may compete with each other with respect to how much memory they can pre-allocate to their users. They may even charge their users for more memory.

[46] Though not required, it is recommended to use meaningful names when declaring variables, as I have done here.

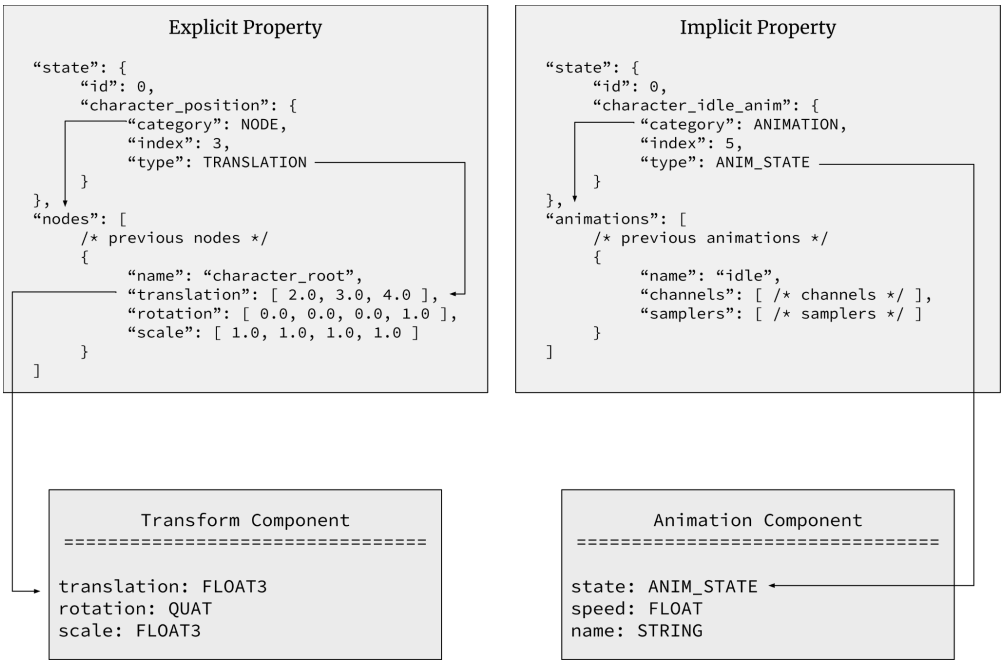

Figure 9: Explicit & Implicit Properties

In Figure 9 the ***state*** variable *character_position* is referencing the *translation* property found in the fourth index of the *nodes* array. Modifying this variable in the script will change the position of the character. The ***state*** variable *character_idle_anim* is referencing the *state* property in the sixth index of the *animations* array, which doesn't exist in the HTF file, but is assumed to exist by the script. Modifying this variable in the script will change the state of the animation[47].

Inside the script there are two functions named *start* and *update*. The *start* function is used to initialize the values of the ***state*** variables and is run only once, whereas the *update* function is run every frame. These are standard constructions that are found in many game engines that game developers are familiar with. Below is an example of a script that modifies the ***state*** variables mentioned in Figure 9:

```
function start()
    character_position = vector3(0.0, 2.0, 0.0)
    character_idle_anim = anim_state.stop
end

function update(dt)
    local velocity = vector3.zero()
    if input["w"] and input["pressed"] then
        character_idle_anim = anim_state.run
        velocity = vector3.forward() * dt
    else
        character_idle_anim = anim_state.stop
    end
    character_position = character_position + velocity
end
```

In the script above variables are references to those in the ***state*** object. The programmer is free to declare her own variables as needed. This is a tiny script that allows the **LPS** to move only forward, which is unsuitable for most users. But it *does* illustrate the gist of what a script must do: 1) read user input, 2) update the relevant ***state*** variables. Once the ***state*** variables have been updated, the owner of the **LPS** can immediately see their effect on her own machine. In order for the other VIDs to see the same effect the values of the ***state*** variables must be copied to their machines:

[47] See Charon's source code to see the list of variables defined as explicit vs implicit: github.com/saccharineboi/Charon

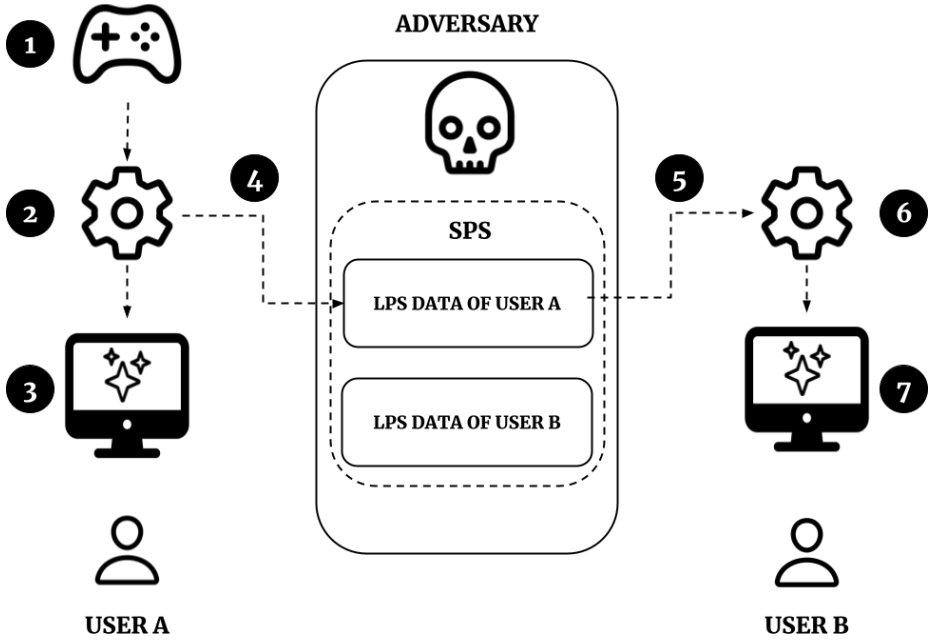

Figure 10: Sharing of ***state*** variables

The figure above shows the process by which the values of the ***state*** variables are copied to other users' machines. Note that in this case we focused on the ***state*** variables of **USER A**, but this process applies to the ***state*** variables of **USER B** as well. Here is the description of each of the steps specified in Figure 10:

1. User input originates from an input device. This can be a keyboard, mouse, joypad, full-body tracker, etc.
2. Input is processed by the script in the **LPS** owned by **USER A**, which causes modification in the ***state*** variables of the **LPS**.
3. **USER A** immediately sees the effects of the modifications done in step 2.
4. New values of the ***state*** variables are sent to the **ADVERSARY**, who stores them inside a block of memory pre-allocated for **USER A**.
5. The **ADVERSARY** transmits the new ***state*** variables to **USER B**.
6. The new values are processed by the client of **USER B**.
7. **USER B** now sees the effects of the modifications done in step 2.

Note that the steps 3 and 4 are concurrent. Naturally there is some delay between when the client of **USER A** modifies the ***state*** variables and when **USER B** sees the result. The delay between steps 2 and 3 depends on the performance of the client of **USER A**, whereas the delay between steps 3 and 7 depends on many more factors, some of which are network-related. The **ADVERSARY** is incapable of reading the contents of these ***state*** variables due to end-to-end encryption[48].

Local programmable states are useful when VIDs wish to share their *own* data with other VIDs, but they become unsuitable when VIDs wish to *modify* the data owned *collectively* by everyone.

[48] Because of AEAD nor can it modify them.

Shared Programmable States

To support a shared world state the Hades protocol makes use of *Shared Programmable States* which are identified by the 0x53465448 magic number in the first 4 bytes of the header. Like **LPSs** the **SPSs** also contain the ***state*** object. The goal of the ***state*** object is three-fold:

1. Establish semantic relationships between the ***state*** variables and the corresponding properties of the HTF object (same as the **LPS**),
2. Allow VIDs within the same **SPS** session to *propose changes* to the values of the ***state*** variables that they all collectively own,
3. Enable the **ADVERSARY** to *verify* the proposed changes.

The "`id`" field of the HTF object is an uint32_t that identifies the $\mathcal{H}$ function to be executed by the **ADVERSARY**. The values are standardized according to the following table:

Value	Purpose
0	Identifies *Local Programmable States*
1	Specifies a **NULL SPS**, where $\mathcal{H}$ = nop
2 - 65535	Specifies a standardized $\mathcal{H}$ function
65536 - 4294967295	Specifies a custom $\mathcal{H}$ function

The value 0 is used to identify *Local Programmable States*. The value 1 specifies a **NULL SPS**, which is a special kind of $\mathcal{H}$ function that returns its input as it is (identity). The range [2, 65535] defines a list of standardized $\mathcal{H}$ functions that the **ADVERSARY** is *required* to support[49]. The range [65536, 4294967295] defines a list of custom $\mathcal{H}$ functions that the **ADVERSARY** may optionally support[50].

SPSs also declare ***state*** variables that are similar to the ones defined by **LPSs**, however, the ***state*** variables defined within **SPSs** are *shared*, in the sense that they refer to a property of a *virtual world* that all VIDs within the same **SPS** session inhabit. Therefore modification of the state variable(s) *affects* the virtual world. Whether VIDs take turns to modify the ***state*** variables or do so concurrently depends on the "`id`" field of the HTF object:

```
{
    "state": {
        "id": 2,                    /* Specifies CHESS_ILLEGAL_LEGAL_CHECKMATE_WHITE_BLACK */
        "white_rook_position": {
            "category": NODE,
            "index": 234,
             "type": TRANSLATION,
        },
        /* etc. */
    },
    /* other properties here */
}
```

HTF files that contain **SPSs** tend to be much larger than those that store **LPSs** due to virtual worlds usually having many more components (i.e. meshes, textures, etc.) than the avatars owned by the VIDs. Similar to **LPSs**, the ***state*** variables in **SPSs** can also refer to *implicit properties*. But unlike **LPSs**, the scripts contained within **SPSs** have a different structure:

[49] Adversaries may compete with each other based on *how many* standardized $\mathcal{H}$ functions they support in addition to *how well* they support them.
[50] Custom $\mathcal{H}$ functions allow adversaries to offer new functionality that may not be standardized due to hardware restrictions or other reasons.

```
EMPTY_SQUARE_ID = 0.0
WHITE_ROOK_ID = 4.0 / 8.0
BLACK_ROOK_ID = -4.0 / 8.0

SQUARE_POSITIONS = {
    square_a1_position,
    square_b1_position,
    -- other 62 remaining squares
}

function update_state_vars(h_in)
    for i = 1, 64 do
        h_in[i] = h_in[i + 64]
    end
    for i = 1, 64 do
        if h_in[i] == WHITE_ROOK_ID then
            white_rook_position = SQUARE_POSITIONS[i]
        elseif h_in[i] == BLACK_ROOK_ID then
            black_rook_position = SQUARE_POSITIONS[i]
        end
    end
end

function consume(h_out, h_in)
    if h_out[0] > h_out[1] then
        lamp_color = vec3(0.0, 1.0, 0.0)
    else
        lamp_color = vec3(1.0, 0.0, 0.0)
    end
    update_state_vars(h_in)
end

function construct_h_in(h_in, piece_id, old_pos, new_pos)
    for i, square in SQUARE_POSITIONS do
        if old_pos == square.position then
            h_in[i + 64] = EMPTY_SQUARE_ID
        elseif new_pos == square.position then
            h_in[i + 64] = piece_id
        end
    end
end

local saved_position = nil

function produce(h_in, dt)
    if input[“mouse_left”] then
        local new_ray = ray.shoot(camera.position, camera.direction)
        local intersected_node = new_ray.intersect(world)
        if intersected_node ~= nil then
            if saved_position == nil and intersected_node.position == white_rook_position then
                saved_position = white_rook_position
            elseif saved_position ~= nil and intersected_node.position ~= white_rook_position then
                construct_h_in(h_in, WHITE_ROOK_ID, saved_position, intersected_node.position)
                h_in.submit() -- async call
                saved_position = nil
            end
        end
    end
end
```

The scripts within **SPSs** have two special functions:

- *consume* function takes the output (h_out) produced by the **ADVERSARY** and updates the world accordingly. In this case if the condition h_out[0] > h_out[1] is true then the move made by the opponent is assumed to be legal. This causes a lamp placed somewhere in the world to shine green. Otherwise the light will turn red. Then it calls the user-defined *update_state_vars* function which updates the state of the board. This function is invoked when a new h_out is received by the **ADVERSARY**.
- *produce* function constructs a new input (h_in) that will be consumed by the **ADVERSARY** to produce the next output (h_out). To do so it must listen for user input, do some game logic and decide what needs to be updated. Once it has finished its job it calls *h_in.submit()* which sends the newly constructed input (h_in) to the **ADVERSARY**. The call *h_in.submit()* returns immediately. This function is invoked every frame for the *active* VID[51].

h_in and h_out are the input and the output of the $\mathcal{H}$ function, respectively. Their types, lengths, the range of values that they hold, etc. is determined by the "`id`" field of the ***state*** object. Unlike **LPSs** no memory is preallocated to store the ***state*** variables by the **ADVERSARY**. For **SPSs** the **ADVERSARY** allocates enough memory to store h_in and h_out and the public parameters of the chosen FHE scheme only. For **NULL SPSs** no memory allocation occurs. These scripts are executed by all VIDs, but the $\mathcal{H}$ function is executed only by the **ADVERSARY**.

Note that the encryption and decryption of h_in and h_out are transparent to the programmer. The execution of cryptographic functions is solely the responsibility of the client.

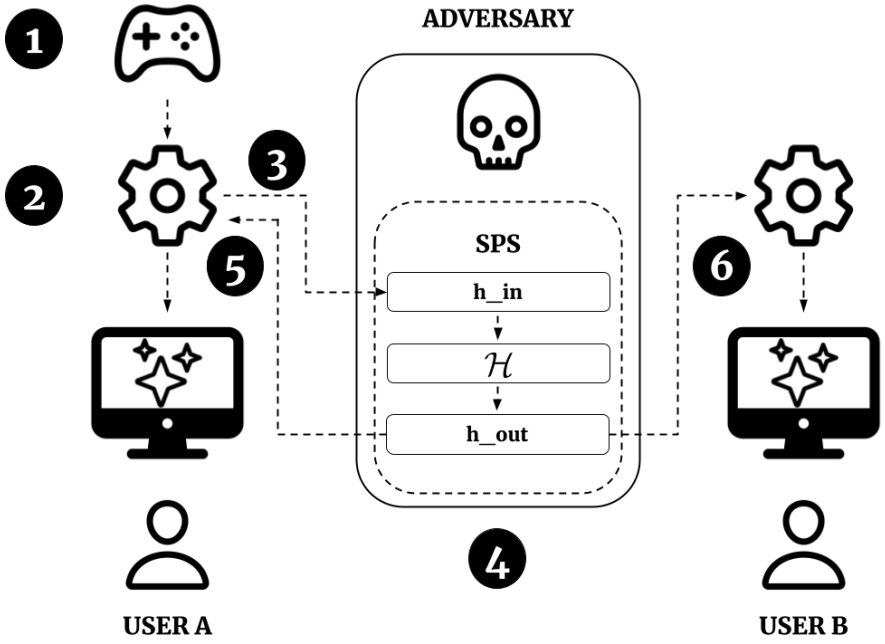

Figure 11: The $\mathcal{H}$ function

The figure above shows the process by which the $\mathcal{H}$ function computes a new world state (h_out) given the previous world state (h_in). Note that in this case we focused on the modification of the world state by **USER A**, but this process applies to the modifications of **USER B** as well. Here is the description of each of the steps specified in Figure 11:

1. User input originates from an input device. This can be a keyboard, mouse, joypad, full-body tracker, etc.
2. Input is processed by the script of the **SPS** by the client of **USER A** (the *active* VID), which constructs a new world state (h_in) based on the output of step 1.
3. The newly constructed world state (h_in) is sent to the **ADVERSARY**.

[51] By *active* VID I mean the VID who must make the next move. For some **SPSs** all VIDs may be active.

4. The **ADVERSARY** executes the $\mathcal{H}$ function on h_in, producing h_out.
5. The output of the $\mathcal{H}$ function is sent to the client of **USER A** (the *active* VID), which then uses it to modify the ***state*** variables of the **SPS**.
6. The output of the $\mathcal{H}$ function is sent to the client of **USER B** (the *passive* VID), which then uses it to modify the ***state*** variables of the **SPS**.

In the beginning of a new **SPS** session the **ADVERSARY** initializes h_in and h_out and sends their default values to the VIDs, who then consume the values and update the relevant ***state*** variables. Based on the "`id`" field of the **SPS** the **ADVERSARY** then determines whose proposed changes he is willing to accept and verify. The VIDs who must make the next move are notified. These VIDs are called *active* VIDs. The **ADVERSARY** waits for the active VIDs to send their proposed changes. Meanwhile *passive* VIDs wait for their turn. The active VIDs invoke the `produce` function and send the new input (h_in) to the **ADVERSARY**, who then executes the $\mathcal{H}$ function and submits the output (h_out) to *all* VIDs. Based on the "`id`" field of the **SPS** the **ADVERSARY** then determines and notifies the new active VIDs. This goes on until either the **ADVERSARY** or one of the VIDs terminates the session. Communication between the **ADVERSARY** and the VIDs is standardized by the use of special constants:

Name	Value	Description
H_OK	0	Successful execution
H_WARN	1	Input is invalid, but executed anyway[52]
H_ERR	2	Execution halted due to an error
H_TERM	3	The session has been terminated by the **ADVERSARY**
H_VID_TERM	4	The session has been terminated by the other VID
H_VID_TIMEOUT	5	The other VID failed to send h_in after **TIMEOUT** milliseconds
H_AUTH_ERR	6	Authentication failed
H_VID_AUTH_ERR	7	The other VID failed to authenticate

It would be more prudent to group these codes into their own categories, similar to HTTP codes. Note that these codes don't carry additional messages. In case there are more than two VIDs, H_VID_AUTH_ERR won't tell us who exactly failed to authenticate unless the session is between *two* VIDs. In any case, the list is incomplete.

The script in page 16 doesn't reveal anything about how the values of h_in and h_out are hidden from the **ADVERSARY**. This is because the encryption and the decryption of h_in and h_out are carried out by the client unbeknownst to the script. The values of h_in are encrypted *after* the h_in.submit() executes. The encrypted values of h_out are decrypted *before* the script executes.

When the "`id`" field equals 1 a special kind of session called the **NULL SPS** session is initiated where the $\mathcal{H}$ function becomes an identity function. The **NULL SPS** is useful in cases where the VIDs are only interested in exchanging E2EE data. Any **SPS** can be turned into a **NULL SPS**.

In the Hadean Metaverse the virtual worlds are designed around the "`id`" field, e.g. for values that denote chess the virtual world contains a chess board and pieces that can be manipulated by the VIDs. Note that the VIDs aren't required to use the *same* virtual world, only one that is *compatible*[53].

[52] This error code is interesting because the **ADVERSARY** isn't *aware* that it produced it.
[53] Two virtual worlds are compatible when they share the same ***state*** variables and their "`id`" fields are equal.

Obols

Before the **SPS** session can commence the VIDs must find and request an **ADVERSARY** to be their arbiter. A special file must be created which includes the following fields:

Name	Description
version	Version of the protocol encoded as a uint32_t value
id	The "`id`" of the **SPS** encoded as a uint32_t value
vids	Array of public identities for corresponding VIDs
passphrase	Secret passphrase known by the VIDs encoded as an ASCII string[54]
fhe_pub	Public parameters of the FHE scheme
ttl	Expiry date of the Obol expressed in Unix milliseconds and encoded as a uint64_t value
extensions	List of extensions requested by the VIDs[55]

This file is called an *Obol*[56]. The act of creating an Obol is called *minting an Obol*. Obols must be minted by the VIDs and sent to the **ADVERSARY** over a secure connection. The act of evaluating an Obol is called *assaying an Obol*. The **ADVERSARY** who assays an Obol may reject it for a variety of reasons, including:

- The Obol may contain invalid information, e.g. it may declare a value for a *version* that is not supported.
- The Obol may be badly constructed, e.g. the names of the properties may be misspelled, or some required properties may be missing, etc.
- The requested capabilities listed in the *extensions* array may not be available or their values may be inappropriate.
- The *id* may specify an **SPS** that the **ADVERSARY** does not support.
- The **ADVERSARY** may be at maximum capacity and cannot afford running another **SPS** session.

The VID who mints and sends the Obol to the **ADVERSARY** is called the *initiator*. The **ADVERSARY** assays the Obol and checks whether the initiator's public identity is in the "`vids`" array. If the Obol is rejected or the initiator's public identity isn't in the "`vids`" array, the connection with the initiator is terminated. In the case of the non-initiator the **ADVERSARY** searches its database to find an Obol that contains the non-initiator's public identity and the passphrase. If no Obol is found, the connection with the non-initiator is terminated. Otherwise, the **ADVERSARY** shares the Obol with the non-initiators. Then the VIDs establish a list of shared secrets and send their encrypted **LPSs** to the **ADVERSARY** who then shares them with the other VIDs. The **ADVERSARY** sends the initial values of h_in and h_out to the VIDs and notifies the active VIDs. The **SPS** session resumes until it is terminated by either the **ADVERSARY** or one of the VIDs.

Obols are ephemeral and are destroyed after use. It is recommended that users generate new VIDs for different sessions, but it is not required. It is the responsibility of users to search for and download a compatible virtual world before they initiate or join an **SPS** session. The difficulties involved in the creation and maintenance of the user identity, virtual identities, obols, **LPSs** and **SPSs** may motivate the emergence of *Hot Clients*, which can destroy some of the cryptographic and epistemological guarantees of the Hades protocol. See Appendix A for further discussion.

---

[54] Anyone can mint an Obol that contains the public identity of someone and coerce them into an unwanted **SPS** session by poisoning the database of the **ADVERSARY**. To prevent this the VIDs include a secret passphrase so that the **ADVERSARY** can match against both the public identity *and* the passphrase.

[55] Each **ADVERSARY** is free to define their own extensions, which allows the VIDs to request modifications of the **SPS** session, e.g. the amount of memory to be preallocated to store the **LPSs** of each VID, the security parameters of the protocol, etc. Note that different adversaries may use different names for the same settings.

[56] Obols, just like other human-readable files, are plain JSON files.

Roles

Hades protocol involves three parties: Ἀχιλλεύς, Πάτροκλος, and Μίνως:

- Ἀχιλλεύς wants to play chess with Πάτροκλος. However, everytime he does, Πάτροκλος either breaks the rules or refuses to admit defeat. So Ἀχιλλεύς requests Μίνως to act as an arbiter. But Ἀχιλλεύς doesn't trust Μίνως, for he thinks that Μίνως favors Πάτροκλος and will choose his side in disputes. Therefore, Ἀχιλλεύς wants Μίνως to enforce the rules of chess without knowing *who* plays *what*.
- Πάτροκλος receives a request from Ἀχιλλεύς for a game of chess. Like Ἀχιλλεύς, he doesn't trust Μίνως. Unlike Ἀχιλλεύς, he thinks that Μίνως favors Ἀχιλλεύς and will choose his side in disputes. Therefore, Πάτροκλος also wants Μίνως to enforce the rules of chess without knowing *who* plays *what*.
- Μίνως offers himself to be the arbiter of a chess game between Ἀχιλλεύς and Πάτροκλος. He *really* doesn't like it when people break the rules, however, so he decides that he will send the delinquent to Τάρταρος. To this end he must identify the players and the moves that they play.

Since neither Ἀχιλλεύς nor Πάτροκλος want to end up in Τάρταρος, they will have to come up with a scheme to make Μίνως enforce the rules of chess without him knowing anything about the moves being played nor whose game he is enforcing. Notwithstanding his biases, Μίνως is willing to act as a semi-honest arbiter, following the protocol while trying to exploit any available information.

Keys

Name	Description
$(IPQDSSPUB_A, IPQDSSPRI_A)$, $(IDSSPUB_A, IDSSPRI_A)$	Ἀχιλλεύς's identity keys
$(IPQDSSPUB_{\Pi}, IPQDSSPRI_{\Pi})$, $(IDSSPUB_{\Pi}, IDSSPRI_{\Pi})$	Πάτροκλος's identity keys
$(EPQKEMPUB_A, EPQKEMPRI_A)$, $(EKEXPUP_A, EKEXPRI_A)$	Ἀχιλλεύς's ephemeral keys
$(EPQKEMPUB_{\Pi}, EPQKEMPRI_{\Pi})$, $(EKEXPUP_{\Pi}, EKEXPRI_{\Pi})$	Πάτροκλος's ephemeral keys
$LSK_A^1, LSK_A^2, LSK_A^3, ..., LSK_A^N$	Secret keys used by Ἀχιλλεύς to encrypt his **LPSs**
$LSK_{\Pi}^1, LSK_{\Pi}^2, LSK_{\Pi}^3, ..., LSK_{\Pi}^N$	Secret keys used by Πάτροκλος to encrypt his **LPSs**
HESK	Secret key used by Ἀχιλλεύς and Πάτροκλος to decrypt the **SPS** (i.e. h_in and h_out)
SS_{HESK}	Shared secret used by Ἀχιλλεύς and Πάτροκλος to securely exchange *HESK*
$HEPUB^1, HEPUB^2, HEPUB^3, ..., HEPUB^N$	Public parameters of the FHE scheme
OSP	Secret passphrase included in the Obol

The Hades Protocol

1. Ἀχιλλεύς and Πάτροκλος generate their *User Identities* as specified in page 8. The parameters of the **KDF** may differ. They may use their previously generated *User Identities* if they wish so.
2. Ἀχιλλεύς and Πάτροκλος generate their *Virtual Identities* as specified in page 9. It is recommended that they use a different *Virtual Identity* for every session, but it is not required.
3. Ἀχιλλεύς and Πάτροκλος generate their *Public Identities* as specified in page 9. They *befriend* each other by exchanging their *Public Identities* via an authenticated channel, e.g. Signal Messenger[57], or in-person.
4. Ἀχιλλεύς becomes an initiator by minting an *Obol* as specified in page 19. He then generates two ephemeral key pairs *((EPQKEMPUB$_A$, EPQKEMPRI$_A$), (EKEXPUP$_A$, EKEXPRI$_A$))* and computes *HASH(Obol || EPQKEMPUB$_A$ || EKEXPUP$_A$)* which he signs using his identity keys *(IPQDSSPRI$_A$, IDSSPRI$_A$)*, producing two signatures *(PQSIG$_A$, SIG$_A$)*. Then he initiates a new TLS session with Μίνως and sends him the minted *Obol*, his public identity keys *(IPQDSSPUB$_A$, IDSSPUB$_A$)*, his public ephemeral keys *(EPQKEMPUB$_A$, EKEXPUP$_A$)*, and the signatures *(PQSIG$_A$, SIG$_A$)*.
5. Μίνως authenticates himself to Ἀχιλλεύς via TLS. He then assays the *Obol* and verifies the signatures. If the *Obol* is rejected or at least one of the signatures is invalid, the session with Ἀχιλλεύς is terminated. Otherwise he saves the *Obol*, the keys, and the signatures until the *Obol* expires.
6. Πάτροκλος generates two ephemeral key pairs *((EPQKEMPUB$_Π$, EPQKEMPRI$_Π$), (EKEXPUP$_Π$, EKEXPRI$_Π$))* and computes *HASH(EPQKEMPUB$_Π$ || EKEXPUP$_Π$ || IPQDSSPUB$_Π$ || IDSSPUB$_Π$ || OSP)* which he signs using his identity keys *(IPQDSSPRI$_Π$, IDSSPRI$_Π$)*, producing two signatures *(PQSIG$_Π$, SIG$_Π$)*. Then he initiates a new TLS session with Μίνως and sends him his public identity keys *(IPQDSSPUB$_Π$, IDSSPUB$_Π$)*, his public ephemeral keys *(EPQKEMPUB$_Π$, EKEXPUP$_Π$)*, the signatures *(PQSIG$_Π$, SIG$_Π$)*, and the secret passphrase *OSP*.
7. Μίνως authenticates himself to Πάτροκλος via TLS and verifies the signatures. If at least one of the signatures is invalid or no *Obol* is found to contain Πάτροκλος's public identity and the secret passphrase *OSP* or the *Obol* has expired, the session with Πάτροκλος is terminated. Otherwise, he sends the *Obol*, the keys *(EPQKEMPUB$_A$, EKEXPUP$_A$)*, and the signatures *(PQSIG$_A$, SIG$_A$)* to Πάτροκλος. At the same time he sends the keys *(EPQKEMPUB$_Π$, EKEXPUP$_Π$)* and the signatures *(PQSIG$_Π$, SIG$_Π$)* to Ἀχιλλεύς. Μίνως then deletes the *Obol*, the keys, and the signatures.
8. Ἀχιλλεύς computes *HASH(EPQKEMPUB$_Π$ || EKEXPUP$_Π$ || IPQDSSPUB$_Π$ || IDSSPUB$_Π$ || OSP)* and verifies the signatures *(PQSIG$_Π$, SIG$_Π$)*. If at least one of the signatures is invalid, the protocol is aborted. Otherwise, he derives *(LSK$_A^1$, LSK$_A^2$, LSK$_A^3$, ..., LSK$_A^N$)* using some cryptographic PRF keyed by *MK$_A$* and sends the ciphertext *PQCT$_A$* to Πάτροκλος via Μίνως:

   - *(PQCT$_A$, PQSK$_A$) = PQKEM-ENC(EPQKEMPUB$_Π$)*
   - *SS = DH(EKEXPUB$_A$, EKEXPUB$_Π$)*
   - *MK$_A$ = HKDF(PQSK$_A$ || SS || EPQKEMPUB$_A$ || EPQKEMPUB$_Π$ || EKEXPUB$_A$ || EKEXPUB$_Π$, S, CTX)*[58]

9. Πάτροκλος computes *HASH(Obol || EPQKEMPUB$_A$ || EKEXPUP$_A$)* and verifies the signatures *(PQSIG$_A$, SIG$_A$)*. If at least one of the signatures is invalid, the protocol is aborted. Otherwise, he derives *(LSK$_Π^1$, LSK$_Π^2$, LSK$_Π^3$, ..., LSK$_Π^N$)* using some cryptographic PRF keyed by *MK$_Π$* and sends the ciphertext *PQCT$_Π$* to Ἀχιλλεύς via Μίνως:

   - *(PQCT$_Π$, PQSK$_Π$) = PQKEM-ENC(EPQKEMPUB$_A$)*
   - *SS = DH(EKEXPUB$_A$, EKEXPUB$_Π$)*
   - *MK$_Π$ = HKDF(PQSK$_Π$ || SS || EPQKEMPUB$_A$ || EPQKEMPUB$_Π$ || EKEXPUB$_A$ || EKEXPUB$_Π$, S, CTX)*

10. Ἀχιλλεύς decapsulates *PQCT$_Π$* and obtains *PQSK$_Π$*, which he uses to retrieve *(LSK$_Π^1$, LSK$_Π^2$, LSK$_Π^3$, ..., LSK$_Π^N$)* using some cryptographic PRF keyed by *MK$_Π$*. He then derives *SS$_{HESK}$* and uses it to encrypt *HESK*, producing *CT$_{HESK}$*, which he sends to Πάτροκλος via Μίνως together with the random nonce *N*:

    - *PQSK$_Π$ = PQKEM-DEC(PQCT$_Π$, EPQKEMPRI$_A$)*
    - *MK$_Π$ = HKDF(PQSK$_Π$ || SS || EPQKEMPUB$_A$ || EPQKEMPUB$_Π$ || EKEXPUB$_A$ || EKEXPUB$_Π$, S, CTX)*

[57] In case Ἀχιλλεύς and Πάτροκλος use the Signal Messenger, I assume that they have compared and validated their identity public keys. See sub-chapters 4.1 (Authentication) and 4.8 (Identity binding) in "The X3DH Key Agreement Protocol" (Marlinspike et al., 2016).

[58] *S* is defined as a zero-filled byte sequence with length equal to the length of the hash output, in bytes, and *CTX* is defined as a concatenation of string representations of the protocol parameters separated by "_".

- $SS_{HESK} = HKDF(MK_A \,||\, MK_{\Pi}, S, CTX)$
- $CT_{HESK} = AEAD\text{-}ENC(SS_{HESK}, HESK, N)$

11. Πάτροκλος decapsulates $PQCT_A$ and obtains $PQSK_A$, which he uses to retrieve $(LSK_A^1, LSK_A^2, LSK_A^3, ..., LSK_A^N)$ using some cryptographic PRF keyed by MK_A. He then derives SS_{HESK} and uses it together with the nonce N to decrypt CT_{HESK} and obtain *HESK:*

    - $PQSK_A = PQKEM\text{-}DEC(PQCT_A, EPQKEMPRI_{\Pi})$
    - $MK_A = HKDF(PQSK_A \,||\, SS \,||\, EPQKEMPUB_A \,||\, EPQKEMPUB_{\Pi} \,||\, EKEXPUB_A \,||\, EKEXPUB_{\Pi}, S, CTX)$
    - $SS_{HESK} = HKDF(MK_A \,||\, MK_{\Pi}, S, CTX)$
    - $HESK = AEAD\text{-}DEC(SS_{HESK}, CT_{HESK}, N)$

Ἀχιλλεύς and Πάτροκλος use *HESK* to decrypt the **SPS**, by which I mean the values of h_in and h_out. Μίνως initializes the values of h_in and h_out, and encrypts them using one of the public parameters $(HEPUB^1, HEPUB^2, HEPUB^3, ..., HEPUB^N)$, and sends the ciphertexts (CT_{h_in}, CT_{h_out}) to Ἀχιλλεύς and Πάτροκλος:

- $CT_{h\_in} = FHE\text{-}ENC(h\_in, HEPUB^i)$
- $CT_{h\_out} = FHE\text{-}ENC(h\_out, HEPUB^i)$

Ἀχιλλεύς and Πάτροκλος decrypt CT_{h_in} and CT_{h_out} and modify h_in[59], which then they encrypt and send to Μίνως:

- $h\_in = FHE\text{-}DEC(CT_{h\_in}, HESK)$
- $h\_out = FHE\text{-}DEC(CT_{h\_out}, HESK)$
- $h\_in\_new = f(h\_in, ...)$
- $CT_{h\_in\_new} = FHE\text{-}ENC(h\_in\_new, HEPUB^i)$

Μίνως computes a predefined list of functions $(F_1, F_2, F_3, ..., F_N)$ on the ciphertext $CT_{h\_in\_new}$ and sends the output $(CT_{h\_out\_new})$ to Ἀχιλλεύς and Πάτροκλος together with $CT_{h\_in\_new}$, which they decrypt to obtain the updated values[60]:

- $X_1 = FHE\text{-}EVAL(Y_1, CT_{h\_in\_new}, F_1, HEPUB^1, HEPUB^2, HEPUB^3, ..., HEPUB^N)$
- $X_2 = FHE\text{-}EVAL(Y_2, X_1, F_2, HEPUB^1, HEPUB^2, HEPUB^3, ..., HEPUB^N)$
- ...
- $CT_{h\_out\_new} = FHE\text{-}EVAL(Y_N, X_{N-1}, F_N, HEPUB^1, HEPUB^2, HEPUB^3, ..., HEPUB^N)$

where Y_i is some other data used by Μίνως which can be either plaintext or ciphertext. This goes on until the **SPS** session is terminated by either Ἀχιλλεύς, Πάτροκλος, or Μίνως.

Implementations may decide to encrypt the **LPS** payload and the ***state*** variables with different keys. While the payload is transmitted over TLS, the ***state*** variables are sent over DTLS (Rescorla et al., 2012) since the VIDs don't mind the packet loss on streaming data:

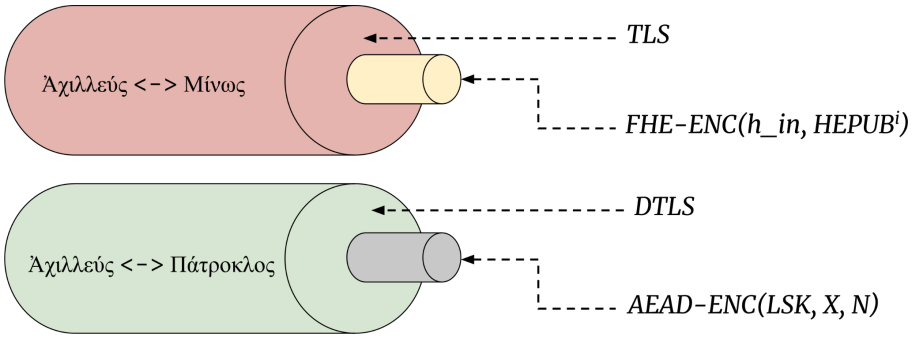

Figure 12: Double-encrypted dual-tunnel data exchange

[59] Only the active VID(s) can modify h_in.
[60] In **NULL SPS** no FHE computations are performed, instead Μίνως sends the old CT_{h_out}.

Figure 13: High-level visual overview of the Hades protocol[61]

[61] Note that some details (e.g. transmission of **LPS** data) are missing. The image of Ἀχιλλεύς is by Jona Lendering licensed under CC 0. The image of Μίνως is by George E. Koronaios licensed under CC BY-SA 4.0. The image of Πάτροκλος is by ArchaiOptix licensed under CC BY-SA 4.0. All images have been cropped and modified into a circular shape with transparent backgrounds using GIMP (GNU Image Manipulation Program).

Karpathian Validators

Before Μίνως can referee a game of chess between Ἀχιλλεύς and Πάτροκλος, he must have full knowledge of the $\mathcal{H}$ function, i.e. its input, its output, and the list of transformations applied to its input to obtain the output. The input can be defined as *some representation* of the move being played, and the output as the *consequence* of that move. There are two ways to represent a chess move:

1. I can write down the name of the piece being moved and its destination. If there are multiple pieces of the same name I can specify its starting square to disambiguate. This is roughly what the standard algebraic notation does. Additional symbols can be used to denote castling, promotions, captures, etc.

2. I can write down the position of the board *before* and *after* the move, concatenated. Forsyth–Edwards Notation (FEN) can be used to denote the positions.

The first method requires initializing/updating the seed/next board position for each move (*stateful*), whereas for the second method the board position is already part of the move (*stateless*):

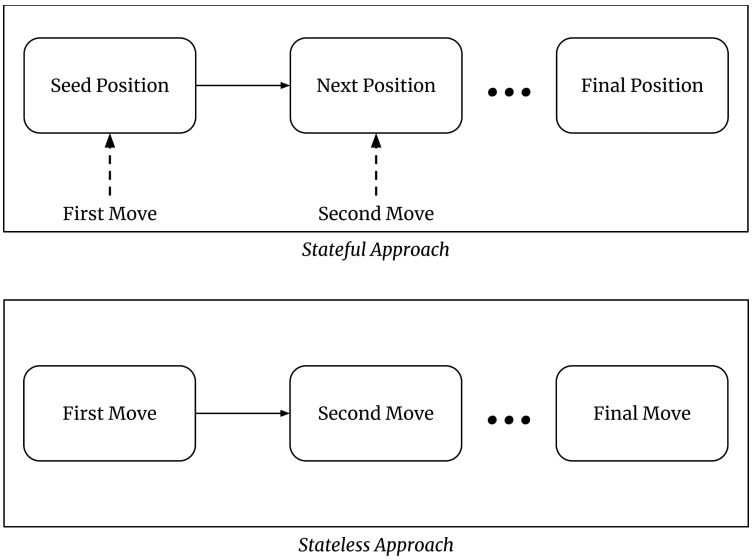

Figure 14: Stateful vs stateless moves

Stateful moves are easier to read for humans, but in this paper I will use the stateless approach. If each move is the input of the $\mathcal{H}$ function, and since Πάτροκλος is known to cheat, the output can be defined as below:

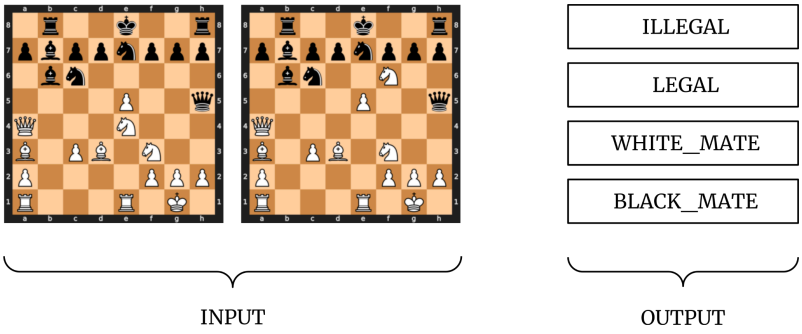

Figure 15: Input and output of the $\mathcal{H}$ function[62]

[62] I have simplified the output by ignoring other possibilities like draws and stalemates.

In Figure 15, Adolf Anderssen plays Nf6 against Jean Dufresne, forking the King and the Queen. Since this is neither illegal nor a checkmate, the $\mathcal{H}$ function outputs "LEGAL". To compute the correct output the chess piece being moved must be identified, its starting and ending squares determined, its movement scrutinized according to the rules of chess, etc. Naturally these steps carve a path in a decision tree:

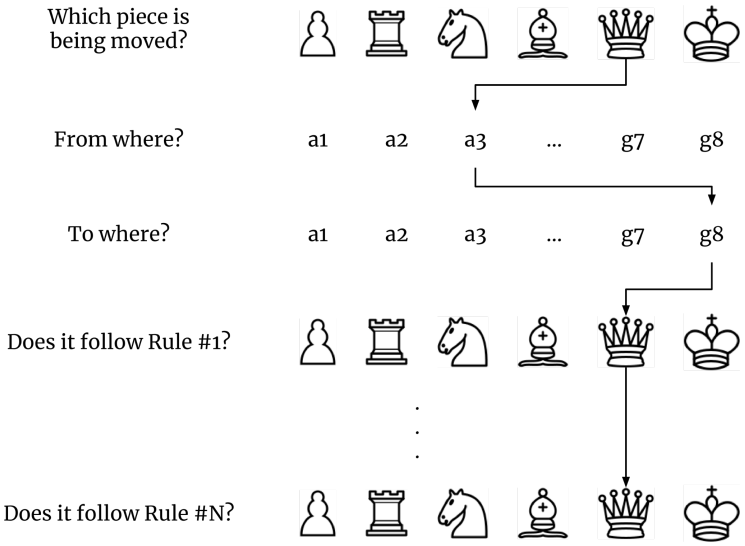

Figure 16: To validate a move one must traverse a tree

In Figure 16, the Queen has moved from a3 to g8. Then, a list of predefined rules is used to test whether this is legal, e.g. the first rule may test whether a3 and g8 lie on a rank, a file, or a diagonal, the second rule may test whether there is an intervening piece, the third rule may test whether a check prevents the Queen from moving, etc. This is a simplified case that doesn't cover promotions and castling. Nevertheless, it is easy to see that the legality of a move depends on a lot of factors that involve the piece(s) being moved, the position of the board, the corresponding rules, etc.

Remark that on page 17 it was mentioned that h_in is encrypted. This means the crucial data needed to test for legality is hidden. We cannot know whether there is a check preventing the Queen from moving when we don't even know if the Queen has moved in the first place. Encryption makes it almost impossible to extract this information from the ciphertext. One way to solve this problem is by designing a circuit for every path in Figure 16, e.g. we may design a circuit to test whether the piece being moved is a Knight, then design another circuit that tests whether the target square is vacant, and so forth. Of course, we have to run all circuits as we don't know the exact path taken in the decision tree:

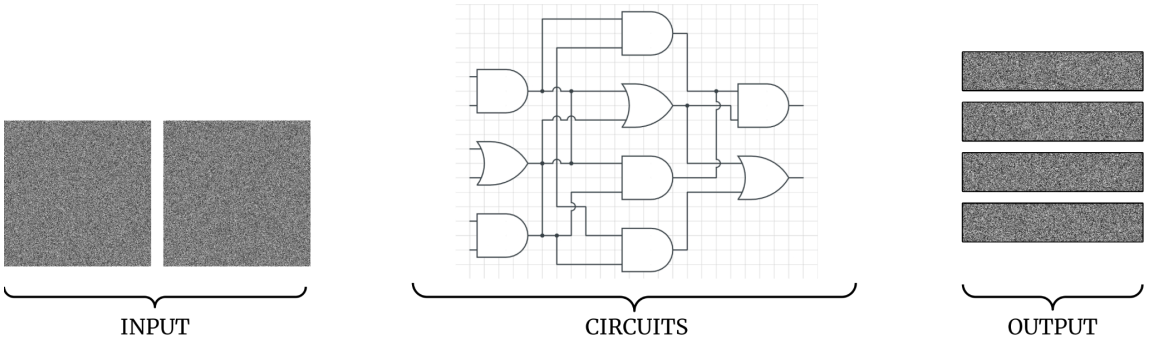

Figure 17: $\mathcal{H}$ function as a network of circuits

Manually designing these circuits, however, while doable, is very cumbersome. Instead I will *search* for them in the circuit space[63]. I begin by associating each piece and the empty square with a floating-point value:

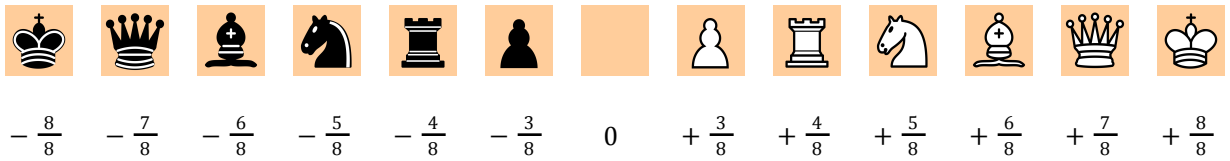

Figure 18: Floating-point representations of the pieces and the empty square

Then I define the output as a four-dimensional vector such that:

ILLEGAL	(0.99, 0.01, 0.01, 0.01)
LEGAL	(0.01, 0.99, 0.01, 0.01)
WHITE_MATE	(0.01, 0.99, 0.99, 0.01)
BLACK_MATE	(0.01, 0.99, 0.01, 0.99)

Thus the problem is reduced to training a model using a dataset of pairs where each pair is a chess move represented by an array of floating-point values as specified in Figure 18, and each label represented by a vector as defined above:

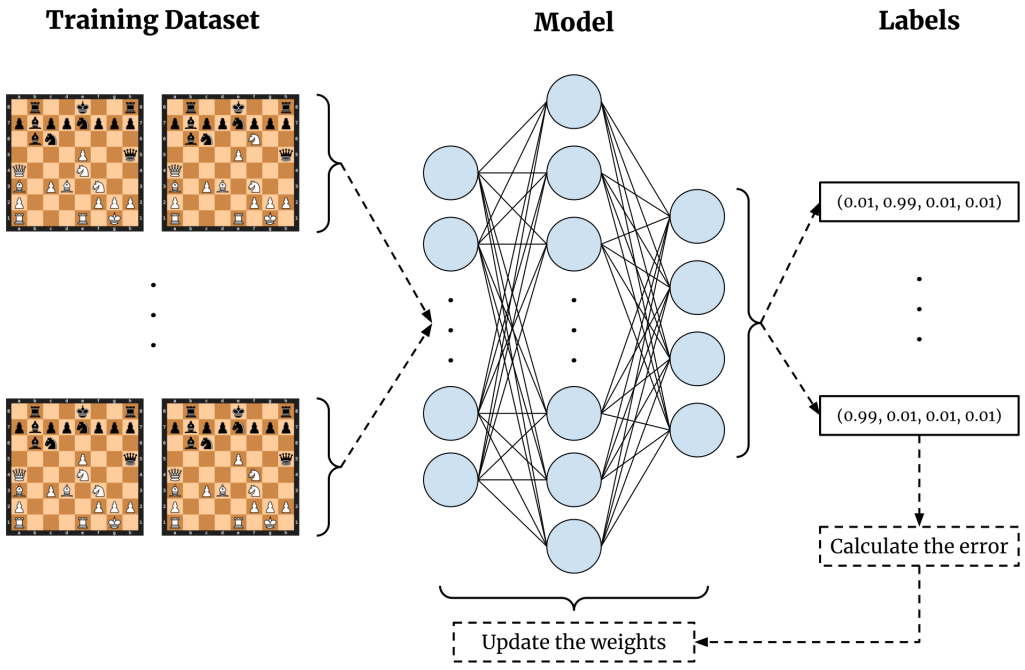

Figure 19: Searching for circuits in the circuit space

I call models trained on such pairs – where each pair is a continuation of some phenomena – that predict whether given phenomena abide by the rules present in the training dataset *Karpathian Validators*[64].

[63] The idea of searching for software in program space was introduced and discussed in "Software 2.0" (Karpathy, 2017).
[64] For a more general case see *Karpathian Simulators* in Appendix B.

I reduce the problem further by training three separate binary classifiers using gradient descent (Cauchy, 1847) with backpropagation (Rumelhart et al., 1986) as the optimization algorithm and the mean squared error (MSE) as the loss function (Bickel et al., 2015):

Model	Purpose	Number of neurons in the input layer	Number of neurons in the hidden layer	Number of neurons in the output layer	Learning rate used during training	Number of training samples used
model-α	Classify legal and illegal moves	128	128	2	0.1	10 million
model-⊠	Classify checkmates and non-checkmates	128	128	2	0.01	2 million
model-γ	Classify white and black checkmates	128	128	2	0.01	2 million

Training samples are extracted from lichess_db_standard_rated_2024-11.pgn[65]. Since there isn't a database for illegal moves I generate my own using the fact that python-chess (a chess library for Python)[66] throws an exception when encountering an illegal move[67]:

```
chess_letter_markings = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
chess_number_markings = ['1', '2', '3', '4', '5', '6', '7', '8']

for move in game.mainline_moves():
    prev_pos = str(board.board_fen())
    illegal_move_uci = str(move.uci())
    try:
        while True:
            if len(illegal_move_uci) == 5:
                illegal_move_uci = illegal_move_uci[:4]
            else:
                illegal_move_uci = illegal_move_uci[0] + illegal_move_uci[1] +
random.choice(chess_letter_markings) + random.choice(chess_number_markings)
                board.parse_uci(illegal_move_uci)
    except chess.IllegalMoveError:
        illegal_move = chess.Move.from_uci(illegal_move_uci)
        board.push(illegal_move)
        move_tensor = tensorize_move(prev_pos, board.board_fen())
        all_moves.add((move_tensor.tobytes(), ILLEGAL_MOVE_LABEL.tobytes()))
        if len(all_moves) >= max_illegal_moves:
            return all_moves
        board.pop()
        board.push(move)
    except chess.InvalidMoveError:
         board.push(move)
         move_tensor = tensorize_move(prev_pos, prev_pos)
         all_moves.add((move_tensor.tobytes(), ILLEGAL_MOVE_LABEL.tobytes()))
         if len(all_moves) >= max_illegal_moves:
             return all_moves
```

[65] Lichess databases are available at https://database.lichess.org/.
[66] For documentation see https://python-chess.readthedocs.io/en/latest/.
[67] The full source code can be found at https://github.com/saccharineboi/Hades.git.

Below are the training runs of model-α, model-⊠, and model-γ:

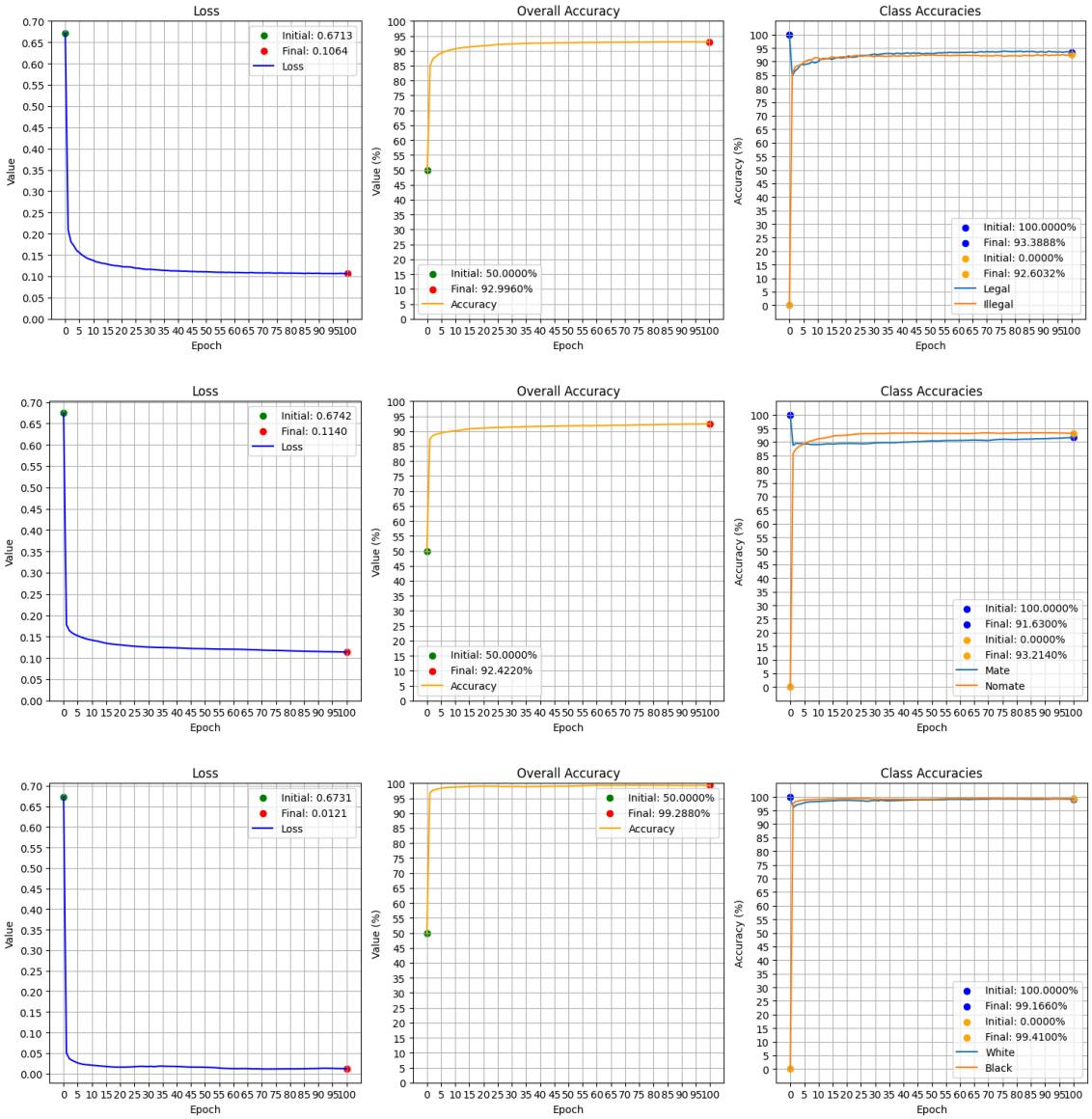

Figure 20: From top to bottom, the training runs of model-α, model-⊠, and model-γ

Their accuracies on Lichess databases from January to June of 2013 are:

Database	Accuracy and loss of model-α		Accuracy and loss of model-⊠		Accuracy and loss of model-γ	
lichess_db_standard_rated_2013-01.pgn	93.813515%	0.093202	92.539078%	0.126696	99.309647%	0.011806
lichess_db_standard_rated_2013-02.pgn	93.820229%	0.092993	92.611618%	0.125757	99.340698%	0.011336
lichess_db_standard_rated_2013-03.pgn	93.843567%	0.092535	92.722206%	0.122951	99.334076%	0.011727
lichess_db_standard_rated_2013-04.pgn	93.875481%	0.092033	92.682701%	0.123651	99.258339%	0.012468
lichess_db_standard_rated_2013-05.pgn	93.874863%	0.091946	92.729904%	0.122439	99.259995%	0.012195
lichess_db_standard_rated_2013-06.pgn	93.907936%	0.091325	92.860413%	0.119425	99.246025%	0.012457

Note that for model-α the accuracy is that of legal moves since Lichess databases don't store illegal moves[68]. Our Karpathian Validator is thus a cascade of models working together to predict the correct output:

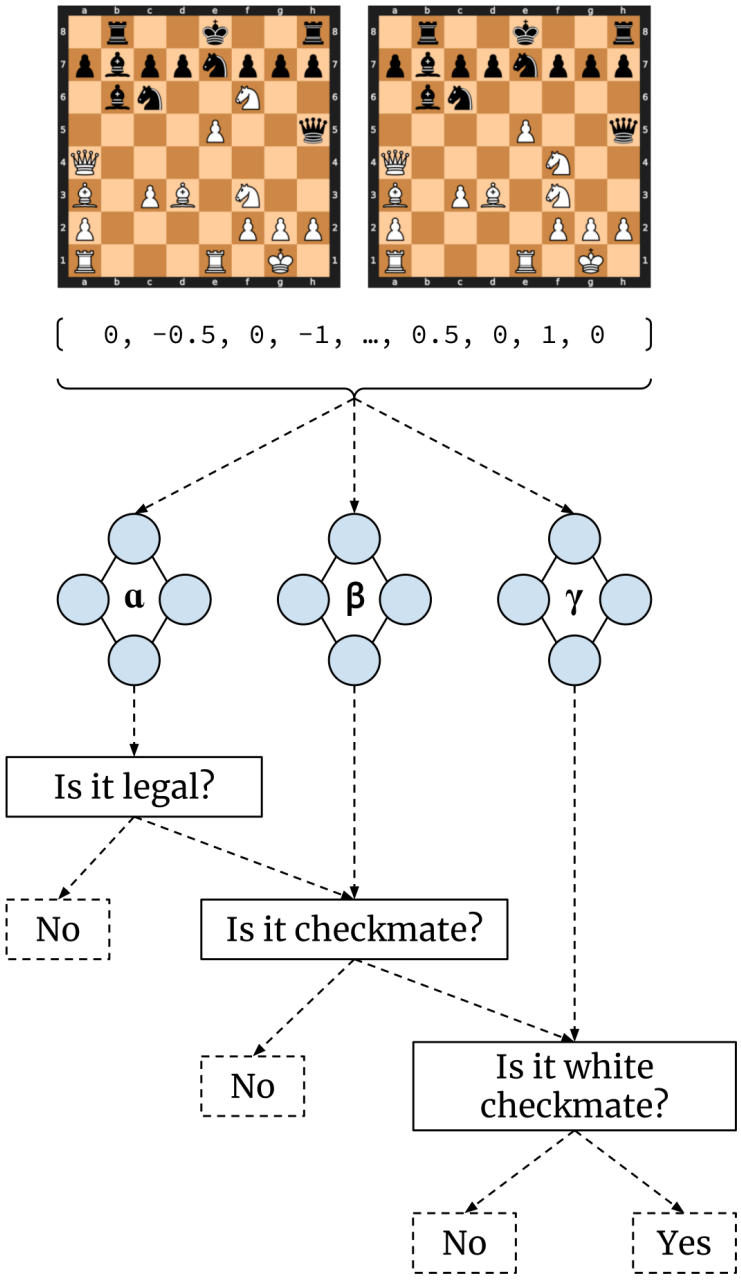

Figure 21: Karpathian Validator as a cascade of models

Since the moves are encrypted, all three models are run for every input, but their outputs are either used or discarded based on the output of the other models. If model-α predicts that the move is illegal, then the outputs of model-◙ and model-γ are discarded. Otherwise, if model-◙ predicts that the move is not a checkmate, then the output of model-γ is discarded. The discarding happens on the client-side. Hadean adversaries can't tell which output(s) should be discarded, because the outputs are also encrypted.

[68] Another problem is the fact that the number of illegal moves in chess is much higher than the number of legal moves, but this isn't taken into account neither by the script on page 28 nor by the training function. This makes it even harder to precisely determine the true accuracy of model-α.

The $\mathcal{H}$ function defined on page 4 becomes:

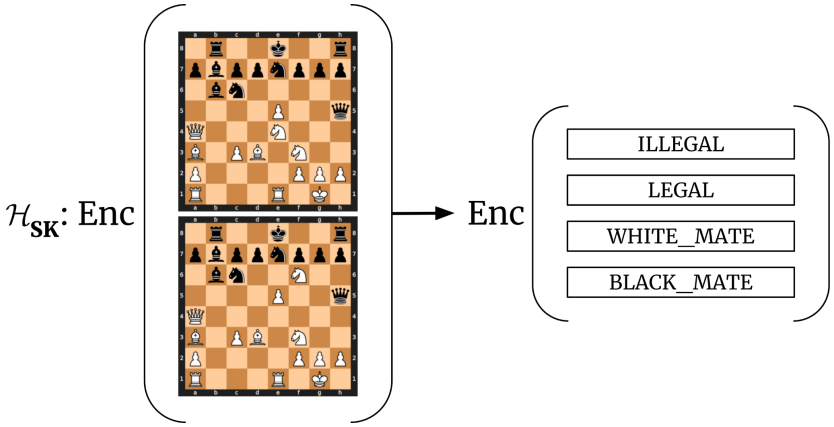

Figure 22: Encrypted input (Θ_i) and output (Θ_k) of the $\mathcal{H}$ function

The idea of computing on encrypted data was first introduced in (Rivest et al., 1978)[69], but it took 30 years before a scheme capable of computing arbitrary number of operations on encrypted data was shown to be feasible in (Gentry, 2009)[70]. In this paper I won't use Gentry's original construction due to its practical limitations.

The mathematical operations performed by the multi-layer perceptron (MLP) during inference can be divided into two groups:

1. *Linear* functions, e.g. weighted sum of inputs to a neuron.
2. *Nonlinear* functions, e.g. the sigmoid activation.

Since these involve floating-point numbers I will use the CKKS scheme by (Cheon et al., 2019). Each inference executes four linear functions in total: two matrix-vector multiplications, and two vector additions. Once trained, the weights of the MLP are packed in diagonal-order (Halevi et al., 2014)[71]. The parallel *systolic* multiplication algorithm[72] is used for matrix-vector multiplications. Each of these multiplications increases the multiplicative depth of the circuit by one. The weights of the MLP that form non-square matrices[73] are packed using the *Hybrid* approach (Juvekar et al., 2018)[74]. Each inference also executes two non-linear functions in total: two sigmoid activations, each of which is a logistic function with the following formula:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

Since CKKS doesn't natively support non-linear functions I use the Chebyshev approximation (Cody, 1970) of the logistic function with the following parameters for each model:

Model	**Range**	**Degree**
model-α	[-450, 350]	100
model-⊠	[-100, 100]	100
model-γ	[-100, 100]	100

[69] Rivest et al. called it *privacy homomorphism*. See https://fhe.org/history for the history of FHE.
[70] Gentry also published an essay on FHE which can be read at https://crypto.stanford.edu/craig/easy-fhe.pdf.
[71] This is called the "Diagonal" approach.
[72] According to Halevi & Shoup, this algorithm was first described in section 3 of Chacha (Bernstein, 2008). See section 4.3 of (Halevi et al., 2014) for more info.
[73] This is due to the output layer having a smaller dimension than the hidden layer.
[74] Juvekar et al. explains that the resulting ciphertext contains partial sums that need to be accumulated using the *rotate-and-sum* algorithm. See section V.F of the paper.

The Chebyshev approximation of the logistic function consumes a multiplicative depth of eight. Therefore the total multiplicative depth of the circuit is 18. The inference on the first 100 encrypted samples of the testing dataset for each model is profiled below:

Model	**128-bit classic security**			**128-bit quantum security**		
	Accuracy	**Loss**	**Performance**	**Accuracy**	**Loss**	**Performance**
model-α	90%	0.247406	97.1 seconds	90%	0.247406	114.54 seconds
model-⊠	88%	0.145914	100.43 seconds	88%	0.145914	175.18 seconds
model-γ	98%	0.011525	132.76 seconds	98%	0.011525	185.13 seconds

The software for the above tests was written in C++ using the OpenFHE library[75] (Al Badawi et al., 2022) and ran on the AMD Ryzen 7 3700U processor with 20 GB of DDR4 memory[76]. The security parameters are defined in (Albrecht et al., 2018). Compare the above statistics with those of inference on cleartext:

Model	**Accuracy**	**Loss**	**Performance**
model-α	96%	0.083560	16.49 microseconds
model-⊠	90%	0.139590	15.07 microseconds
model-γ	99%	0.010177	16.33 microseconds

The numbers reported for performance on both the ciphertext and the cleartext are the averages for 100 samples, e.g. it takes on average 97.1 seconds for each inference on ciphertext with 128-bit classic security versus on average 16.49 microseconds for each inference on cleartext for model-α. Private inference is therefore about a million times slower. The sizes (in bytes) of the input and the output ciphertexts, the cryptocontext, and the keys are shown below:

Size of the input ciphertext	**Size of the output ciphertext**	**Size of the cryptocontext**	**Size of the public key**	**Size of the relinearization key**	**Size of the rotation key**
20976069	2099281	1513	28316527	84943405	10787409369[77]

[75] With the latest commit hash being 7b8346f4eac27121543e36c17237b919e03ec058. The library was compiled with -DWITH_NATIVEOPT=ON and -DWITH_TCM=ON.
[76] The machine in question (Asus Vivobook X512DA/F512DA) had two memory devices: 4 GB of DDR4 memory with 2400 MT/s and 16 GB of DDR4 memory with 2667 MT/s. Note that 2 GB of memory are reserved for the GPU.
[77] This is close to 11 gigabytes!

Limitations

The Hades protocol doesn't natively support a "chatbox", which is ubiquitous in Metaverse applications. This is because Hadean adversaries only store and distribute the most recently received **LPS** data from the VIDs. There is an ad hoc solution: preallocate N bytes of zero-initialized block of memory to store N characters[78] and continuously shift the cursor for every new message. The clients decrypt the messages and store them in their own internal buffers, giving an illusion of a chat history. However, this is not as secure as other E2EE messaging protocols like the Signal protocol, which, in addition to encrypting the messages, also offers perfect forward secrecy and cryptographic deniability.

The Hades protocol is compute, memory, and bandwidth-intensive. The size of the input ciphertext is more than 40,000 times the size of the same input in cleartext (20976069 bytes vs 512 bytes[79]). The size of the output ciphertext is more than 260,000 times the size of the same output in cleartext (2099281 bytes vs 8 bytes[80]). Of all the keys required the largest are the rotation keys at close to 11 gigabytes, which must be uploaded together with the other keys to the server before the **SPS** session can commence. The inference on encrypted input is about six orders of magnitude slower than the same inference on cleartext, and it has higher loss and thus worse accuracy. More optimizations and better hardware are needed before consumer-friendly applications that implement the protocol can emerge.

The Karpathian Validator that was used to solve the problem of classifying encrypted chess moves has a worse accuracy than its traditional *Software 1.0* counterparts, which have 100% accuracy when classifying cleartext moves. A much larger (and deeper) model trained on a bigger dataset is needed to achieve higher accuracies. But it's unclear whether the same approach can be generalized to other games. Another limitation of Karpathian Validators in the context of board games is the difficulties involved in:

1. Generating datasets for legal and illegal moves,
2. Distinguishing a much larger class of examples (i.e. illegal moves) from a smaller class (i.e. legal moves),
3. Determining the accuracy of the model despite the lack of real-world samples for the larger class.

Methods other than Karpathian Validators should be investigated for the solutions of the sort of classifications involved.

Currently the only implementations of the protocol are Charon (client-side) and Minos (server-side), which at the moment run only on Linux and have much higher system requirements than other Metaverse implementations. Although other Metaverse applications don't focus on the privacy of their users to the extent that the Hades protocol does, the high cost of running the protocol will likely discourage adoption, except for **NULL SPS** sessions.

The protocol is also limited to only two participants, and many modifications are required to achieve the more general N-participant solution. For a general case, N copies of the secret key (*HESK*) must be shared, which may reduce the overall security of the protocol, especially in cases where some of the participants use closed-source proprietary implementations that are harder to audit.

Conclusion

The Internet as of today is lacking mainly in two respects: immersion and privacy. The Hades protocol aims to solve both of these problems using its own modified version of the glTF standard and strong cryptography. The results show that the private experiences offered by the protocol have high costs, showcasing a need for more optimizations and better hardware. The source code of the client-side and the server-side implementations of the protocol (Charon and Minos) are available under the GPLv3 license.

The Hades protocol posits a vision of an E2EE Metaverse – a salmagundi of private experiences hidden from view, and aims to be yet another tool in the arsenal of the privacy-conscious individual.

---

[78] Assuming the ASCII characters.
[79] Calculated using 32-bit floating-point values, of which there are 128 in the input array.
[80] Calculated using 32-bit floating-point values, of which there are 2 in the output array.

Appendix A – Hot Clients

On page 19 I wrote that *the difficulties involved in the creation and maintenance of the user identity, virtual identities, obols,* ***LPSs*** *and* ***SPSs*** *may motivate the emergence of Hot Clients, which can destroy some of the cryptographic and epistemological guarantees of the Hades protocol*.

Charon is a cold client. It tries to prevent the personal information of its users from being leaked to unauthorized parties through its careful implementation of the protocol and other *security-aware* measures. Most importantly it only shares the minimum amount of information with the **ADVERSARY** – sufficient for the functioning of the protocol. Hot clients are implementations that share *more* information with the **ADVERSARY** than required by the protocol. A typical hot client binds itself to some platform[81], or many platforms, and requests proof of identity upon use: the platform – which also acts like a Hadean adversary – *knows who its users are*.

The advantages of hot clients include easy handling of the various types of identities, automatic management of obols, **LPS/SPS** handling, etc. They can be particularly attractive for users who don't have the time nor the resources to learn and use cold clients. *Proprietary* hot clients are harder to audit, which makes them potentially more damaging to the privacy of their users[82]. Some hot clients may not even require installation, and instead run on a web browser.

The motivations behind programming hot clients go beyond the desire to track users. Some platforms may offer additional functionality in their hot clients to compete with other platforms. Cold clients add new functionality only insofar it doesn't reduce the privacy of their users nor the overall security of the implementation. But hot clients have no such guarantees. Hot clients also enable platforms to offer subscription services and targeted advertising, which privacy-conscious users may not be so fond of.

It can be hard to quantify the damage to the privacy of someone who uses a hot client, especially a proprietary one. In the worst case the platform may store the decryption keys (i.e. *HESK*) in its database, which would remove the need to upload potentially large keys to its servers. This results in a lightweight client that can run on many more devices compared to a typical cold client. But if the platform can access the decryption keys of its users, then the privacy guarantees of the protocol are effectively destroyed.

Note that it's possible for platforms to track its users via cold clients, too. Here's how it is done: a platform offers its users to upload their virtual identities to its database. The users befriend each other not by exchanging their public identities in-person or via some other authenticated channel, but through the platform, which handles the exchange of public identities under the hood. The users then use their locally-generated virtual identities (that they have shared with the platform) to connect to the platform with a cold client. Because the platform knows which virtual identity belongs to whom, the users can no longer hide their real identities. Unfortunately here the cold clients are powerless. The most they can do is maintain a database of domains for platforms that engage in this behavior and warn their users of the dangers involved before they connect to one.

[81] Here *platform* is a service running in the cloud by an individual or an organization that requires identity verification for access to its functions.
[82] Cold clients can also be proprietary, and their use should be avoided for the same reason.

Appendix B – Karpathian Simulators

On page 27 I defined Karpathian Validators as *models trained on pairs of continuous phenomena that predict whether given phenomena abide by the rules present in the training dataset*. A more general notion is that of a *Karpathian Simulator* – a model that is also trained on pairs of continuous phenomena but instead predicts *future* phenomena given *past* phenomena. Consider the following model:

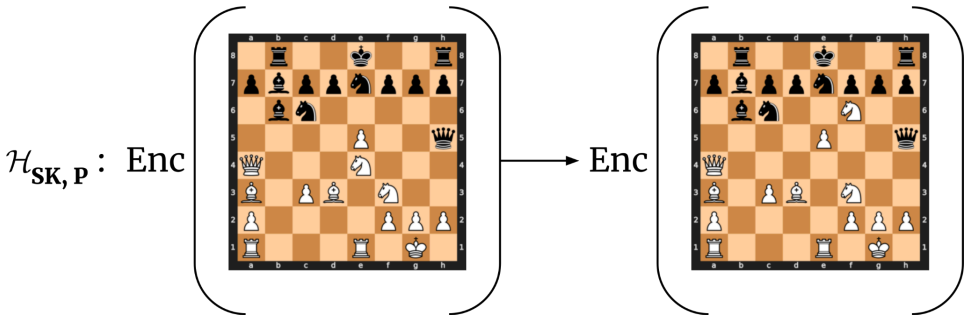

Figure 23: $\mathcal{H}$ function as a Karpathian Simulator

Because there are multiple valid continuations, I have included **P** to denote a set of parameters that influence the simulation in *some way*. In this case $\mathcal{H}$ may be a function of the human brain and **P** the mental state of Adolf Anderssen at time t when he decided to make the move. The difference is that the move played by Anderssen is available only and only to those who possess the secret key **SK**. This has some interesting consequences.

Bostrom defines *ancestor simulation* as the *simulation of the entire mental history of mankind*. Ignoring the computational requirements and the ethical implications of running such simulations, if ancestor simulations can be reduced to:

$$\mathcal{H}_{\mathbf{SK,\,P}} : Enc(\Theta_{\mathrm{i}}) \rightarrow Enc(\Theta_{\mathrm{k}})^{83}$$

then it should be possible to *hide* ancestor simulations among non-ancestor simulations. To further illustrate the point, let us define an entity – call it *Bostrom's Demon* – that checks all the simulations run by some technologically mature civilization[84] and shuts down any that it thinks is an ancestor simulation. BDs may not even require invasive solutions – side-channel analysis may be sufficient[85]. Let us assume that BDs can train binary classifiers to distinguish ancestor simulations from non-ancestor simulations with a sufficiently high accuracy[86].

The key idea is that BDs are *rendered incapable* by Karpathian Simulators that run E2EE ancestor simulations. But there are some caveats. First, BDs cannot have access to the secret key **SK**. How can the *Simulator* – an entity that is running the simulation – prevent BDs from accessing **SK**? One solution is for the *Simulator* to encrypt its own mind under **SK**, and then continue to *homomorphically observe* the encrypted simulation. But this introduces two problems:

1. The *Simulator* loses the ability to influence its outer environment. Even if the *Simulator* memorized the secret key **SK** before encryption, a homomorphic decryption of its own mind would remain encrypted[87].

2. Even though Bostrom assumes substrate-independence, we need an additional assumption of algebraic-independence – that mental states can supervene on any of a broad class of algebraic structures[88].

---

[83] Although the simulation is discretized, there's no reason to think that the simulated experiences of the simulacra won't involve a sense of continuity.
[84] By definition a *technologically mature* civilization is a civilization capable of running ancestor simulations.
[85] This may involve analyzing the electromagnetic spectrum of the waste heat emitted by computers that run ancestor simulations.
[86] This presumes that there are running simulations of both kinds that can generate enough samples for the training function.
[87] Homomorphic decryption is also used for bootstrapping – a method of resetting the noise in the ciphertext.
[88] Can an encrypted mind be conscious?

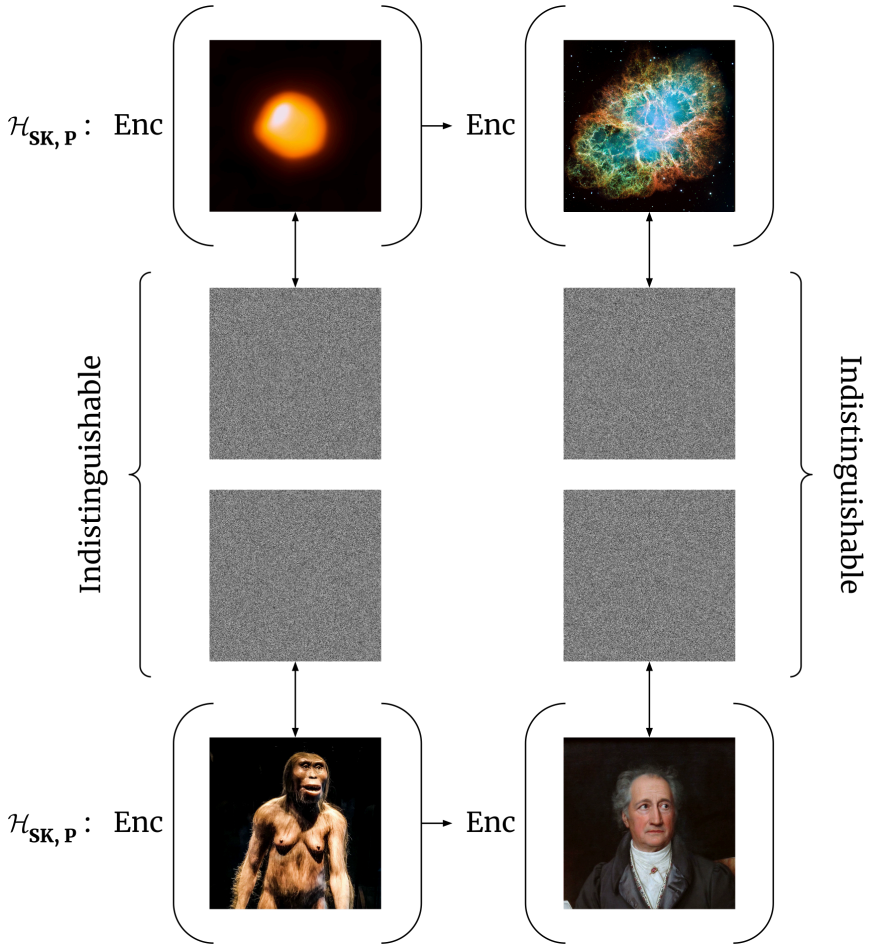

Figure 24: E2EE ancestor simulations are indistinguishable from E2EE non-ancestor simulations[89]

E2EE ancestor simulations form a hierarchy in which the *Simulators* have access only and only to their own simulations and the simulations that run inside their simulations.

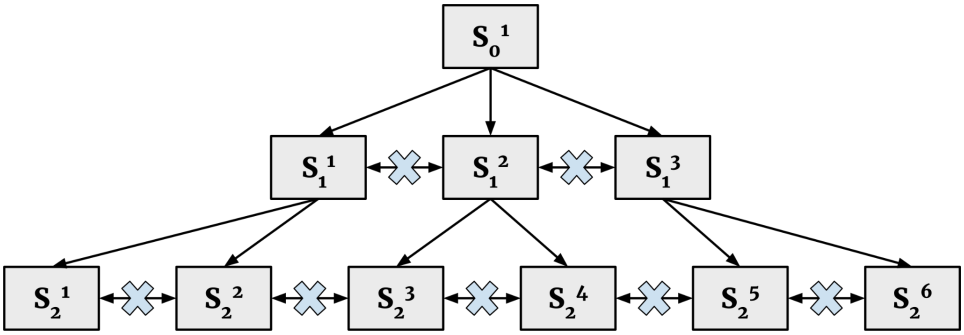

Figure 25: Each *Simulator* has access only to the simulations of its children, but not to the simulations of its neighbors nor to that of its parent, since they are all encrypted under different keys

[89] The top-left is the image of Betelgeuse by Atacama Large Millimeter Array licensed under CC BY 4.0. The top-right is the image of Crab Nebula by NASA's Hubble Space Telescope licensed under CC 0. The bottom-left is the reconstruction of Lucy at the National Museum of Anthropology in Mexico by ErnestoLazaros licensed under CC BY-SA 4.0. The bottom-right is the image of Goethe in 1828 by Joseph Karl Stieler licensed under CC 0. All images have been cropped into square-form.

References

1. AL BADAWI, A., ALEXANDRU, A., BATES, J., BERGAMASCHI, F., COUSINS, D. B., ERABELLI, S., GENISE, N., HALEVI, S., HUNT, H., KIM, A., LEE, Y., LIU, Z., MICCIANCIO, D., PASCOE, C., POLYAKOV, Y., QUAH, I., SARASWATHY, R.V., ROHLOFF, K., SAYLOR, J., SUPONITSKY, D., TRIPLETT, M., VAIKUNTANATHAN, V., ZUCCA, V. (2022). *OpenFHE: Open-Source Fully Homomorphic Encryption Library*, Cryptology ePrint Archive, Paper 2022/915, https://eprint.iacr.org/2022/915.
2. ALBRECHT, M., CHASE, M., CHEN, H., DING, J., GOLDWASSER, S., GORBUNOV, S., HALEVI, S., HOFFSTEIN, J., LAINE, K., LAUTER, K., LOKAM, S., MICCIANCIO, D., MOODY, D., MORRISON, T., SAHAI, A., VAIKUNTANATHAN, V. (2018). *Homomorphic Encryption Security Standard*, HomomorphicEncryption.org.
3. ARCINIEGAS, D. B. (2015). *Psychosis*, Behavioral Neurology and Neuropsychiatry, Vol.21, No.3, Pages 715-736, DOI 10.1212/01.CON.0000466662.89908.e7.
4. ARCISZEWSKI, S. (2020). *XChaCha: eXtended-nonce ChaCha and AEAD_XChaCha20_Poly1305*, Internet Engineering Task Force, https://datatracker.ietf.org/doc/draft-irtf-cfrg-xchacha/03/.
5. AUMASSON, J., NEVES, S., WILCOX-O'HEARN, Z., WINNERLEIN, C. (2013). *BLAKE2: simpler, smaller, fast as MD5*, Cryptology ePrint Archive, https://eprint.iacr.org/2013/322.
6. *Base64 encoding/decoding*, libsodium.org, https://doc.libsodium.org/helpers#base64-encoding-decoding.
7. BASTIAN, W., KARLITSKAYA, A., POETTERING, L., LÖTHBERG, J. (2021). *XDG Base Directory Specification*, https://specifications.freedesktop.org/basedir-spec/latest/.
8. BERNSTEIN, J., D. (2008). *The ChaCha family of stream ciphers*, https://cr.yp.to/chacha.html.
9. BICKEL, P. J., DOKSUM, K. A. (2015). *Mathematical Statistics: Basic Ideas and Selected Topics*, Vol. I (Second ed.), page 20.
10. BIRYUKOV, A., DINU, D., KHOVRATOVICH, D. (2015). *Argon2: the memory-hard function for password hashing and other applications*, https://www.password-hashing.net/argon2-specs.pdf.
11. BLACK, J., GREEN, A. (1992). *Gods, Demons and Symbols of Ancient Mesopotamia: An Illustrated Dictionary*, University of Texas Press, ISBN 978-0292707948.
12. BOS, J., DUCAS, L., KILTZ, E., LEPOINT, T., LYUBASHEVSKY, V., SCHANCK, J. M., SCHWABE, P., SEILER, G., STEHLÉ, D. (2017). *CRYSTALS -- Kyber: a CCA-secure module-lattice-based KEM*, Cryptology ePrint Archive, DOI 10.1109/EuroSP.2018.00032, https://eprint.iacr.org/2017/634.
13. BOSTROM, N. (2003). *Are You Living in a Computer Simulation?,* Philosophical Quarterly, Vol. 53, No. 211, Pages 243-255.
14. CAUCHY, A. (1847). *Méthode générale pour la résolution des systèmes d'équations simultanés*, C. R. Acad. Sci. Paris, 25:536–538.
15. CHEON, J.H., HAN, K., KIM, A., KIM, M., SONG, Y. (2019). *A Full RNS Variant of Approximate Homomorphic Encryption*, In: Cid, C., Jacobson Jr., M. (eds) Selected Areas in Cryptography – SAC 2018, SAC 2018, Lecture Notes in Computer Science(), vol 11349, Springer, Cham, https://doi.org/10.1007/978-3-030-10970-7_16.
16. CHEON, J.H., KIM, A., KIM, M., SONG, Y. (2017). *Homomorphic Encryption for Arithmetic of Approximate Numbers*, In: Takagi, T., Peyrin, T. (eds) Advances in Cryptology – ASIACRYPT 2017, ASIACRYPT 2017, Lecture Notes in Computer Science(), vol 10624, Springer, Cham, https://doi.org/10.1007/978-3-319-70694-8_15.
17. CLINE, E. (2017). *Ready Player One: A Novel*, Ballantine Books, ISBN 978-1524763282.
18. CLOTTES, J., LEWIS-WILLIAMS, D. (1998). *The Shamans of Prehistory*, Harry N. Abrams Publishers, ISBN 978-0810941823.
19. CODY, W. J. (1970). *A survey of practical rational and polynomial approximation of functions*, SIAM Review. 12 (3): 400–423. doi:10.1137/1012082
20. DAVIDSON, R. (1973). *Genesis 1-11 (Cambridge Bible Commentaries on the Old Testament),* Cambridge University Press, Page 33, ISBN 978-0521097604.
21. DIÓSZEGI, V., ELIADE, M. (2024). *shamanism*, Encyclopedia Britannica, https://www.britannica.com/topic/shamanism.
22. DUCAS, L., KILTZ, E., LEPOINT, T., LYUBASHEVSKY, V., SCHWABE, P., SEILER, G., STEHLÉ, D. (2017). *CRYSTALS-Dilithium (Algorithm Specifications and Supporting Documentation),* https://pq-crystals.org/dilithium/data/dilithium-specification.pdf.
23. *ECMA-404: The JSON data interchange syntax*, 2nd edition. (2017). Ecma International, https://ecma-international.org/publications-and-standards/standards/ecma-404/.
24. FRY, S. (2017). *Mythos: The Greek Myths Retold*, Michael Joseph, An Imprint Of Penguin Books, ISBN 978-0718188726.

25. GENTRY, C. (2009). *Fully homomorphic encryption using ideal lattices*, In Proceedings of the forty-first annual ACM symposium on Theory of computing (pp. 169-178).
26. *glTF™ 2.0 Specification version 2.0.1.* (2021). The Khronos® 3D Formats Working Group, https://registry.khronos.org/glTF/specs/2.0/glTF-2.0.html.
27. HALEVI, S., SHOUP, V. (2014). *Algorithms in HElib*, Cryptology ePrint Archive, Paper 2014/106, https://eprint.iacr.org/2014/106.
28. JOSEFSSON, S., SCHAAD, J. (2018). *Algorithm Identifiers for Ed25519, Ed448, X25519, and X448 for Use in the Internet X.509 Public Key Infrastructure*, RFC 8410, DOI 10.17487/RFC8410, https://www.rfc-editor.org/info/rfc8410.
29. JUVEKAR, C., VAIKUNTANATHAN, V., CHANDRAKASAN, A. (2018). *Gazelle: A Low Latency Framework for Secure Neural Network Inference*, https://arxiv.org/abs/1801.05507.
30. KRAWCZYK, H., ERONEN, P. (2010). *HMAC-based Extract-and-Expand Key Derivation Function (HKDF),* RFC 5869, DOI 10.17487/RFC5869.
31. *KTX™ File Format Specification*. (2024). The Khronos® Group Inc., https://registry.khronos.org/KTX/specs/2.0/ktxspec.v2.html.
32. LÉVY-BRUHL, L. (2020). *Primitive Mentality*, Alpha Edition, ISBN 978-9354151521.
33. *Lua Programming Language*, lua.org, https://www.lua.org/.
34. MARLINSPIKE, M., PERRIN, T. (2016). *The X3DH Key Agreement Protocol*, https://signal.org/docs/specifications/x3dh/x3dh.pdf.
35. *Mortynight Run (Rick and Morty).* (2015). Adult Swim, written by PHILLIPS, D., directed by POLCINO, D., https://www.adultswim.com/videos/rick-and-morty.
36. NIR, Y., LANGLEY, A. (2018). *ChaCha20 and Poly1305 for IETF Protocols*, RFC 8439, DOI 10.17487/RFC8439, https://www.rfc-editor.org/info/rfc8439.
37. PARK, G. K. (2023). *animism*, Encyclopedia Britannica, https://www.britannica.com/topic/animism.
38. PAVERD, A.J., MARTIN, A.C. (2014). *Modelling and Automatically Analysing Privacy Properties for Honest-but-Curious Adversaries*, https://www.cs.ox.ac.uk/people/andrew.paverd/casper/casper-privacy-report.pdf.
39. PETERS, M. E., TREISMAN, G. (2017). *Dissociative Identity Disorder*, Johns Hopkins Psychiatry Guide, The Johns Hopkins University, https://www.hopkinsguides.com/hopkins/view/Johns_Hopkins_Psychiatry_Guide/787069/all/Dissociative_Identity_Disorder.
40. PETRU, S. (2019). *Identity and Fear – Burials in the Upper Palaeolithic*, Documenta Praehistorica, Vol. 45, Pages 6-13, DOI 10.4312/dp.45-1.
41. *Playtest (Black Mirror).* (2016). Netflix, written by BROOKER, C., directed by TRACHTENBERG, D., https://www.netflix.com/az/title/70264888.
42. *Puhoy (Adventure Time).* (2013). Cartoon Network, story by MCHALE, P., OSBORNE, K., WARD, P., written by HERPICH, T., WOLFHARD, S., directed by CASH, N., JENNINGS, N., https://www.cartoonnetwork.co.uk/show/adventure-time.
43. RESCORLA, E. (2018). *The Transport Layer Security (TLS) Protocol Version 1.3*, RFC Editor, RFC 8446, DOI 10.17487/RFC8446, https://www.rfc-editor.org/info/rfc8446.
44. RESCORLA, E., MODADUGU, N. (2012). *Datagram Transport Layer Security Version 1.2*, RFC 6347, DOI 10.17487/RFC6347, <https://www.rfc-editor.org/info/rfc6347>.
45. RIVEST, R. L., ADLEMAN, L., DERTOUZOS, M. L. (1978). *On data banks and privacy homomorphisms*, Foundations of secure computation, 4(11), 169-180.
46. *Semantic Versioning 2.0.0*, semver.org, https://semver.org/.
47. SOMOV, A. B. (2017). *Representations of the Afterlife in Luke-Acts (The Library of New Testament Studies)*, T&T Clark, ISBN 978-0567667113.
48. STEPHENSON, N. (1992). *Snow Crash*, Bantam Books, ISBN 0-553-08853-X.
49. YERGEAU, F. (2003). *UTF-8*, a transformation format of ISO 10646, STD 63, RFC 3629, DOI 10.17487/RFC3629, <https://www.rfc-editor.org/info/rfc3629>.
50. 庄子. (476-221 BC.). *The Butterfly Dream.*

Contact


```
-----BEGIN PGP PUBLIC KEY BLOCK-----

mDMEZ0BXpRYJKwYBBAHaRw8BAQdAPXJMSA1N1C8mXWz9Yz9RC2CFMbQmoxTgKV0y
bjjMyp20J09tYXIgSHVzZXlub3YgPHNhY2NoYXJpbmVib2lAZ21haWwuY29tPoiZ
BBMWCgBBFiEET4bgHhQUmf9tqSqZjBcGpMYjvZcFAmdAV6UCGwMFCQlmAYAFCwkI
BwICIgIGFQoJCAsCBBYCAwECHgcCF4AACgkQjBcGpMYjvZf+iAEA8ig7TijKbbDJ
98LO9H+jlNrwY9nFSXnFg5mlU9DkOFQA/j019UtZ/c1u6ldR+0PIlGTcKiN2nKea
3NoCKhMPpSAAuDgEZ0BXpRIKKwYBBAGXVQEFAQEHQIuCW+IXSrqgmcTlm58b9GjG
U+HhiYG75wGCxGPRCxAKAwEIB4h+BBgWCgAmFiEET4bgHhQUmf9tqSqZjBcGpMYj
vZcFAmdAV6UCGwwFCQlmAYAACgkQjBcGpMYjvZcbvQEAjnwQ1KsC6ALBlGa6LwR/
GIQKWjEWbOAn/xCY2O2Z4wsA/2gJrVI3V5Y91wKtnHys68QlxGjfk0uNLiKKm4lX
6iQN
=PLle
-----END PGP PUBLIC KEY BLOCK-----
```