Uotchi: Using Secure Hardware To Create, Interact and Exchange Non-Fungible Objects

Rachel Chen
Lexington High School

Jules Drean
MIT CSAIL

Abstract

Recently, the possibility to exchange and to possess digital objects has garnered attention from the video game industry, the art world, the finance industry and many others. Virtual assets have been all over the news. Nevertheless, making digital resources scarce is not an easy problem. In a setting where duplicating an object has no cost, representing virtual ownership is a conundrum that hasn't found a satisfying answer yet.

In this paper we present Non-Fungible Objects or NFOs, a new type of virtual assets that users create and interact with using Uotchis, our new trusted hardware platform with minimal requirements. NFOs are the combination of a virtual object, a proof of ownership and a proof of authenticity. When a user create, modifies, or sells a NFO, the Uotchi updates or extends the relevant proof using a custom attestation mechanism to keep track of the NFO's validity. As a result, any user can verify the authenticity of the object even if it leaves on an Uotchi local storage. In this paper, we detail an implementation for Uotchi and NFOs and the custom attestation mechanism. We also show several examples of applications for our NFOs to demonstrate how it addresses limitations of existing solutions and how easy it is to leverage our platform to develop new applications.

1 Introduction

In the past few years, there has been an increase of interest for systems that try to decentralize proof of ownership for virtual objects. Such mechanism would allow the democratization of the exchange and selling of virtual assets without relying on a centralized authority. This has many real world implications; it could provide a way for independent artists to sell their art or for video-game developers to create virtual artifacts that can live beyond the limited realm of their own video game [10, 20, 13]. Several propositions have tried to tackle this problem in the past, exploring different paradigms. Beyond centralized databases that track ownership, the currently most widely used decentralized technique for virtual ownership is NFTs or Non-Fungible Token [23, 4, 16]. These tokens, stored on a blockchain, are simple pointers that can be exchange in order to represent ownership of the object they link to. Unfortunately, NFTs present several major flaws. Many artists have seen their work being stolen and sold as NFTs without their consents [8], users have seen their NFTs disappear as they were left with an expensive dead link [2, 14] and highly centralized platforms have emerged to regulate copies and compensate other flaws of the technology.

To address these limitations and more we introduce Non-Fungible Object (NFO), a special virtual asset that is created and exchanged by our secure hardware called Uotchi. NFOs live on Uotchis. Users can interact with their NFOs by running functions through the attestation mechanism provided by the secure Uotchi. A NFO is associated with two proofs, a proof of authenticity that is constantly updated to keep track of the functions run over the NFO and a proof of ownership that is composed of a chain of certificates from the original creator of the NFO to the current user. The functions are easy to develop and give a lot of creative freedom. When a new NFO is created, it is committed to a specific family of functions that are the only one that will be authorized to modify its state. As a result, a developer can create a new collection of NFOs by developing a family of functions that will be the only ones accepted by the attestation mechanism for the corresponding NFOs. This make it possible for a potential buyer to easily verify the authenticity of the NFO as only the original artist or creator can generate a NFO with a valid proof of authenticity. If the NFO is an interactive, only the current owner can modify its state while keeping its proof of authenticity valid. Additionally, using our exchange protocol, a NFO owner can send their virtual asset to another user securely. Our protocol prevent a user to send or receive a NFO twice or to give it to

two different users. Finally, we showcase three different use case for our NFOs. We highlights how easy it is for a developer to leverage our platform to create secure NFOs and how it address limitations form previous systems.

2 Motivation

2.1 Proof of Authenticity

In our proposal, we build a proof of authenticity for our NFOs so that it is difficult to duplicate and replicate the virtual objects. This means that given a NFO, anyone is able to attest of its authenticity i.e. that it was formed by running a series of defined and auditable functions on a set of authorized secure Uotchis. The proof will be constantly updated by the secure device using its attestation mechanism, and if an attacker was attempting to tamper with the state of a NFO without going through the attestation mechanism, then the associated authenticity proof would not verify correctly and the NFO would become invalid.

2.2 Exchange of Virtual Asset on Local Storage

With NFOs, we want the data file to be truly owned and traded by the user. As a result, the proof of ownership and of authenticity are directly carried with the data itself. When the user wants to trade the NFO, they will trade the entire NFO state along with the proof of authenticity and ownership to the next user. This way, the link between the certificate of ownership and the virtual asset doesn't not depend on some third party architecture like an external storage system to host the asset it-self.

2.3 Using Secure Hardware to Replace the Need for Consensus

NFTs rely on a blockchain and complex and inefficient consensus protocol to determine the validity of each transfer of ownership. In our proposal, we do not rely on a distributed consensus protocol. For the safe exchange of the NFO, we rely on our secure hardware to transfer the NFO across a network using our *exchange protocol*. The protocol enforces that no duplicates or forks of the NFO can be accepted by any of our Uotchis. This involve than no Uotchi can give the same NFO twice and no Uotchi can receive the same NFO twice.

2.4 We Don't Need Secure Enclaves

Secure enclaves [6, 3, 15] are a trusted hardware primitive that can run arbitrary user-level code remotely, even in the presence of a compromised operating system. Enclaves attempt to replace the trust required in the hosts and the entire operating system into in a small security monitor which orchestrates enclave interaction with the outside world. Enclaves try to enforce two security guarantees : the integrity *and* the privacy of the code execution. On another hand, enclaves could be considered as too complicated of a primitives and many industrial deployments have been catastrophic as complexity of the solutions increased [21, 22].

With Uotchi, we present a secure hardware that does not require privacy of the code execution or of the data for our protocol to be protected against adversaries. Has a result, only a small portion of the trusted code base needs to be protected from side-channels and our devices can be built using existing out-of-the-shelves processors.

2.5 Current Limitations and Simplifications

This version of our paper uses an oversimplified version of what virtual assets backed by secure hardware could be. In particular, our scheme present the following limitations.

- We mostly ignore micro-architectural side-channel attacks.
- We ignore row-hammer type attacks and delay the study of their impact on our system to future work.
- We ignore secure key storage.
- We limit Uotchis so they can only store one NFO.

3 Background

3.1 Virtual Assets & Ownership

Virtual assets are intangible goods, such as digital art, video game artifacts or any kind of entity developed and traded on a network. Like any other good, their value is defined by how much a buyer is willing to spend to obtain it. Factors like rarity, mainstream notability, aestheticism and overall entertainment play a huge role in determining this made-up price tag of a virtual asset. Although they are sold as virtual products, these digital representations are intrinsically easy to duplicate and replicate, so customers are buying what more closely resembles the service of providing a sense of ownership over an object rather than the object itself. As our lives become more based on virtual spaces, there is a growing demand for new technological solutions that redefine and can enforce the notion of virtual ownership. In order to understand how virtual ownership is implemented in the real world,

let us examine Non-Fungible Tokens (NFTs), the leading technology for establishing virtual ownership in the digital art and gaming industry. NFTs rely on a public shared happen-only ledger, a blockchain, to track ownership and assert that no two users own the same virtual token simultaneously. NFTs do this by holding a pointer which points to the virtual asset, and whenever they want to represent a change in ownership, they chain a new block on the ledger to say that the ownership has changed to a different user. This scheme proved to be incredibly lucrative in the virtual asset business of digital art. As of March in 2021, the digital artist, Beeple, sold a NFT of a photo collage for $69 million dollars. Another prevalent use of NFTs can be seen in the video-game industry, where free-to-purchase games depend on in-game purchases for profit. Virtual goods in the video-game industry are projected to grow up to $138 billions dollars, outweighing both the film and music industry.

3.2 CryptoKitties & Tamagotchis

This project started off as a game inspired by combining CryptoKitties and the hand-held Tamagotchi tablets. CryptoKitties [9] is a blockchain-based game released in 2017 where users can breed and collect digital cats. Elements such as rare physical traits, limited edition tags, and the generation number helped establish rarity among their virtual products and made them more or less valuable in the eyes of the users. For example, the first generation 0 kitties made were dubbed the Founder Cats which, at the height of the game's craze, were sold at a record of 250 ETH ($125K at the time).

Tamagotchis (the inspiration for our Uotchis) were handheld digital pets largely popular in the late 1990s. While they utilized little to no security primitives to guarantee non-fungibility, they inspired to us the idea of binding virtual assets to a dedicated hardware.

In section 8, we build a concrete example of a NFO called CryptoCuties that can be seen as an homage to CryptoKitties.

3.3 Available Primitives

3.3.1 Cryptographic Hash Function

We use cryptographic hash functions to authenticate any files, code or data we use.

Let us define a hash function *H(x)* that goes form $\{0,1\}^*$ to $\{0,1\}^{256}$. The security goal of this hash function is that it is computationally infeasible for any adversary (with polynomial compute power) *Adv* to find two distinct inputs that have the same hash value. In other words, $P[H(m_0) = H(m_1) : (m_0, m_1) \leftarrow Adv()] \leq$ "*negligible*", with m_0 and $m_1 \in \{0,1\}^*$.

3.3.2 Public Key Signing Scheme

We do an extensive use of public key signing scheme in our protocols.

Let us define Pk, Sk as the public and private key pair, m as a message of undetermined length, and σ as a certificate made from the signing scheme. Sk is kept secret whereas Pk can be easily accessible to anyone. Pk, Sk, and σ are in $\{0,1\}^\lambda$ where $\lambda = 128$. We also define the three classic associated algorithms:

$$Gen() \rightarrow (Pk, Sk)$$
$$Sign(Sk, m) \rightarrow \sigma$$
$$Ver(Pk, \sigma, m) \rightarrow True/False$$

The public key signing scheme works by creating a digital certificate over a message using one's private key, which then can be verified that the message indeed came to the owner of that private key through the public key.

Correctness. The public signature scheme is considered correct if, for any message m, $P[Ver(Pk, Sign(Sk, m), m) = \text{ True} : (Pk, Sk) \leftarrow Gen()] = 1$.

Security. To define the security of the public signature scheme, we use a classic game-based definition. Suppose any adversary that runs in polynomial time and can ask for any message sent over a network as well as the signature over said message. The adversary wins the game and breaks security if they can forge a signature on a message that is not part of the query set (set of the messages and signatures that they've queried throughout the network).

3.3.3 Hardware Primitives

Privilege Mode and Secure Monitor. We assume that the processor of our secure device possesses several mode of executions. In particular, we assume the existence of a privileged mode (we will call it the machine mode "M" like in RISCV [24]) and an unprivileged mode (we will call it the user mode "U" like in RISCV). These could also correspond to the "un-secure world" and "secure world" abstraction provided by ARM TrustZone [15].

Access Restricted Storage. We assume that software running in M mode is able to access the entire device memory without restriction. On the other hand, we assume software running in M mode is able to restrict access to part of memory to software running in U mode. These assumptions are realistic and can be archived using technologies like RISCV PMP primitives or ARM TrustZone primitives.

System Calls. System Calls (or syscalls) are a type of hardware interrupts that can be triggered by any code in U Mode by executing a given instruction (ecall in RISCV). The interrupt mechanism will be triggered and the core will switch to M mode and run syscall interrupt handler and return to the user code that called it. This mechanism ensure that code run within these interrupt handler is only executed through a precise entry point and that user code cannot run arbitrary code snippets in M mode.

True Random Number Generator. Uotchis are equipped with hardware TRGN that can generate fresh randomness on demand that is suitable for cryptographic use.

4 Threat Model

In this section we will describe 1) the security guarantees we are defining for our NFOs 2) the different adversarial models we consider and which guarantees hold under which model and 3) the assumptions we need to make for these guarantees to hold.

4.1 Security Guarantees

As the name states, our NFOs should be *non-fungible*. At a high level, this means that a given NFOs cannot be easily replaced by another equivalent NFO and they each represents a unique digital asset. We also require two more specific guarantees from our NFOs.

1. **Non-fungible:** It is not possible to forge a valid NFO with a given state without following the appropriate protocol on the secure hardware.

2. **Authenticity:** Reproducing a NFO is "difficult": it requires a skilled counterfeiter to guess how the different functions were executed to produce the NFO.

3. **Liveliness:** A NFO cannot be made invalid intentionally by a malicious adversary.

4.2 Adversarial Models

We consider two different adversaries:

Local Adversary. A local adversary has physical access to the secure device and can interact in many ways with the hardware by running arbitrary code in unprivileged mode and reading all memory except the device's private key. We also consider that the malicious user has complete control over the network where the exchange protocol will happen. Under this strong adversarial model, only non-fungible and authenticity guarantee holds.

Remote Adversary. A remote adversary does not have physical access to the device but can still control the network used for the exchange protocol. Under this adversarial model, all the guarantees hold.

4.3 Security Assumptions

We also do several security assumptions required in order to be able to uphold the security guaranties.

- We trust the hardware to be part of the Trusted Code Base i.e. to be correct and rightfully implemented.

- We trust the code of the privileged code of the Security Monitor and consider it as part of the Trusted Code Base (more details in section 5.)

- We trust the manufacturer to produce the hardware and to provision the Uotchis with keys correctly.

- We assume the secret keys stored in our Uotchis cannot be extracted by the adversary.

5 The Uotchis Platform

5.1 Platform Overview

We assume that the only piece of software able to run in privilege mode is a Secure Monitor (SM). The SM is a small piece of software similar to what is described in some TEE platforms[7]. It is responsible for filtering hardware interrupts, managing access to special memory regions and running an attestation mechanism (see section 7) that will make it possible for the user to modify the NFOs while updating the proof of authenticity to keep it valid. The attestation mechanism's API is accessible to the user through system calls. The SM is in charge of enforcing the non-fungibility property of the NFOs by checking for specific preconditions on the NFO state before accepting to running the different system call handlers.

5.2 Memory Layout

Leveraging the hardware capabilities to restrict access to some part of memory (see section 3.3.3) the SM will split the memory in three main sections at boot and set up memory access rights accordingly. The address and layout of these sections are static, public, part of the secure

device specification and known to developers when writing new families of NFO.

First the *Device-Secrets Storing Section* is only readable and writable by the SM. It is used to store the device secret key Sk_D (see section 5.3).

Second the *Security Monitor Section* is only writable by the SM but does not store any secrets and can be read by unprivileged code. It is used to store the SM code but also the SM data structures detailed in section 10.2.

Third the *Unprotected Section* is public and can be read and written by unprivileged code. It is used to store any code and data that does not belong to the SM. In particular, the Verifiable Function declarations section 10.3, the code of the corresponding functions and the data of the NFO hosted by the device as described in section 10.3.

Concretely, the SM is able to make the region that stores secret keys completely inaccessible to unprivileged code but also to make the part of memory containing the SM code and state non writable from U mode.

5.3 Initialization and Key Provisioning

At the factory, the manufacturer is equipped with a signing key pair a public key signature scheme (see section 3.3.2). We will assume that the manufacturer's secret key is always kept secret while the manufacturer's public key is advertised publicly and considered easily accessible. We also assume that the factory is trusted to follow the correct protocol. To set up the device, the manufacturer generates the device's key pair by running the $Gen()$ algorithm. The device's key pair can be loaded onto the device's memory but its secret key is loaded in a part of DRAM that is only accessible to the Security Monitor (see section 3.3.3). As a result, the device cannot be turned off such that the key will not be erased from memory. The manufacturer will then commit to the device's key by signing a certificate over the device's public key. This certificate is loaded on the device's main memory and can be presented by the device to prove that it, and its key pair was manufactured by a trusted factory. The device key pair is then used to create and update the NFO's proof of authenticity. The NFO also contains a chain of certificates that describe its ownership history. Each block of that certificate chain contains a device public key, a certificate from a trusted manufacturer over that public key and a commitment to the next block (see details in listing 12).

In this scheme, the trusted manufacturer are our root of trust. Anyone can verify the ownership history of the NFO by verifying the chain of certificates. Given that the factory's public keys are easily accessible, a user can use the factories' public keys to verify the different devices' public keys. Once these public keys are verified, the proof of authenticity can be checked by using the public key of the current owner.

Similarly, a dedicated data-structure known to the SM (see details in listing 9) is used by the user to declare the public keys of the manufactures it deems trusted. Note that it is the responsibility of the user to choose the appropriate list of trusted manufacturer to make sure she will not accept any NFO coming from insecure devices as these might be untrusted. If she accepts such an NFO, she will find it difficult to re-sell it to other honest users that keep their list up-to-date.

6 Non-Fungible Objects

Non-Fungible Objects, or NFOs for short, are virtual objects that are created, modified and exchanged between our specialized secure hardware called Uotchis. Uotchis modify these virtual assets in an auditable manner by using an attestation mechanism to constantly update an embedded proofs that guarantees the authenticity of the virtual object. Only special functions that are compatible with our Uotchi called *Verifiable Functions* can modify these virtual assets by being run through our attestation mechanism section 7. When created, a NFO is made part of a NFO collection by committing to a specific family of functions. Every time a Verifiable Function is called, the Uotchi will jump to the Security Monitor to create an attestation proof for the function execution, and update the NFO's proof of authenticity. In order to trade our NFOs securely, we also implement a secure exchange protocol to ensure that our security guarantees are upheld even as NFOs leave and arrive on different devices. As the NFO is exchanged onto different devices, users can verify the validity of the proof of authenticity with their own devices. In this section, we will describe the different pieces of this complex system in detail.

6.1 Adapting the Attestation Mechanism Overview

We modify remote attestation mechanisms similar to [17, 5, 19] to be able to manipulate and modify NFOs in an authenticated manner. In our implementation, the host device attests the authenticity of the functions used to interact, modify or exchange the NFO state. The functions that are attested by the secure device are functions we call Verifiable Functions. The user of the secure device can call the attestation mechanism through system calls (syscalls) to the SM. Without calling the attestation mechanisms over the functions, the user cannot modify the NFO while maintaining a valid proof of authenticity for the NFO, making it invalid.

6.2 Verifiable Functions Specifications

Developers can use limitless creativity to define functions over the NFO data structures—as long as they are Verifiable Functions. Verifiable Functions are functions that the attestation mechanism can verify. For a Verifiable Function to be called through the attestation mechanism, it must match the following calling convention: The input is limited by the underlying architecture of the Uotchi and correspond to the maximum number of registers reserved to function arguments. On another hand, verifiable functions do not have outputs, but as the location and memory layout of the NFO is known to the programmer at compile time, the function can and is expected to modify the NFO state. Finally, any execution of the function should not return but terminate by performing a `END_ATTEST` system call. The SM will run the appropriate handler to finish the attestation.

6.3 NFOs Collections and VF Commitment

In order to define a new collection of NFOs, a NFO developer needs to create a family of Verifiable Functions. A user can download these functions to memory and declare them to the SM to be able to create a NFO for a particular collection. Note that the verifiable function declarations and the verifiable function code is public and their integrity is verified by the attestation mechanism. As a result, this data and code can be placed in the Unprivileged Section of memory.

During its creation, a new NFO is committed to the corresponding family of Verifiable Functions in addition to a transition graph that represent in which order these functions can be called. These functions will be the only legitimate functions the SM will let modify and manipulate the corresponding NFOs of that collection. The NFO commitment to the Verifiable Function family is composed of a list of verifiable functions measurements and a transition matrix of size $(n+4)*(n+4)$ where n is the number of Verifiable Functions that define the NFO collection (see appendix for details section 10). Note that the first 4 entries of the matrix are reserved for the `INIT`, `OFFER`, `GIVE` and `RECEIVE` syscalls. Every time the attestation mechanism is called, the SM will check if the measurement of the function being executed is part of the Verifiable Functions family committed to and if the function can be called according to the transition matrix. This will ensure that the appropriate function is being called over the NFO.

Ultimately, what will make an NFO collection valuable is for future buyers to recognize it as such. This requires the developer to convince the buyers to consider the family of verifiable functions that define the NFOs collection to be trusted. A developer does this by convincing the users that their functions and their collection are secure, i.e. uphold the security guarantees defined by the NFOs (see section 4). In the next section, we provide the set of necessary conditions on Verifiable Functions to create a secure collection of NFOs.

6.4 Security of an NFOs Collection

For a verifiable function to be secure (i.e. for the attestation of these functions to be successful and relevant) it needs to enforce the following restrictions.

Control Flow Integrity. The function control flow should be limited to the part of memory that is measured by the attestation mechanism (i.e. declared by the verifiable function as such).

Restricted Memory Access. The function should not access memory beyond the memory measured by the attestation mechanism and the NFO data structure.

Hard to Counterfeit. It is also up to the NFO developer to ensure that their collection is hard to counterfeit (see example in section 8).

Note that the auditing of the two first conditions could be automated using tools that can perform control flow and memory integrity checks on small binary. We defer the development of such a tool to future work.

6.5 History of Ownership

Accompanying the proof of authentication is the local history of ownership. This chain of certificates keeps track of the changes in the NFO's ownership from device to device. Each block is composed of a certificate from the manufactures that signs the device's public key. When a user wants to transfer ownership of an NFO, it will create a signature over the other device's certificate and add it onto the chain in the NFO's state. The exchange-ownership data structure (listing 12) is detailed in the appendix.

6.6 Verifying the NFO's Validity

Verifying the validity of NFO is quite straight forward. Given the different manufacturers public key Pk_M, the certificate endorsing the device's public key Pk_D and the attested proof block, a verifier will 1) Use Pk_M to verify that Pk_D belongs to a secure device. 2) Use Pk_D to verify the proof on the NFO's state. If all the steps succeed then the attested proof of authenticity is considered valid and the verifier can continue to interact with their NFO.

6.7 Protection against Local Duplication

In order to uphold the NFOs security guarantees, we must be aware and protect against undetected duplication of NFOs. In this section, we will examine how to prevent the duplication of a NFO on a single device that stores a single NFO. We do this by taking the hash of the NFO's current state, and storing it in the SM private memory where a user cannot edit it. This stored hash will be used to check the hash of the NFO every time the device calls the SM to run the attestation mechanism. If the hash stored in the SM matches the hash of the NFO on the device, that means that the user is using the most up-to-date NFO. However, if the hash stored in the SM does *not* match the hash of the NFO on the device, that means that the user is using an out-of-date NFO. In this case, the SM will refuse to run the attestation mechanism for any Verifiable Functions. This prevent a user from forking the NFO and creating two diverging versions of it and potentialy to sell it to to two users. With this mechanism in mind, we may now consider the more difficult obstacle of preventing the duplication of NFOs between multiple devices. We detail the exchange protocol in section 7.3.

7 The Security Monitor Interface

The interface for the attestation mechanism is composed of five syscalls. These syscalls make it possible to interract, modify and exchange the NFO over its lifespan.

7.1 Initialization of NFO

INIT To create a NFO, a user calls the `INIT` syscall to initialize the NFO data structure and the authenticity proof. At initialization, the SM generates a series of bits using the TRNG which will be the NFO's unique identifier. Then, the SM sets the NFO's owner to the device's public key, initializing the device user's ownership over the NFO. The NFO data structure is filled out with the appropriate values (more details in section 10). In particular, the NFO state will include the description of the family of Verifiable Functions it can be modified by. The SM does this by copying the data structure representing the family of functions in the NFO (more details in section 10). Finally, the SM terminates the initialization process, and the user is left with a NFO of a particular collection.

7.2 Attestation Mechanism

When a user wants to call a Verifiable Function, the SM will be called to perform the attestation mechanism. The attestation mechanism begins with `START_ATTEST` and ends with `END_ATTEST`.

START_ATTEST In the beginning of the syscall, the SM will check a series of conditions: 1) the NFO is valid 2) the owner of the NFO is the current Uotchi. 3) no other attestation mechanism is being run tangentially. 4) the hash of the NFO stored in the SM matches the hash of the current NFO state (more details in section 6.7.) 5) the hash of called function matches one of the hashes of the Verifiable Functions the NFO has been committed to. 6) according to the transition matrix for valid function transitions, the current VF can be executed. If any of these conditions are not satisfied, the SM will refuse to perform the attestation mechanism. If all these conditions are satisfied, the SM will prepare the data structure for attestation within the SM-only memory detailed in listing 7; the attestation mechanism begins.

At the beginning of the attestation mechanism, the SM will disable all interrupts. Then, in order to remember where to jump back to after it finishes attestation, the SM will save the return address of the syscall. It will then compute the hash of the current NFO state and store it in the SM state. The SM will then copy the input of the function to its state and set up the register to match the architecture's calling convention. Finally, the SM will deactivate most hardware interrupts and jump back to unprivileged execution mode to the return address. The function will then run in user mode until it encounters the second syscall we introduce: `END_ATTEST`.

END_ATTEST `END_ATTEST` will update the NFO proof block given the Attestation Proof data structure (as described in section 10.3) which was populated during the attestation mechanism. Then, it will copy the attestation data saved in the SM, hash the memory located between `a` and `b` to obtain the function's measurement and also hash the final state of the NFO(). Similarly, it will hash the previous ProofBlock and store its hash. It will then produce a certificate of attestation by hashing `AttestationData` block and by producing a signature using the device's private key `Pk_D`. Finally, it will set the hardware interrupts back on, clean the SM state and returns to the address previously saved.

7.3 Exchange Protocol

Exchanging NFOs between devices is necessary: we want to be able to exchange NFOs virtually without needing to exchange devices physically. This operation requires special care. The space where NFOs travel between devices is especially difficult to guarantee that the NFOs wont be duplicated. As a result, we develop our own secure exchange protocol that relies on the SM. One of the first challenges we face is the fact that our devices work offline. This means that, when receiving and sending NFOs, a device cannot easily check from a

third-party platform whether or not the NFO is a duplicate someone forged. We patch this loophole by utilizing remote attestation and the SM to check NFOs being received and sent on a device.

When NFOs are traveling between devices during exchange, our Uotchis must ensure that the NFOs are not duplicated over that space. Without a properly defined secure exchange protocol, there are a plethora of attacks an adversary can make to try to create a valid duplicate. For example, a seller can duplicate their NFO before selling one to a buyer but then keeping their old one in their own device.

In this exchange protocol, let us define two users, the *Seller* and the *Buyer*, whose roles are defined as such: the *Seller* is selling the NFO while the *Buyer* is receiving the NFO. Let us also assume that these two users have both already agreed on exchanging the NFO (scams such as the *Seller* not truthfully sending the NFO is out of scope for our guaranteed security.)

OFFER The *Buyer* initiates the exchange by calling their SM to generate a nonce which is sent to the *Seller*. The nonce is a random value generated by the TRNG and is used to prevent replay attacks.

GIVE Now, the *Seller* calls their own sell syscall to prepare the NFO. The sell syscall will update the NFO's owner to be the *Buyer*, thus also updating the NFO hash stored in the SM, before the user sends the NFO over the network with the Buyer's nonce. Even if the *Seller* was an attacker who duplicated the NFO before evoking the send function, they could no longer interact with the duplicates on the device because the hash stored in the SM does not match with the duplicates. This renders any duplicates made by the *Seller* invalid.

RECEIVE Once the *Buyer* recieves the nonce and the NFO, the *Buyer* calls the receive syscall. During this syscall, the SM checks that the nonce is the one that was originally sent in the `OFFER` syscall. The SM also checks that it is receiving a NFO whose most recently executed verifiable function is in the GIVE syscall. Lastly, the SM checks the chain of ownership stored on the NFO by using the list of manufacturer keys we trust as described in section 5.3. After all these checks are performed, the SM can confirm that the exchange was done securely and accept the NFO. The SM will then get rid of the nonce used during that exchange.

8 NFO Applications

Our Uotchis enables the creation of a dynamic certificate of authenticity over virtual objects that also attests each action performed on the object. This means that users can be confident that their NFOs are non-fungible because they were created by trusted hardware executing verifiable function, and thus could not have been forged or duplicated along the process.

Unlike NFTs, which lack proof of authenticity over a virtual asset, NFOs pave the way for true, secure non-fungibility in the virtual world. In this section, we will describe four examples for how a NFO developer might use and build on top of our infrastructure.

Digital Art. There exists one glaring problem with digital art: insufficient remuneration for the artists. This is due to the lack of an effective way to sell digital piece as these can be easily reproduced. With digital art, people can easily "steal" virtual artwork by, duplicating the files, claiming ownership and maybe profiting off of other artists' work by selling the file to others. If the pieces' files don't have value, one solution is to sell an unique certificate of ownership for the art pieces. This certificate should have the important property of not being duplicatible and not easily reproducible. This is what spawned the idea of Non-Fungible Tokens (NFT) [18, 11, 12]. NFT are tokens that points to the artist's artwork (using, for instance, a URL pointing to a server hosting the art piece). These tokens and who own them is stored on a blockchain, or a distributed public ledger. Artists can produce NFTs from their work and publish them onto NFT marketplaces. The public ledger will also hold any traces of ownership changes. Buyers have already spent millions of dollars over this idea of ownership [12]. However, NFTs are limited: given any file or virtual artwork, anyone can create an NFT for it and sell it on one of the public markets. This stems from the fact that NFTs lack a mechanism to prove the authenticity of a piece of artwork; As a result, it is, yet again, up to the subjectivity of the community to decide whether or not the artwork is truly legitimate.

With NFOs, the certificate of authenticity can guarantee that the ownership was first transferred by the artists themselves. Indeed, if a malicious user was trying to create an NFO over the same piece of art using trusted hardware, then the associated proof of authenticity would be false, unless they were able to reproduce every step that led the artist to produce their work of art. This could make it as difficult as counterfeiting a physical painting.

In this implementation of our technology, the Uotchi will act as a drawing tablet, where artists build their NFOs artwork. In this example, we will outline a possible Verifiable Function a developer of a NFO art collection could make. Let's look at a rudimentary example. Here, the `drawLine()` function is the only function and can be called anytime after the initialization of a NFO.

```
drawLine(x0, x1, y0, y1) {
  dx = x1 — x0
  dy = y1 — y0
  D = 2*dy — dx
  y = y0

  for x from x0 to x1
    setBlack(x,y)
    applyGrey(x, y—1)
    applyGrey(x, y+1)
    applyGrey(x—1, y)
    applyGrey(x+1, y)
    if D > 0
      y = y + 1
      D = D — 2*dx
    end if
    D = D + 2*dy
}
```

Listing 1: Draw Line

Here, the developer must be aware that duplication of the NFO artwork is easy if a counterfeiter could just copy the placement of every single pixel onto their own NFO. To fix this issue, the "draw line" function must also have an underlying property where each stroke creates a unique mark to the stroke made. As a result, in our example, each particular stroke *builds* a unique drawing onto the NFO's canvas, making the process of reproducing such NFO harder. Even if we don't entirely remove the possibility of counterfeits, one would have to redraw the NFO from scratch in order to create a valid and identical NFO. Our Uotchi technology has made copying a more difficult endeavor than the NFT technology. Being a Verifiable Function, every time the "draw line" function is executed, the attestation mechanism is called to update the NFO's state and proof. The final product you get is a work of art NFO that can be traded and which authenticity can be easily and automatically.

Video Games. The property of non-fungibility is also useful in the video game industry, where players build, collect, purchase and trade digital assets. As a result, this is yet another area of demand where we can apply our Uotchis and NFOs technology. We call this simple example "CryptoCuties," a gamified virtual asset scheme similar to CryptoKitties and Tamagotchis described earlier in section 3.2. CryptoCuties are digital pets that you will be able to give presents to in order to increase their happiness, with an off chance that it hurts their happiness. The value of a CryptoCutie lies in having a high happiness level as well as unique aesthetic traits which can be generated using a random seed, like how it works in CryptoKitties. The data structure and function outlined below serves the purpose of showing developers how easy developing NFOs collections are and how it can be applied to video games. More complex and interesting combinations of functions will create a more interactive NFO collection, and that, a developer can define with their own imagination.

```
CryptoCutie:
  Byte[32] seed
  Int      happiness
```

Listing 2: CryptoCutie Data Structure

In order to create a CryptoCutie-specific asset on top of the NFO, the Uotchi will run the `GenerateCryptoCutie()` algorithm. This algorithm will define two elements; the seed and the `happiness` level. The seed is a string of bits generated by a TRGN. This seed can be seen as the CryptoCutie's DNA, as it determines the inherent physical traits of the CryptoCutie. We will derive the physical appearance of the CryptoCutie from the seed by defining bits individually and in tandem with other bits. For example, we could say the 17th bit is one which indicates whether or not the CryptoCutie has blue hair. These are traits the developer may define with creative freedom. The integer value of `happiness` will also be generated by the TRNG within a set range.

```
GenerateCryptoCutie() {
  CryptoCutie.seed = TRNG()
  CryptoCutie.happiness = TRNG()
  NFO.data[] = CryptoCutie

  syscall END_ATTEST
}
```

Listing 3: CryptoCutie Generation

Here we show how a developer might create Verifiable Functions specific to their NFO collection for users to interact with its state. The `GiftCryptoCutie()` function allows the user to interact with a CryptoCutie by randomly increasing or decreasing a CryptoCutie's `happiness` state every time it is executed. If the CryptoCutie dips below the happiness level, the CryptoCutie will die. In this game scheme, the meaning of death means that the NFO state `is_dead` will be changed to true. Once `is_dead` is changed to true, the SM of any Uotchi will not be able to further modify or exchange the NFO. The transition matrix of this example is simple, given that `GiftCryptoCutie()` is the only function.

```
GiftCryptoCutie() {
  Int random = TRNG()
  if(random % 17 == 0){
    CryptoCutie.happiness —= random
  } else {
```

```
    CryptoCutie.happiness += random
  }
  if(CryptoCutie.happiness < 0){
    NFO.is_dead = true
  }
  syscall END_ATTEST
}
```

Listing 4: Gift Function

As one can see, the development of such Verifiable Function can be straightforward and built-upon. The developer must only pay attention to the rules of Verifiable Functions as outlined in section 6.2 and section 6.4, which makes developing functions such as `GiftCryptoCutie()` simple. Developers can also leverage the transition matrix to include many different evolution paths for the CryptoCutie to take. Although `GiftCryptoCutie()` is simple, there lies the potential for the creation of a whole spectrum of functions to create an assortment of games and unique NFO collections.

Digital Coin. The usage of digital currency over the past few years has risen [1]. Most digital coins today utilize a blockchain to store and exchange funds. However, in this application, we will describe a scheme where our Uotchi becomes a wallet for offline digital currency. Owners of a NFO coin will be able to initialize, exchange and spend the digital coin as defined by the developer.

The initialization of such coin means that a NFO is generated on a device. However, the initialization of a digital coin requires more attention than the other examples because we do not want any user to be able to initialize however many coins as they want. As a result, we want the initialization of these NFO coins to be backed by a legitimate bank or a third-party authority which the people trust. In order to limit the generation of valid NFO coins to only be generated by these verified parties, we must have a special initialization function which displays proof that the NFO coin was developed by those verified parties.

```
Signature proof = Sign(Sk_bank, NFO.uid)
```

Listing 5: Bank Proof of Digital Coin Generation

This proof will be passed into the generation function and will be stored in the data field of the NFO. Users will be able to verify the signature, and thus the legitimacy of the coin, using the verified authority's public key. Along with the proof, the verified authority must also input the value they want the coin to be initialized as. The following function details the generation Verifiable Function.

```
GenerateCoin(Signature proof, int val) {
  int index = 0;
  for(Byte x : proof){
    data[index] = x;
    index++;
  }

  data[index] = Ints.toByteArray(val)
  syscall END_ATTEST
}
```

Listing 6: Digital Coin Generation

It is up to the bank to execute this function on their secure devices correctly and only once per device—and we can trust that it is in the bank's best interest to do so.

Another intuitive Verifiable Functions for a digital coin is to spend the coin. In our scheme, spending the coin is equivalent to exchanging the coin to another Uotchi. However, one main concern we must be aware of is the threat of double spending, where a user spends the coin multiple times. Fortunately, our NFOs already have a proper exchange protocol which protects against such an attack. The security of our exchange protocol can be found in section 7.3. The transition matrix between the generate and spend functions is as follows: Once generated, a coin can only be exchange by subsequent users.

9 Conclusion

In this paper, we presented a new form of non-fungible digital assets we call NFOs backed by custom secure hardware, our Uotchis. We show how developers might use our infrastructure to create their own family of virtual assets on top of our generic functions. We can guarantee the validity of the NFOs because all functions used to interact with the NFOs are ran within the attestation mechanism. We detailed the implementation for our Uotchi and present several use cases for our platform.

References

[1] BAREFOOT, K., CURTIS, D., JOLLIFF, W., NICHOLSON, J. R., OMOHUNDRO, R., ET AL. Defining and measuring the digital economy. *US Department of Commerce Bureau of Economic Analysis, Washington, DC 15* (2018).

[2] BEN MUNSTER. People's expensive nfts keep vanishing. this is why. https://www.vice.com/en/article/pkdj79/peoples-expensive-nfts-keep-vanishing-this-is-why. Accessed on 08.06.2022.

[3] BOURGEAT, T., LEBEDEV, I., WRIGHT, A., ZHANG, S., AND DEVADAS, S. Mi6: Secure enclaves in a speculative out-of-order processor. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (2019), pp. 42–56.

[4] CHOHAN, U. W. Non-fungible tokens: Blockchains, scarcity, and value. *Critical Blockchain Research Initiative (CBRI) Working Papers* (2021).

[5] COKER, G., GUTTMAN, J., LOSCOCCO, P., HERZOG, A., MILLEN, J., O'HANLON, B., RAMSDELL, J., SEGALL, A., SHEEHY, J., AND SNIFFEN, B. Principles of remote attestation.

International Journal of Information Security 10, 2 (2011), 63–81.

[6] COSTAN, V., AND DEVADAS, S. Intel sgx explained. *IACR Cryptol. ePrint Arch. 2016*, 86 (2016), 1–118.

[7] COSTAN, V., LEBEDEV, I., AND DEVADAS, S. Sanctum: Minimal hardware extensions for strong software isolation. In *25th USENIX Security Symposium (USENIX Security 16)* (2016), pp. 857–874.

[8] DAVID YAFFE-BELLANY. Thefts, fraud and lawsuits at the world's biggest nft marketplace. https://www.nytimes.com/2022/06/06/technology/nft-opensea-theft-fraud.html. Accessed on 08.06.2022.

[9] EVANS, T. M. Cryptokitties, cryptography, and copyright. *AIPLA QJ 47* (2019), 219.

[10] KUGLER, L. Non-fungible tokens and the future of art. *Communications of the ACM 64*, 9 (2021), 19–20.

[11] LEHDONVIRTA, V. Real-money trade of virtual assets: ten different user perceptions. *Proceedings of Digital Arts and Culture (DAC 2005), IT University of Copenhagen, Denmark* (2005), 52–58.

[12] MATTHIEU NADINI, LAURA ALESSANDRETTI, F. D. G. M. M. L. M. A., AND BARONCHELLI, A. Mapping the nft revolution: market trends, trade networks, and visual features, 2021.

[13] MUTHE, K. B., SHARMA, K., AND SRI, K. E. N. A blockchain based decentralized computing and nft infrastructure for game networks. In *2020 Second International Conference on Blockchain Computing and Applications (BCCA)* (2020), pp. 73–77.

[14] OPENSEA. Why are my items and collections delisted? https://support.opensea.io/hc/en-us/articles/1500010625362-Why-are-my-items-and-collections-delisted-. Accessed on 08.06.2022.

[15] PINTO, S., AND SANTOS, N. Demystifying arm trustzone: A comprehensive survey. *ACM Computing Surveys (CSUR) 51*, 6 (2019), 1–36.

[16] RAMAN, R., AND RAJ, B. E. The world of nfts (non-fungible tokens): The future of blockchain and asset ownership. In *Enabling Blockchain Technology for Secure Networking and Communications*. IGI Global, 2021, pp. 89–108.

[17] SHEPHERD, C., MARKANTONAKIS, K., AND JALOYAN, G.-A. Lira-v: Lightweight remote attestation for constrained risc-v devices, 2021.

[18] THWAITES, D. A token sale: Christie's to auction its first blockchain-backed digital-only artwork.

[19] UCI, U. E. I. Smart: Secure and minimal architecture for establishing a dynamic root of trust.

[20] VALEONTI, F., BIKAKIS, A., TERRAS, M., SPEED, C., HUDSON-SMITH, A., AND CHALKIAS, K. Crypto collectibles, museum funding and openglam: challenges, opportunities and the potential of non-fungible tokens (nfts). *Applied Sciences 11*, 21 (2021), 9931.

[21] VAN BULCK, J., MINKIN, M., WEISSE, O., GENKIN, D., KASIKCI, B., PIESSENS, F., SILBERSTEIN, M., WENISCH, T. F., YAROM, Y., AND STRACKX, R. Foreshadow: Extracting the keys to the intel {SGX} kingdom with transient out-of-order execution. In *27th {USENIX} Security Symposium ({USENIX} Security 18)* (2018), pp. 991–1008.

[22] VAN SCHAIK, S., KWONG, A., GENKIN, D., AND YAROM, Y. Sgaxe: How sgx fails in practice, 2020.

[23] WANG, Q., LI, R., WANG, Q., AND CHEN, S. Non-fungible token (nft): Overview, evaluation, opportunities and challenges. *arXiv preprint arXiv:2105.07447* (2021).

[24] WATERMAN, A., LEE, Y., AVIZIENIS, R., PATTERSON, D. A., AND ASANOVIC, K. The risc-v instruction set manual volume ii: Privileged architecture version 1.9. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-129* (2016).

10 Appendix

10.1 Basic Types

In order to describe the different data structures efficiently, we refer to a handful of abstract basic types. `Int` represents an integer. `Bool` represents a boolean. `Ptr` represents a pointer to an address in the device's memory. `Opcode` is an `Int` that represent a verifiable function Opcode. This is used by the user to refer to functions it wants to execute through the attestation mechanism (see section 7). `Byte[N]` represents an array of N bytes (or of undefined size if $N = \emptyset$). `TimeStamp` in an `Int` that represents a time stamp as outputted by the device's clock. `Hash` is a `Byte[N]` that represents the output type of the hash function used by the device as described in section 3.3.1. `Pk_D` is a `Byte[N]` that represents a device's public key as described in section 3.3.2. `Signature` is a `Byte[N]` that represents the output type of the Sign algorithm used by the device as described in section 3.3.2.

10.2 Security Monitor Data Structure

The SM has access to the device secret key (Sk_D) stored in the Device Secret Storing Section. It also stores some state in the Security Monitor Section. First the the current hash of the NFO to guarantee its integrity but also data required to perform the attestation scheme described in detail in section 7. This data is organized as follow:

```
SMAttestationData:
  Bool     is_attesting
  Ptr      return_address_attestation
  Hash     hash_state_NFO_in
  Hash     function_measurement
  Int      input_size
  Byte[]   input
```

Listing 7: SM Attestation Data Structure

```
SMData:
  SMAttestationData attest_data
  Hash              NFO_current_hash
```

Listing 8: SM Data Structure

10.3 Public Data Structures

These data structures are public accessible to the user and the SM and located in a platform-specific location. In particular, the SM is able to find them when needed after they've been populated by the user.

Trusted Manufacturers' Public Keys Declaration This describes the set of manufactures keys the SM can trust. Every time a user receives a NFO, the SM will use this data-structure to verify the chain of certificate used as an history of ownership.

```
TrustedManuDeclaration:
  Pk_M[] trusted_manufacturers
```

Listing 9: Trusted Manufacturers' Public Keys Declaration

Verifiable Functions' Declaration Each function that wants to be executed through the attestation mechanism needs to be declared to the SM by storing its signature using the following data structure in the Unprotected Section of the device's memory. At the state of the attestation mechanism, the SM must "measure" the function. It does this by hashing the function between pointers `a` and `b` and storing it in secure memory. This data is used by the SM to efficiently search for the declaration corresponding to a given function `opcode` when needed.

```
FunctionDeclaration:
  Opcode function_opcode
  Ptr    a #memory start
  Ptr    b #memory end
  Ptr    start #execution start
  Int    input_size
  Byte[] input
```

Listing 10: Function Declaration Data Structure

Function Family Declaration This data structure declares a Verifiable Function Family by describing the set of measurement for the corresponding Verifiable Function and a transition matrix to determine in which order the functions can be called. At the start of the attestation mechanism, the SM will use this data-structure to see whether or not the Verifiable Function can be called.

```
FunctionFamily:
  Int                  num_func
  Hash[num_func]       functions_measurement
  Bool[(num_func+4)^2] transition_matrix
```

Listing 11: Function Family Matrix

NFOs State Each Uotchi is able to store one NFO in its memory. The device's current NFO is stored in the Unprotected Section of the device's memory. The NFO is composed of two elements. First its state, representing the current view of the NFO and an attested that can be verified to prove that the NFO state was reached by executing a limited set of functions on secure hardware.

```
OwnerHistBlock:
  Pk manufacturer_pk
  Signature device_cert
  Pk_D device_pk
  Signature proof_of_handover
```

Listing 12: Virtual Asset Owner History Block

```
VirtualAssetState:
  Byte[32]         uid
  Int              depth_owner_hist
  OwnerHistBlock[] owner_hist
  Bool             is_dead
  Int              data_size
  Byte[]           data
  FunctionFamily   func_fam_measurement
```

Listing 13: Virtual Asset State Data Structure

```
VirtualAsset :
  VirtualAssetState   state
  AuthenticityProof[] proof
```

Listing 14: Virtual Asset Data Structure

Attestation Proof The proof stored aside the NFO data structure is generated and updated by the attestation mechanism executed by the SM. It is composed of the following entries:

```
AttestationData:
  Opcode   function_opcode
  Ptr      relative_pc_start
  Ptr      relative_pc_end
  Hash     hash_state_crypto_cutie_in
  Hash     hash_state_crypto_cutie_out
  Hash     function_measurement
  Hash     hash_previous_proof
  Int      input_size
  Byte[]   input
```

Listing 15: Attestation Data Structure

```
AuthenticityProof:
  Pk_D            prover
  AttestationData data
  Signature       sigma
```

Listing 16: Attestation Proof Data Structure