Detection of Illegally Duplicated Fish in a Fish It-Inspired Game Trading System Using Digital Signatures

Ferro Arka Berlian - 18223027
Program Studi Sistem dan Teknologi Informasi
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
Email: ferroarkab@gmail.com, 18223027@std.stei.itb.ac.id

***Abstract*—A player in a collectible fishing game may duplicate a fish certificate and try to trade the copy. This paper evaluates that problem in a Fish It-inspired simulation, not in the real game. The proposed mechanism combines Ed25519, a unique `fish_id`, a server-side ownership registry, certificate versions, and an atomic transfer. Four schemes were compared through 1,500 security trials, ten-way race tests, workloads of up to 10,000 fish, and a thread benchmark with 1 to 32 workers. The combined mechanism rejected every modified, forged, copied, stale, and double-traded certificate while accepting legitimate transfers. Exactly one request succeeded in each race. At 10,000 fish, median signing, verification, and complete validation took 74.60, 111.90, and 196.95 microseconds. Threaded verification remained near 3,000 checks per second without speedup, while independent-trade throughput decreased from 1,693 to 1,486 operations per second between 1 and 32 workers. Signatures authenticate issued data but cannot identify an exact copy. The registry establishes current ownership, and the atomic update prevents two buyers from consuming the same state.**

***Index Terms*—digital signature, Ed25519, fish duplication, game trading, ownership registry, atomic transfer, parallel verification**

I. Introduction

Online games often assign scarcity and exchange value to collectible items. In a fishing game, players may distinguish fish by species, rarity, weight, mutation, location, and other attributes. Public Fish It pages and community guides describe a game centered on catching and trading fish with such differences [1]–[3]. These sources provide the setting for this study. They are not evidence that the real Fish It game contains the vulnerability examined here.

The simulated problem begins when a client can copy or edit an item record. A rare fish can then appear in more than one inventory or be offered to several buyers. Accepting those copies weakens scarcity, creates conflicting ownership records, and makes trades unfair. The problem is especially awkward because two different failures are easy to confuse. An attacker may alter the contents of a certificate, or the attacker may copy a valid certificate without changing a byte.

A digital signature handles the first case well. If the name, rarity, weight, owner, or version changes after issuance, signature verification fails. It does not handle the second case by itself. The original and its exact copy contain the same message and signature, so both pass the same cryptographic verification. A valid signature says that an authority signed a statement. It does not say that the presenter is still the current owner.

The research question is how a game trading service can combine signed item data with authoritative state to reject modified, forged, copied, stale, and double-traded fish before ownership changes. The proposed design uses a signed fish certificate with a permanent identifier and versioned ownership data, coupled with an atomic registry transfer. Four schemes isolate the effects of signatures and registry checks. The evaluation covers 15 security scenarios, sequential latency and storage, thread-level verification, independent trades, and conflicting requests for the same fish.

The implementation is a local Python simulation. It does not reverse engineer Roblox, inspect Fish It servers, reproduce the real database, or claim an officially confirmed Fish It defect. This boundary matters because community descriptions can motivate a virtual-item model, but the security conclusions in this paper come from the simulation and its recorded experiments.

II. Background

A. Virtual Fish and Online Game Trading

Fish It-inspired item fields make the example concrete. Community indexes describe fish using names, rarity categories, catch locations, mutations, and values [4], [5]. The simulation retains name, rarity, weight, and mutation as signed attributes. It adds fields that the research mechanism needs: `fish_id`, current and original owner identifiers, an issue time, a schema version, and a certificate version.

Cheating in online games includes actions that violate the intended rules or exploit weaknesses in system design [6]. Markets for virtual goods also create incentives to move or sell game assets outside ordinary play [7]. This study does not attempt to classify every form of cheating. It focuses on one narrow consistency rule: one authenticated fish identity at one version may move from its current owner only once.

B. Digital Signatures and Ed25519

A digital signature binds a message to a private key. The signer produces a signature, and a verifier checks it with the corresponding public key [8]. FIPS 186-5 specifies approved digital-signature methods and their role in detecting unauthorized modification and authenticating a signatory [9]. Sound key management keeps the authority private key away from clients and limits public-key selection to trusted key identifiers [10]. Other public-key signature constructions include ElGamal, DSA, ECDSA, and Schnorr signatures [11].

The simulation uses Ed25519, the EdDSA variant over edwards25519 specified in RFC 8032 [12]. The Python `cryptography` implementation supplies key generation, signing, and verification [13]. The game authority holds the private key. The trade service receives a map from trusted `key_id` values to public keys. Players receive certificates and signatures, never the signing key.

For a certificate body m, private key sk, and signature σ, issuance computes

$$\sigma = \text{Ed25519.Sign}(sk, m). \tag{1}$$

The trade service checks $\text{Ed25519.Verify}(pk, m, \sigma)$ before consulting ownership state. A changed field produces different message bytes and therefore fails verification unless an attacker can create a new valid signature.

C. Deterministic Certificate Encoding

Signing structured data requires a stable byte representation. The implementation serializes JSON with sorted keys, fixed separators, UTF-8 encoding, and integer values for weight, time, and version. This produces deterministic bytes for the supported certificate structure. The encoding is specific to this implementation and is not presented as compliance with a broader canonicalization standard.

This matters because a verifier must recreate the exact bytes signed by the authority. Differences in whitespace or property order should not change the message. The claim here is limited to consistent encoding of the certificate structure supported by the implementation.

D. Why Signatures Cannot Detect Exact Copies

Suppose a valid envelope $E = (m, \sigma)$ is copied from one inventory to another. Verification receives the same public key, message, and signature for both files. It must return the same result. No property of a conventional signature labels one byte sequence as the original physical instance and another as its copy.

The system therefore treats duplication as an ownership-consistency problem. The signature authenticates the authority's statement at issuance. The registry answers whether the stated owner and version are still current. An atomic transfer ensures that two requests cannot both consume that state. The experiments below test whether these checks work together as intended.

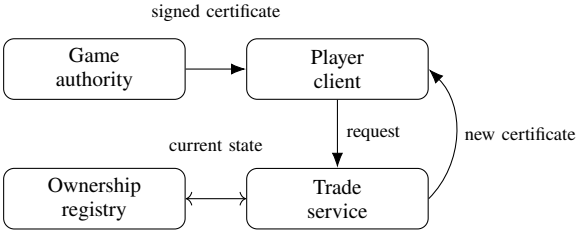

Fig. 1. Components of the simulated trading system. The client is outside the trusted boundary; the authority, service, and registry are trusted.

TABLE I
SIGNED FISH CERTIFICATE FIELDS

Field	Purpose
`schema_version`	Format version for parsing
`fish_id`	Permanent unique identity
`name`, `rarity`	Collectible attributes
`weight_grams`	Integer fish weight
`mutation`	Signed mutation label
`original_owner_id`	First owner, unchanged
`current_owner_id`	Owner stated at issuance
`caught_at`, `issued_at`	Integer Unix times
`version`	Transfer sequence number

III. System and Threat Model

A. Components and Trust Boundary

Figure 1 shows the four components. The authority creates fish identities and signs certificates. A player client stores a certificate and submits it with a seller identity. The trade service verifies the signature and applies trading rules. The ownership registry stores the canonical owner, version, status, and certificate hash. Roblox Data Stores illustrate the broader platform idea of persistent server-side state, although this simulation uses SQLite instead [14].

The attacker controls a simulated client. The attacker can read, copy, and edit certificates; create arbitrary records; attach random signatures; reuse old certificates; substitute one fish record for another; sign with an unknown key; and send sequential or concurrent trade requests. The attacker cannot obtain the authority private key, edit the registry directly, alter the trade service, break Ed25519, or make the trusted authority sign arbitrary data.

B. Certificate and Registry Records

Table I lists the signed certificate fields. The permanent `fish_id` is a UUID generated at creation. The original owner never changes. The current owner and version change after a successful transfer. The envelope stores the certificate body, the literal algorithm name `Ed25519`, a `key_id`, and a base64url signature.

Figure 2 shows a certificate body after two completed transfers. The signature does not appear inside this body. The envelope stores it beside `signature_algorithm` and `key_id`, so verification covers the body without recursively signing the signature field.

```
{
  "schema_version": 1,
  "fish_id": "028f5b21-7a8d-7d33-a0c1-4f54561c8e31",
  "name": "Nessie",
  "rarity": "Limited",
  "weight_grams": 124750,
  "mutation": "Lightning",
  "original_owner_id": "player-001",
  "current_owner_id": "player-003",
  "caught_at": 1781300000,
  "issued_at": 1781305000,
  "version": 2
}
```


Fig. 2. Example signed fish certificate body.

The SQLite registry has one row per `fish_id`, enforced by a primary key. Each row stores `current_owner_id`, `current_version`, `status`, and a SHA-256 hash of the current envelope. Status values include `ACTIVE`, `TRADE_LOCKED`, `QUARANTINED`, and `RETIRED`; the experiments require `ACTIVE`. The implementation uses explicit transactions as the boundary for each registry update.

C. Attack Definitions

A *modified fish* has at least one signed field changed after issuance. A *forged fish* has a record or signature not produced by the trusted authority. A *copied fish* is an exact valid identity presented from another inventory or by another seller. A *stale certificate* was once valid but no longer matches the registry owner or version. A *double-traded fish* is the same owner and version offered to more than one buyer.

IV. Proposed Duplicate-Detection Mechanism

A. Issuance

At fish creation, the authority generates a UUID, constructs the certificate body, serializes it deterministically, and signs the resulting bytes. The service wraps the body with its algorithm, key identifier, and signature. It then inserts one registry row containing the fish identity, owner, version 1, active status, and envelope hash.

Issuance binds valuable attributes to the authority statement. For example, changing a normal fish into a limited fish or adding a rare mutation changes the signed bytes. Reusing a signature on a random `fish_id` also fails because the identifier is inside the signed body.

B. Pre-trade Checks

The trade service applies checks in a fixed order. It confirms that the envelope declares Ed25519, finds the public key for `key_id`, and verifies the signature. It then requires the certificate's current owner to equal the submitted seller. Next, it fetches `fish_id` from the registry and requires an active row with the same seller and version. Any failed check rejects the request before ownership changes.

This ordering keeps cryptographic failure separate from state failure. Modified data, random signatures, unknown keys, and mismatched records fail signature verification. An exact copy remains cryptographically valid, but a wrong seller fails the certificate-owner check. A formerly valid certificate fails after a completed transfer because the registry contains a different owner or version.

Require: Certificate envelope E, seller s, buyer b
Verify the Ed25519 signature in E
if the signature or signing key is invalid **then**
  return reject
end if
Read fish state r from the ownership registry
if $E.owner \neq s$ or $r.owner \neq s$ **then**
  return reject
end if
if $E.version \neq r.version$ or $r.status \neq active$ **then**
  return reject
end if
Create a new certificate for b at version $r.version + 1$
Begin an immediate database transaction
Compare the owner, version, and status again
if the registry state has changed **then**
  Roll back and **return** reject
end if
Atomically update the owner and version
Commit and **return** the newly signed certificate

Fig. 3. Secure certificate validation and atomic transfer pseudocode.

TABLE II
Compared trading schemes

Scheme	Signature	Registry	Atomic
A: Plain record	No	No	No
B: Registry only	No	Yes	Yes
C: Signature only	Yes	No	No
D: Proposed	Yes	Yes	Yes

C. Atomic Transfer and Reissuance

After pre-validation, the service prepares a new certificate for the buyer with version $v + 1$ and a new issue time. It signs that certificate, begins a SQLite `BEGIN IMMEDIATE` transaction, reads the current row, and checks the expected owner, version, and active status again. The update includes those same values in its `WHERE` clause. A transfer commits only when exactly one row changes; otherwise, it rolls back.

Figure 3 summarizes the validation and atomic transfer procedure.

The repeated state check matters under concurrency. Ten requests may all begin with the same valid envelope. They cannot all commit against the same registry state. Once one request changes the owner and version, later requests see a mismatch and fail. The successful buyer receives the only certificate whose state matches the registry.

V. Experimental Method

A. Compared Schemes

The experiment compares the four schemes in Table II. Scheme A accepts records without cryptographic or registry validation. Scheme B checks registry identity, owner, version, and status but does not authenticate the presented attributes. Scheme C verifies the signature but does not check current ownership. Scheme D is the proposed combined system.

Five comparison cases isolate normal transfer, a modified attribute, an exact copy used by the wrong seller, stale-certificate reuse, and the second request in a sequential double trade. This small matrix explains which protection produces each outcome. The full Scheme D evaluation then runs scenarios S1 through S15.

B. Security Scenarios and Metrics

The scenarios cover legitimate transfer (S1), edits to species, rarity, weight, and mutation (S2 through S5), a random identifier with a copied signature (S6), a random signature (S7), an exact certificate copied to another inventory (S8), stale reuse (S9), a manually increased version (S10), sequential double trade (S11), ten simultaneous buyers (S12), a certificate paired with another record (S13), an unknown signing key (S14), and valid transfer through multiple owners (S15).

Each scenario ran 100 trials, producing 1,500 recorded cases. For invalid requests, a pass means rejection; for legitimate requests, a pass means acceptance. S12 is recorded as a pass only when exactly one of ten requests succeeds. The reported metrics are passed cases, false-acceptance rate, false-rejection rate, and accepted concurrent trades per fish.

C. Performance Procedure

Performance workloads contained 100, 1,000, and 10,000 issued fish. Signing was measured once per issued certificate. Signature verification and complete trade validation used 30 measured repetitions per workload. Timing used `time.perf_counter_ns`; results report mean, median, standard deviation, p95, and p99. Certificate size is the UTF-8 byte length of the deterministic envelope JSON.

The recorded environment was Windows 11, 64-bit CPython 3.14.5, `cryptography` 48.0.0, and an in-memory SQLite database. The program requested WAL mode, but the result log records the request rather than a confirmed persistent WAL journal. The parallel log identified an Intel64 Family 6 Model 154 processor with 16 logical CPUs. It did not capture the exact processor name, physical-core count, or memory configuration, so the values characterize this machine rather than a general deployment.

D. Parallelism Procedure

The parallel benchmark separates three workloads. The verification workload repeatedly checks one immutable valid certificate and does not access the registry. The independent-trade workload gives every request a different fish, seller, and buyer. Its registry operations share one SQLite connection and one process-wide reentrant lock. The contention workload sends one valid owner and version to several buyers, so exactly one request should succeed.

Verification and independent trading used 1, 2, 4, 8, 16, and 32 threads. Each setting ran five synchronized batches. A verification batch contained 10,000 checks, giving 50,000 observations per worker count. An independent-trade batch contained 1,000 unique fish, giving 5,000 observations per count. A barrier released all worker threads together. Wall-clock throughput starts at that release and ends after the last operation; thread creation is outside the interval. Per-operation latency starts when a worker calls the operation, so it includes lock waiting but not executor queue time.

Same-fish contention used 2, 4, 8, 16, and 32 threads for 100 trials at each count. Every trial created a fresh fish and submitted one request per buyer. The recorded metrics are accepted and rejected requests, median batch completion time, and p99 request latency. Throughput is reported for verification and independent trades. Speedup is throughput divided by the corresponding one-worker result.

VI. Results and Discussion

A. Scheme Comparison

Table IV gives the security outcome for the five diagnostic cases. All schemes accepted the legitimate trade. Plain records also accepted every attack. Registry-only validation rejected copied, stale, and sequential duplicate ownership, but it accepted a modified signed attribute because it had no signature check. Signature-only validation rejected modification, but it accepted the exact copy, stale reuse, and second sequential trade. Scheme D produced the expected secure outcome in all five cases.

The comparison shows why a single detection percentage would be insufficient. Signatures and registry state address different problems. Scheme C cannot reject a byte-for-byte clone because the copied signature remains valid. Scheme B knows who owns an identity, but it cannot detect a client changing the fish's rarity or weight. The combined system requires both checks.

B. Security Scenarios

All 1,500 scenario trials matched the expected decisions. S1 accepted 100 legitimate current-owner trades. S2 through S11 and S13 through S14 rejected all 1,200 invalid requests. S15 accepted all 100 multi-owner trade chains. No trial produced a false acceptance or false rejection.

Table V groups the scenarios by the mechanism responsible for the decision. A failed Ed25519 verification rejected modified attributes, random identifiers paired with copied signatures, random signatures, changed versions, substituted records, and unknown signing keys. Certificate and registry ownership checks rejected the exact copy in another inventory and stale reuse. Registry state rejected the second sequential transfer. These results are conditional on the threat model, especially the secrecy of the authority private key and the integrity of the server and database.

C. Concurrent Double Trading

For S12, every trial sent the same valid seller certificate to ten buyers using a ten-worker thread pool. Across 100 trials, each group had one accepted request and nine rejected requests. The average was therefore 1.00 accepted and 9.00 rejected trades per fish. The registry lock, immediate transaction, expected owner and version, and conditional update reduced the race to one committed state transition.

TABLE III
DETAILED SECURITY SCENARIOS FOR SCHEME D

ID	Scenario	Accepted	Rejected	Decision reason
S1	Legitimate fish traded by current owner	100	0	Accepted
S2	Species changed after signing	0	100	Bad signature
S3	Rarity changed after signing	0	100	Bad signature
S4	Weight increased after signing	0	100	Bad signature
S5	Valuable mutation added after signing	0	100	Bad signature
S6	Random `fish_id` with copied signature	0	100	Bad signature
S7	Randomly generated signature	0	100	Bad signature
S8	Exact certificate copied to another inventory	0	100	Certificate-owner mismatch
S9	Old seller reuses certificate after transfer	0	100	Registry-owner mismatch
S10	Certificate version manually increased	0	100	Bad signature
S11	Same fish offered sequentially to two buyers	0	100	Second seller no longer current
S12	Same fish offered concurrently to ten buyers	100	0	Exactly one accepted per trial
S13	Valid certificate paired with another record	0	100	Bad signature
S14	Unknown public key used for signing	0	100	Bad signature
S15	Legitimate transfers through multiple owners	100	0	All three transfers accepted

TABLE IV
SECURITY OUTCOME OF THE SCHEME COMPARISON

Scenario	A: Plain	B: Registry	C: Signature	D: Proposed
Legitimate current-owner trade	Secure	Secure	Secure	Secure
Modified signed attribute	Insecure	Insecure	Secure	Secure
Exact copy presented by wrong seller	Insecure	Secure	Insecure	Secure
Old certificate reused after trade	Insecure	Secure	Insecure	Secure
Sequential double trade, second request	Insecure	Secure	Insecure	Secure

TABLE V
SCHEME D SECURITY RESULTS, 100 TRIALS PER SCENARIO

Scenarios	Category	Passed	Errors
S1	Legitimate trade	100	0
S2–S7	Modified or forged	600	0
S8–S11	Copy, stale, double	400	0
S12	Concurrent, one winner	100	0
S13–S14	Substitution or key	200	0
S15	Multi-owner chain	100	0
Total		1500	0

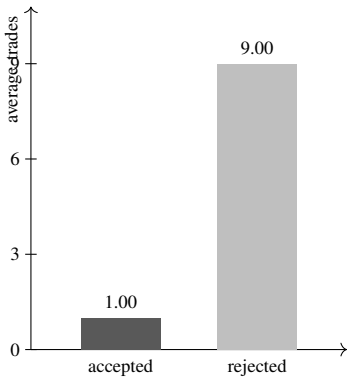

Fig. 4. Average outcome of ten concurrent requests for one fish across 100 trials.

This does not mean that Ed25519 prevented the race. Every concurrent request began with a valid signature. The decisive control was atomic consumption of registry state. The signature ensured that each request referred to authority-issued data; the transaction determined which request could still act on that data.

D. Latency and Size

Table VI reports the measured latency distributions. At the 10,000-fish workload, median signing took 74.60 µs, median verification took 111.90 µs, and median complete validation took 196.95 µs. The corresponding means were 88.01, 118.08, and 222.73 microseconds. The highest recorded p99 in the table was 756.50 µs for complete validation at 10,000 fish.

The medians do not increase monotonically with workload. That is plausible because the registry uses a primary-key lookup in a small in-memory database, while interpreter scheduling and measurement noise remain visible at microsecond scale. The experiment does not establish asymptotic scaling or production throughput. It does show that, under the recorded setup, adding signature and registry checks kept median complete validation below a quarter millisecond for all three workloads.

The serialized certificate envelope was 485 bytes at every workload and percentile. Workload size does not change the fields in one certificate, so the constant result is expected. Database storage was not measured separately. The 485-byte value should therefore be read as certificate-transfer overhead, not total per-fish server storage.

E. Parallelism and Contention

Table VII shows that adding threads did not increase throughput in this implementation. Verification processed

TABLE VI
PERFORMANCE RESULTS IN MICROSECONDS

Fish	n	Op.	Mean	Median	p95	p99
100	100	Signing	76.28	70.75	97.70	112.30
100	30	Verification	137.93	129.00	180.70	229.20
100	30	Complete validation	245.44	227.35	325.80	431.70
1000	1000	Signing	88.98	77.40	145.70	215.60
1000	30	Verification	138.98	127.05	197.10	269.10
1000	30	Complete validation	253.30	227.30	454.80	533.90
10000	10000	Signing	88.01	74.60	171.60	267.40
10000	30	Verification	118.08	111.90	141.80	248.80
10000	30	Complete validation	222.73	196.95	304.40	756.50

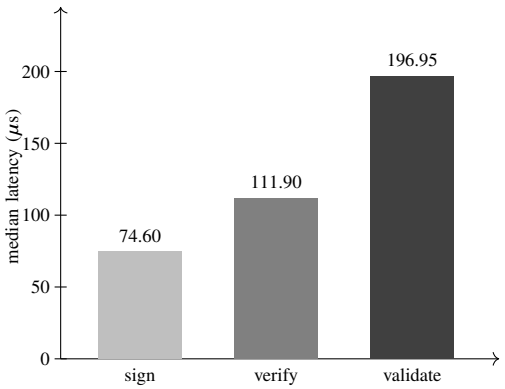

Fig. 5. Median latency at the 10,000-fish workload.

3,017.87 checks per second with one worker and 2,973.21 with 32 workers, a speedup of 0.985. The median stayed near 320 µs, but p99 rose from 0.54 ms to 126.20 ms. This tail growth is consistent with scheduling stalls under heavy thread counts even though the typical call time changed little.

The 320.30 µs single-worker median is not directly interchangeable with the 111.90 µs verification result in Table VI. The first comes from 50,000 sustained checks in the parallel harness; the second comes from 30 isolated measurements in the workload-size experiment. Parallel speedup is therefore calculated only against the one-worker result from the same harness.

Independent-fish throughput fell from 1,693.26 to 1,485.77 trades per second, a 0.877 speedup. Median latency increased from 0.57 ms to 21.06 ms. These requests did not contend for the same fish, but they did contend for the shared registry connection and its reentrant lock. The result is therefore a property of this thread-based Python and SQLite design. It does not show that Ed25519 verification is inherently sequential.

The contention tests preserved the ownership invariant at every worker count. Each of the 500 trials had one accepted request. At 32 workers, 100 requests succeeded and 3,100 were rejected across 3,200 submissions. Median batch completion increased from 1.95 ms at two workers to 25.53 ms at 32. Atomicity remained correct while latency grew with the number of losing requests.

TABLE VII
THREAD-LEVEL PARALLEL PERFORMANCE

Workers	Verify/s	Verify p99 (ms)	Trade/s	Trade med. (ms)
1	3017.87	0.54	1693.26	0.57
2	3029.44	16.36	1603.72	0.60
4	3020.09	31.20	1519.21	1.02
8	3024.38	32.77	1427.04	5.36
16	2984.11	94.33	1487.15	10.58
32	2973.21	126.20	1485.77	21.06

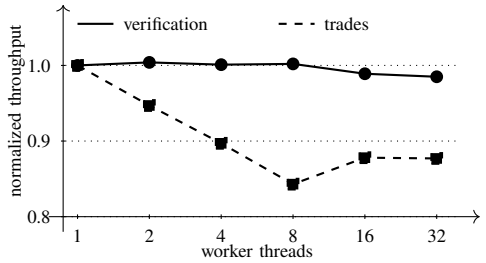

Fig. 6. Measured throughput relative to one worker. Neither workload gained thread-level speedup.

VII. LIMITATIONS

The main limitation is scope. This is a Fish It-inspired Python model, not an integration with Fish It or Roblox. It does not test network behavior, platform APIs, real inventory formats, or a production deployment. Public and community sources support only the gameplay context.

The authority, trade service, registry, and private key are trusted in this model. A stolen private key could produce valid fraudulent certificates. A compromised service or database could bypass ownership checks or rewrite canonical state. The design also cannot detect duplicate identities that a flawed trusted issuer deliberately signs and registers as separate legitimate fish.

The experiment uses one process and an in-memory SQLite database protected by a Python reentrant lock. This lock deliberately serializes registry access, including trades for different fish. The parallel benchmark does not test process pools, native worker services, multiple database connections, or a distributed database. It also cannot isolate how much of the verification plateau comes from Python scheduling, the cryptographic binding, the small operation size, or the operating system. A production study would need separate processes or hosts, durable storage, network delay, retries, and operational key management.

The environment log identifies 16 logical CPUs but omits the exact processor name, physical-core count, memory, power state, and background workload. The 32-thread test is oversubscribed. Signing repetitions also differ from verification and validation repetitions. These measurements support a local comparison, not a capacity forecast for a live game.

VIII. CONCLUSION

The experiments show that Ed25519 and the ownership registry have separate roles. Ed25519 authenticated the issued

fish data and rejected post-issuance changes, random signatures, record substitution, and unknown keys. It could not distinguish an exact certificate from its byte-for-byte copy. The registry checked the current state, certificate versions made old statements stale, and the atomic conditional update allowed only one request to consume a seller and version.

Within the stated threat model, the proposed system matched the expected outcome in all 1,500 security trials. It accepted exactly one winner in every race, including 500 additional contention trials with up to 32 buyers. Median complete validation remained between 196.95 and 227.35 microseconds in the sequential workload, and each envelope was 485 bytes. The thread benchmark found no parallel speedup: verification stayed near 3,000 checks per second, while independent-trade throughput at 32 workers was 87.7 percent of the one-worker result. This finding is limited to the tested runtime and shared registry design. The security conclusion is broader within the threat model: a signature proves what the authority issued, while registry state and atomic transfer decide whether that statement can still be used.

Source Code Link at GitHub

https://github.com/varisgoated/Detecting-Illegally-Duplicated-Fish-Paper

Acknowledgment

All praise and gratitude belong to Allah SWT, whose blessings and grace enabled the author to complete this paper. Sincere appreciation is extended to Dr. Ir. Rinaldi Munir, M.T., lecturer of II4021 Cryptography, for his teaching and guidance throughout the course. Deep gratitude is also expressed to the author's parents for their constant support and encouragement. Also thanks to the author's friends who took the time to give fresh perspectives, provide feedback, and help improve the paper.

References

[1] Fish Atelier, "Fish it!" Roblox, 2026. [Online]. Available: https://www.roblox.com/games/121864768012064/Fish-It

[2] S. Aruya, "Fish it trading guide," TechWiser, Mar. 2026. [Online]. Available: https://techwiser.com/fish-it-trading-guide/

[3] Fish It Wiki Staff, "Fish it trading value list (2026): Roblox item prices," Fish It Wiki, unofficial community source, Jan. 2026. [Online]. Available: https://fishit.web.id/guides/trading-values/

[4] Fish It Wiki Team, "Fish mutations guide: Shiny, abyssal and more," Fish It Wiki, unofficial community source, Jun. 2026. [Online]. Available: https://fishit.web.id/guides/mutations/

[5] FishIt-Wiki.com, "Fish it all fish locations and rarity," Unofficial Fish It community wiki, 2026. [Online]. Available: https://fishit-wiki.com/fish/

[6] J. Yan and B. Randell, "A systematic classification of cheating in online games," in *Proceedings of the 4th ACM SIGCOMM Workshop on Network and System Support for Games*. ACM, 2005, pp. 1–9.

[7] E. Lee, J. Woo, H. Kim, and H. K. Kim, "No silk road for online gamers! using social network analysis to unveil black markets in online games," in *Proceedings of the 2018 World Wide Web Conference*, 2018, pp. 1825–1834. [Online]. Available: https://arxiv.org/abs/1801.06368

[8] R. Munir, "Tanda-tangan digital," Materi Kuliah II4021 Kriptografi, Institut Teknologi Bandung, 2026. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Kriptografi-dan-Koding/2025-2026/28-Tanda-tangan-digital-STI-2026.pdf

[9] National Institute of Standards and Technology, "Digital signature standard (DSS)," National Institute of Standards and Technology, FIPS 186-5, 2023. [Online]. Available: https://csrc.nist.gov/pubs/fips/186-5/final

[10] E. Barker, "Recommendation for key management: Part 1—general," National Institute of Standards and Technology, NIST Special Publication 800-57 Part 1 Revision 5, 2020. [Online]. Available: https://csrc.nist.gov/pubs/sp/800/57/pt1/r5/final

[11] R. Munir, "Elgamal signature scheme, DSA, ECDSA, dan schnorr signature scheme," Materi Kuliah II4021 Kriptografi, Institut Teknologi Bandung, 2026. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Kriptografi-dan-Koding/2025-2026/29-Elgamal-Signature-DSA-ECDSA-Schnorr-Signature-Scheme-STI-2026.pdf

[12] S. Josefsson and I. Liusvaara, "Edwards-curve digital signature algorithm (EdDSA)," Internet Research Task Force, RFC 8032, Jan. 2017. [Online]. Available: https://www.rfc-editor.org/rfc/rfc8032.html

[13] The Cryptography Developers, "Ed25519 signing," Cryptography Documentation, 2026, accessed June 13, 2026. [Online]. Available: https://cryptography.io/en/latest/hazmat/primitives/asymmetric/ed25519/

[14] Roblox Corporation, "Data stores," Roblox Creator Hub Documentation, 2026. [Online]. Available: https://create.roblox.com/docs/cloud-services/data-stores

Statement

I hereby declare that this paper is my own work, not an adaptation, paraphrase, or translation of someone else's paper, and not plagiarism.

Bandung, 19 Juni 2026

Ferro Arka Berlian
18223027