PASETO as a Structurally Secure Alternative to JWT: Implementation and Security Evaluation Against Algorithm Confusion Attacks

Sendi Putra Alicia – 18223063
Program Studi Sistem dan Teknologi Informasi
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: 18223063@std.stei.itb.ac.id

***Abstract*—JSON Web Token (JWT) is the de-facto standard for stateless authentication, yet its security model lets the token itself declare how it should be verified through the header alg field. This design has produced a recurring class of algorithm confusion attacks, most notably alg:none and the RS256-to-HS256 confused-deputy. This paper presents a design, implementation, and experimental security evaluation of PASETO v4.public as a structurally secure alternative. We implement PASETO v4.public (Ed25519 with Pre-Authentication Encoding) directly from the specification, build a JWT subsystem with both a best-practice and a deliberately naive verifier, and execute the attacks against all targets. Results show that the naive JWT verifier is forged by both alg:none and RS256-to-HS256, whereas PASETO rejects every attempt because it has no negotiable algorithm field and binds the cryptographic contract to a non-negotiable version-purpose prefix. A complementary survey of five real library and version combinations across Python and Node.js finds that current implementations already neutralize the classic forges, but only through accumulated, implementation-specific mitigations and correct calling conventions, while PASETO attains the same result by construction. We further benchmark signing and verification speed and compare token sizes across HS256, RS256, ES256, and PASETO v4.public. The findings give developers concrete evidence for choosing PASETO where token integrity and resistance to misconfiguration are critical.**

Keywords*—*PASETO; JSON Web Token; algorithm confusion; token authentication; Ed25519; cryptographic protocol design

I. Introduction

Stateless, token-based authentication has become the backbone of modern web and API security. Instead of keeping session state on the server, the issuer hands the client a self-contained, cryptographically protected token that the verifier can validate on every request. JSON Web Token (JWT), standardized in RFC 7519, is by far the most widely deployed format and underpins OAuth 2.0, OpenID Connect, and countless bespoke APIs.

The strength of any such token rests entirely on the integrity of its verification step: a forged token that passes verification is a full authentication bypass. Here JWT carries a structural weakness. Each token embeds a header whose alg field announces the algorithm the verifier should use. In effect, the token tells the verifier how to check the token, and many libraries historically trusted that answer. This has produced a durable family of algorithm confusion attacks, including the alg:none bypass, in which the signature is stripped and the algorithm set to none, and the RS256-to-HS256 confused-deputy, in which an asymmetric verifier is tricked into treating a public key as a symmetric HMAC secret.

PASETO (Platform-Agnostic Security Tokens) was designed as a direct response to this class of failures. Rather than offering a menu of negotiable algorithms, PASETO fixes the cryptographic suite to a small set of versioned, opinionated protocols, and binds the choice to a non-negotiable version and purpose prefix carried at the front of every token. The claim that motivates this study is that PASETO removes the algorithm confusion attack surface not by adding checks, but by making the vulnerable pattern impossible to express.

The stakes are high because these tokens routinely encode authorization decisions. A field such as role or scope inside the payload often determines whether a request is treated as an ordinary user or an administrator. Consequently, a verification bypass is not a partial weakness but a direct path to privilege escalation, and the affected surface is large: JWT verification sits in front of a substantial share of contemporary APIs and single-sign-on flows. The recurring nature of algorithm confusion, resurfacing across many libraries and years, suggests the problem is rooted in the format's design philosophy rather than in any single buggy implementation.

This paper evaluates that claim empirically. The specific objectives are: (1) to implement PASETO v4.public directly from the specification and a comparable JWT subsystem; (2) to implement and execute representative algorithm confusion attacks against both schemes; (3) to quantify the resilience of each scheme through an attack-success matrix; (4) to survey real, deployed JWT libraries across languages and versions to see which attacks still succeed in practice; and (5) to benchmark the signing/verification performance and token size of JWT (HS256, RS256, ES256) against PASETO v4.public, and to discuss the resulting deployment trade-offs.

The contribution of this work is therefore not a literature survey but a working implementation and a reproducible experiment. All code, attacks, and measurement scripts are released so that the results can be independently verified.

II. Background and Related Work

A. Token-Based Authentication and JWT

A JWT is a string of three Base64URL-encoded parts joined by dots: a header, a payload, and a signature. The header is a JSON object that always contains an alg field, for example HS256 (HMAC-SHA-256, symmetric), RS256 (RSA-PKCS1v1.5 with SHA-256, asymmetric),

or ES256 (ECDSA over P-256). The payload holds the claims, and the signature authenticates the concatenation of the encoded header and payload under the algorithm named in the header.

This indirection, where the verifier consults a token-controlled field to decide how to verify, is the root cause of the attacks studied here. The algorithms themselves (HMAC, RSA, ECDSA) are sound; the weakness lies in letting untrusted input select among them.

Concretely, a JWT header and the resulting token take the form below, where the first segment decodes to a JSON object whose alg value is fully attacker-visible and, in vulnerable verifiers, attacker-trusted:

```
header  = { "alg": "RS256", "typ": "JWT" }
payload = { "sub": "alice", "role": "user" }
token   = b64url(header) "." b64url(payload)
          "." b64url(signature)
```

An attacker who can edit the header and recompute a signature that the verifier will accept gains complete control over the claims, including privilege-bearing fields such as role.

B. Algorithm Confusion Attacks

alg:none. RFC 7518 defines a none algorithm for unsecured JWTs. If a verifier honors it, an attacker can set alg to none, drop the signature entirely, and submit arbitrary claims. A correctly configured library that pins the expected algorithm rejects such tokens, but permissive configurations accept them.

RS256-to-HS256 confusion. This is the canonical confused-deputy attack, first widely publicized in 2015. A server issues RS256 tokens and verifies them with its RSA public key, which is not secret. A vulnerable verifier exposes a single verify routine that selects the algorithm from the token. The attacker changes alg to HS256 and computes an HMAC over the forged header and payload using the server's public key bytes as the HMAC secret. Because the verifier feeds the same public key into an HMAC check, the forged signature matches and the token is accepted, despite the attacker never possessing the private key.

The attack succeeds through the following steps, all of which use only public information:

1) obtain the server's RSA public key, which is published for legitimate verification.

2) craft a header with alg set to HS256 and a payload with elevated claims (for example role=admin).

3) compute sig = HMAC-SHA256(public_key_PEM, b64url(header).b64url(payload)).

4) submit the token; a verifier that dispatches on the header's alg and reuses the public key as the HMAC secret accepts it.

The defect is not in HMAC or RSA but in the verifier's willingness to let the token choose the algorithm while reusing one key across algorithm families.

C. Threat Model

The adversary in this study is an unauthenticated network attacker who can read and modify tokens in transit or craft tokens from scratch, and who knows all public parameters including the issuer's public key and the token format. The adversary does not possess the server's private signing key (for asymmetric schemes) or its symmetric secret (for HS256 or PASETO local). Success is defined as producing a token that a verifier accepts while carrying claims the adversary was never authorized to assert. Side-channel and implementation-bug attacks on the underlying primitives are out of scope; the focus is the protocol-level attack surface created by algorithm negotiation.

D. PASETO and Protocol Versioning

PASETO replaces algorithm negotiation with protocol versioning. A token has the shape version.purpose.payload[.footer]. The version (v3 or v4 in the current specification) pins the entire cryptographic suite, and the purpose is either local (symmetric authenticated encryption) or public (asymmetric signatures). For v4.public the suite is fixed to Ed25519 signatures. Crucially, there is no header and no alg field: the version-purpose pair at the front of the token is the cryptographic contract, and it is not negotiable.

Integrity is protected by Pre-Authentication Encoding (PAE), which deterministically serializes the header constant, payload, footer, and an optional implicit assertion into a single byte string before signing. PAE length-prefixes every field, preventing an attacker from shifting bytes between fields (a canonicalization attack). The v4.public signing and verification can be summarized as:

```
m2  = PAE([ "v4.public.", payload, footer, impl ])
sig = Ed25519_Sign(sk, m2)            // 64 bytes
token = "v4.public." || b64url(payload || sig)

verify: split token, recompute m2,
        Ed25519_Verify(pk, sig, m2)
```

A v4.public verifier is configured for exactly one version-purpose pair. A token that does not begin with v4.public is rejected before any cryptographic step, and the only operation performed is an Ed25519 signature check against the configured public key. There is no branch on token-supplied algorithm metadata, so the confused-deputy pattern simply cannot be written.

The specification offers two current versions for different compliance needs. Version 4 uses Ed25519 for public tokens and an XChaCha20 with keyed-BLAKE2b construction for local tokens, and is recommended for general use. Version 3 substitutes NIST-approved primitives (ECDSA over P-384 and AES-CTR with HMAC-SHA-384) for environments that require FIPS-validated modules. Within a version, the purpose cleanly separates symmetric from asymmetric use: local tokens are encrypted and authenticated for a party that holds the shared key, while public tokens are signed for verifiers that hold only the public key. This study targets v4.public because it is the direct counterpart to JWT's asymmetric algorithms and the precise locus of the RS256-to-HS256 confusion.

E. Related Work

JWT vulnerabilities have been documented extensively in security-practitioner literature and standardized cheat sheets, and the RS256-to-HS256 confusion has driven mitigations in mainstream libraries such as mandatory algorithm pinning. PASETO's design rationale is captured in its specification and IETF draft. Prior work tends to discuss these schemes separately or descriptively; this paper instead implements both and measures attack

outcomes and performance side by side under identical conditions.

The recurrence of these flaws is itself instructive. After the original 2015 disclosure, equivalent issues continued to appear across independent libraries and language ecosystems for years, each time because a verifier trusted the token's declared algorithm or reused a key across algorithm families. The pattern is not a single bug but a repeated consequence of one design decision: making the verification algorithm a negotiable, token-controlled parameter.

This places JWT within a broader and well-known security anti-pattern often called excessive cryptographic agility. The same root cause, allowing one party or the message itself to negotiate the cryptographic mechanism, has produced downgrade and confusion failures in other protocols, where attackers force a peer toward a weaker or attacker-favorable option. The defensive lesson distilled from that history is to minimize negotiable choices and to make the secure configuration the only configuration. PASETO operationalizes exactly this lesson: by pinning the suite to a version and removing the algorithm field, it converts a runtime negotiation into a compile-time constant, which is why the present experiments find its attack surface empty rather than merely guarded.

Against this backdrop, the contribution here is to provide a small but complete and reproducible artifact that makes the abstract argument concrete: identical attacks, identical claims, run against both designs, with the outcomes and costs tabulated so that the structural difference can be observed rather than asserted.

III. Methodology

The evaluation comprises a from-scratch PASETO v4.public implementation, a JWT subsystem with two verifier models, an attack module, and three experiments. All components are written in Python 3.12 using the cryptography library for primitives (Ed25519, RSA-2048, ECDSA P-256) and PyJWT for standards-compliant JWT issuance. Table IV lists the environment and parameters.

TABLE IV. Experimental Setup

Component	Value
Language	Python 3.12
Primitives	cryptography (Ed25519, RSA-2048, P-256)
JWT library	PyJWT 2.x
PASETO	v4.public, implemented from spec
Benchmark N	20000 iterations/operation
Timer	time.perf_counter()

A. PASETO v4.public Implementation

PASETO v4.public is implemented directly from the specification rather than through a third-party library, to demonstrate that its security is structural. The implementation provides PAE with 64-bit little-endian length prefixes, Base64URL without padding, and Ed25519 sign/verify. It was validated with three self-tests: a round-trip test (sign then verify must succeed), a tamper test (any single-byte change to the token must fail verification), and a cross-key test (verification under a different public key must fail). All three pass.

PAE is the load-bearing detail. Given an ordered list of byte strings, PAE emits a 64-bit little-endian count, then for each element its 64-bit little-endian length followed by the element itself. The most significant bit of each length word is cleared to avoid any signed-integer ambiguity. Because every field is length-prefixed, an attacker cannot move bytes across the header, payload, footer, or implicit-assertion boundaries without changing the signed pre-image, which defeats canonicalization attacks. For v4.public the signed pre-image is PAE over the constant header v4.public., the payload, the footer, and the implicit assertion, and the 64-byte Ed25519 signature is appended to the payload before Base64URL encoding.

B. JWT Subsystem and Verifier Models

Tokens are issued correctly with PyJWT for HS256, RS256, and ES256. Two verifiers are then defined. The secure verifier follows best practice by pinning the expected algorithm and supplying the matching key. The naive verifier is a minimal re-implementation that reproduces the historical vulnerable pattern: it reads alg from the token header and dispatches accordingly, using a single key variable for every algorithm.

This design choice is methodologically important. Modern PyJWT already mitigates these attacks by requiring an explicit algorithms list and by refusing PEM-shaped keys as HMAC secrets. The naive verifier therefore deliberately omits those guards in order to faithfully recreate the attack surface that caused real-world incidents. The comparison remains fair because, on the PASETO side, the equivalent vulnerable pattern cannot be written at all: there is no algorithm field to read.

The essence of the naive verifier is the dispatch on token-supplied alg with a single shared key, sketched below:

```
alg = header["alg"]              #
attacker-controlled
if alg == "none":  return claims # no signature
if alg == "HS256": check_hmac(key, ...)
if alg == "RS256": check_rsa(key, ...)
# same `key` reused across algorithm families
```

By contrast, the PASETO verifier has no such dispatch: it asserts the prefix is v4.public and performs a single Ed25519 verification with the configured public key.

C. Attack Implementation

Four operations are implemented. The alg:none forge constructs a token with alg set to none and an empty signature. The RS256-to-HS256 forge computes an HMAC-SHA-256 over the forged signing input using the server's RSA public key (PEM) as the secret. The payload tamper modifies claims of a legitimate token while keeping its original signature. The prefix downgrade rewrites a PASETO token's version-purpose prefix. In every attack the adversary's goal is to obtain a token carrying the claims sub=admin, role=admin without knowledge of any private key or secret.

D. Experimental Design

1) Attack-success matrix: each attack is executed against the JWT naive verifier, the JWT secure verifier,

and the PASETO v4.public verifier. Each cell records FORGED (the forged token is accepted) or REJECTED (it is refused).

2) Performance benchmark: signing and verification are each timed over twenty thousand iterations with time.perf_counter() after a warm-up, for HS256, RS256, ES256, and PASETO v4.public, reported in microseconds per operation.

3) Token-size comparison: for an identical claim set, the encoded token length and the overhead relative to the raw payload are measured for every scheme.

4) Ecosystem survey: the same forged tokens are submitted to real, widely used JWT libraries rather than a hand-written verifier, across two languages and several versions, under both a permissive and an algorithm-pinned calling convention. Each cell records FORGED (accepted), REJECTED (verification fails), or BLOCKED (the library refuses the dangerous operation outright, for example by rejecting an asymmetric key used as an HMAC secret). The libraries exercised are PyJWT 1.7.1 and 2.8.0, python-jose 3.5.0, and Authlib 1.7.2 in Python, and jsonwebtoken 9.0.3 in Node.js.

IV. Results and Analysis

A. Attack Resilience

Table I and Fig. 1 summarize the attack-success matrix. The naive JWT verifier is forged by both alg:none and the RS256-to-HS256 confusion: in each case a token with administrative claims is accepted without any valid secret or private key. The secure JWT verifier rejects both, confirming that algorithm pinning is an effective but configuration-dependent mitigation. PASETO v4.public rejects every attack unconditionally.

TABLE I. Attack-Success Matrix

Attack	JWT naive	JWT secure	PASETO v4
alg:none	FORGED	REJECTED	REJECTED
RS256→HS256	FORGED	REJECTED	REJECTED
tamper payload	REJECTED	REJECTED	REJECTED
downgrade prefix	–	–	REJECTED

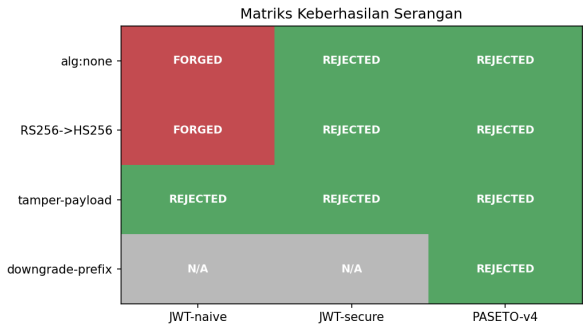

Fig. 1. Attack-success matrix. Red = forged (attack succeeds); green = rejected (secure).

The downgrade-prefix row is specific to PASETO and shows that rewriting the version-purpose prefix invalidates the token: the recomputed PAE no longer matches the signature. This illustrates why downgrade and confusion attacks are structurally absent rather than merely filtered.

It is worth emphasizing the qualitative difference between the two secure columns. The JWT secure verifier rejects the attacks because the developer remembered to pin the algorithm and isolate keys; remove that discipline, as the naive column shows, and the door reopens. PASETO rejects them for a different reason entirely: there is no algorithm to pin, no second algorithm to confuse, and no way for the token to request a different verification path. Security that depends on correct configuration is fragile across large teams and long-lived systems, whereas security that follows from the absence of a feature cannot be misconfigured into existence.

B. Performance

Table II and Fig. 2 report average signing and verification times. As expected, symmetric HS256 is fastest overall. RS256 has an expensive signing step (about one millisecond) but fast verification, reflecting RSA's asymmetric cost profile. ES256 and PASETO v4.public (Ed25519) are both moderate, completing signing and verification well under a millisecond; on the test machine PASETO was somewhat slower than ECDSA for both operations but remained in the same order of magnitude. The cost is dominated by the asymmetric primitive, not by the token format itself, and for typical request rates it is negligible, so performance is not a deciding factor between the schemes.

TABLE II. Sign / Verify Time (µs, N=20000)

Scheme	Sign	Verify
JWT HS256	6.45	7.77
JWT RS256	1018.29	36.71
JWT ES256	35.32	68.41
PASETO v4.public	85.76	196.05

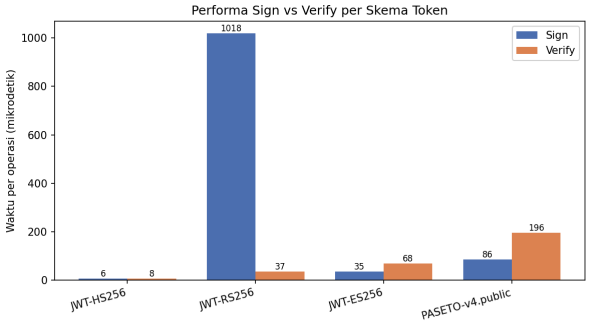

Fig. 2. Signing vs. verification time per scheme (lower is faster).

These figures are machine-dependent and should be read as relative rather than absolute. The practical conclusion is that the choice between JWT and PASETO is governed by security and robustness, not by raw speed.

To put the numbers in context, even the slowest single operation measured here (RS256 signing at roughly one millisecond) corresponds to on the order of a thousand operations per second per core, while verification across all asymmetric schemes stays under about two hundred microseconds. Token issuance happens once per login and verification once per request, so for typical web workloads the cryptographic cost is dwarfed by network and database latency. None of the schemes is performance-limited in practice, which reinforces that the decision should rest on the security properties summarized later in Table V.

C. Token Size

Table III and Fig. 3 compare token sizes for an identical 68-byte claim payload. RS256 is by far the largest because the RSA-2048 signature alone is 256 bytes. PASETO v4.public is compact: its Ed25519 signature is only 64 bytes, and it carries no JSON header. Although JWT HS256 is the smallest, it is symmetric and unsuitable where the verifier must not also be able to mint tokens; among asymmetric options, PASETO is the most space-efficient.

TABLE III. Token Size (bytes), payload = 68 B

Scheme	Token	Overhead
JWT HS256	163	95
JWT RS256	462	394
JWT ES256	206	138
PASETO v4.public	186	118

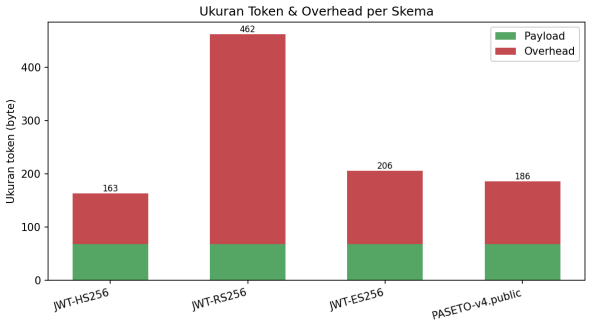

Fig. 3. Token size and overhead (payload vs. structural overhead).

D. Ecosystem Survey

Experiment 1 used a hand-written verifier to expose the attack surface in isolation. Experiment 4 instead probes how real libraries behave today. Table VI and Fig. 4 report the outcomes. The central result is unambiguous: no forge succeeded against any tested library, version, or configuration. Every attempt was either rejected through a failed verification or, more strongly, blocked outright.

TABLE VI. JWT Library Attack Survey

Library / Version	alg:none	conf. (perm.)	conf. (pin)
PyJWT 1.7.1	REJECTED	BLOCKED	REJECTED
PyJWT 2.8.0	REJECTED	BLOCKED	REJECTED
python-jose 3.5.0	REJECTED	BLOCKED	REJECTED
Authlib 1.7.2 *	BLOCKED	REJECTED	–
jsonwebtoken 9.0.3	REJECTED	BLOCKED	REJECTED

** Authlib was tested under its default decode API, which determines key and algorithm handling automatically rather than via a caller-supplied list.*

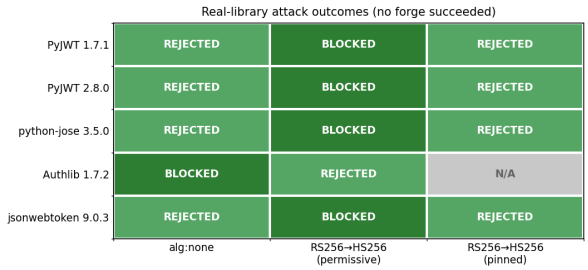

Fig. 4. Real-library attack outcomes. No forge succeeded; green = rejected, dark green = blocked proactively.

Two observations follow. First, the defensive mechanism is library-specific: most implementations proactively block the RS256-to-HS256 confusion by refusing to use an asymmetric key as an HMAC secret, while alg:none is variously blocked (Authlib) or rejected through signature failure (PyJWT, jsonwebtoken). Second, and notably, even PyJWT 1.7.1, a release from 2018, already carries these guards. The genuinely exploitable window therefore predates the versions in common use today; the disclosure-driven mitigations have long since been absorbed into the mainstream ecosystem.

This is a positive result for JWT, but it must be read carefully. The safety observed here is earned, not inherent. It rests on a decade of accumulated, per-library defensive code and on developers continuing to pin algorithms and isolate keys. Remove any one of those conditions, as the naive verifier in Experiment 1 does, and the forge returns. PASETO reaches the identical outcome without depending on any of them: with no algorithm field to confuse and no key-type ambiguity, the column for PASETO would be uniformly secure by construction rather than by mitigation. The contrast is the paper's central point, made sharper rather than weaker by the maturity of today's JWT libraries.

E. Usability–Security and Deployment Trade-offs

The results expose a clear trade-off in JWT. Its flexibility, an open algorithm field and a broad library ecosystem, is precisely what creates the confusion attack surface; safety depends on every verifier being configured correctly, every time. PASETO trades that flexibility for a fixed, opinionated contract that is secure by default but interoperates only where both sides speak the same version. PASETO is also outside the OAuth/OIDC standard stack, so it best fits systems whose token issuance and verification are fully controlled by one party.

For new deployments outside FIPS-constrained environments, PASETO v4.public is the recommended choice; v3 exists for NIST/FIPS settings. Where JWT must be used, the mitigations confirmed here are essential: always pin the expected algorithm explicitly, never accept alg:none, and never reuse a verification key across algorithm families.

Table V condenses the design-level differences that drive these outcomes.

TABLE V. Design Properties: JWT vs. PASETO

Property	JWT	PASETO v4
Algorithm field	alg in header	none
Suite selection	negotiable	fixed by version
alg:none risk	possible	absent
Confusion risk	possible	absent
Safe by default	no	yes
Ecosystem	very broad	limited

F. Limitations

Several limitations should be noted. First, the naive JWT verifier is intentionally constructed to expose the historical attack surface; current default configurations of maintained libraries are not vulnerable to these exact forges, as the ecosystem survey confirms, so the results characterize a class of misconfiguration rather than an unavoidable flaw of every JWT deployment. Second, the survey covers a sample of five library-version

combinations and does not include genuinely ancient releases (such as pre-1.5 PyJWT) or every language ecosystem; the Node.js jose library was not installed in the test run and is left for future coverage. Including a truly pre-mitigation release would likely yield the FORGED cells that current versions avoid, further sharpening the contrast. Third, the PASETO v4.public implementation, while validated by round-trip, tamper, and cross-key tests, was not cross-checked against the official reference test vectors in this study. Finally, the performance numbers are single-machine measurements and serve only for relative comparison, and the evaluation covers the public (signature) purpose where algorithm confusion lives, not the local (encrypted) purpose.

V. Conclusion

This paper implemented PASETO v4.public from its specification, built a comparable JWT subsystem, and experimentally evaluated both against algorithm confusion attacks. The naive JWT verifier was forged by alg:none and by the RS256-to-HS256 confused-deputy, while PASETO rejected every attack. A survey of real libraries across Python and Node.js showed that current implementations already neutralize these forges, but through accumulated, version- and configuration-dependent mitigations. The decisive difference is structural: PASETO has no negotiable algorithm field and binds its cryptographic suite to a non-negotiable version-purpose prefix, so the vulnerable pattern cannot be expressed rather than merely being filtered out, and its safety does not depend on the discipline of every verifier.

Performance and size measurements showed that all schemes are cryptographically inexpensive and that PASETO v4.public is the most compact asymmetric option, confirming that the choice between the two should be driven by security and robustness. Future work includes implementing and evaluating PASETO v4.local for encrypted tokens, cross-validating the from-scratch implementation against the reference pyseto library, and comparing PASETO with emerging standards such as FIDO2/WebAuthn for end-to-end authentication.

From the findings, three practical recommendations follow for developers and system administrators:

1) Prefer PASETO for self-contained systems: where both token issuance and verification are under one party's control and OAuth/OIDC interoperability is not required, PASETO v4.public removes an entire class of misconfiguration risk at no meaningful performance cost.

2) Harden JWT when it is mandatory: always pass an explicit allow-list of algorithms to the verifier, reject alg:none outright, and never reuse a single key variable across symmetric and asymmetric algorithms. These steps neutralize the forges demonstrated here.

3) Treat configuration as a security boundary: because JWT safety depends on correct verifier configuration, that configuration should be tested and reviewed like any other security control, whereas PASETO shifts the same guarantee from configuration to protocol design.

Source Code Repository (GitHub)
https://github.com/sendialicia/paseto-jwt-experiment

Video Link (YouTube)
https://www.youtube.com/watch?v=wExbt6KGr4E

Acknowledgment

The author thanks Dr. Ir. Rinaldi Munir, M.T. for his guidance throughout the II4021 Cryptography course, and the course community for their support.

References

[1] M. Jones, J. Bradley, and N. Sakimura, "JSON Web Token (JWT)," RFC 7519, IETF, May 2015.

[2] M. Jones, "JSON Web Algorithms (JWA)," RFC 7518, IETF, May 2015.

[3] T. McLean, "Critical vulnerabilities in JSON Web Token libraries," Auth0, 2015. [Online]. Available: https://auth0.com/blog/critical-vulnerabilities-in-json-web-token-libraries/

[4] PortSwigger, "JWT attacks," Web Security Academy. [Online]. Available: https://portswigger.net/web-security/jwt

[5] S. Arciszewski et al., "PASETO: Platform-Agnostic Security Tokens," IETF Internet-Draft draft-paragon-paseto-rfc, 2022.

[6] paseto-standard, "PASETO Specification," GitHub. [Online]. Available: https://github.com/paseto-standard/paseto-spec

[7] D. J. Bernstein, N. Duif, T. Lange, P. Schwabe, and B.-Y. Yang, "High-speed high-security signatures," J. Cryptographic Engineering, vol. 2, pp. 77–89, 2012.

[8] S. Josefsson and I. Liusvaara, "Edwards-Curve Digital Signature Algorithm (EdDSA)," RFC 8032, IETF, Jan. 2017.

[9] OWASP, "JSON Web Token Cheat Sheet," OWASP Cheat Sheet Series. [Online]. Available: https://cheatsheetseries.owasp.org/

[10] K. Dajiaji, "PySETO: A Python implementation of PASETO/PASERK," GitHub. [Online]. Available: https://github.com/dajiaji/pyseto

[11] José Padilla, "PyJWT documentation," Read the Docs. [Online]. Available: https://pyjwt.readthedocs.io/

[12] M. Panchenko et al., "python-jose: JOSE implementation in Python," GitHub. [Online]. Available: https://github.com/mpdavis/python-jose

[13] Auth0, "node-jsonwebtoken," GitHub. [Online]. Available: https://github.com/auth0/node-jsonwebtoken

[14] Authlib, "Authlib: JavaScript Object Signing and Encryption," Authlib documentation. [Online]. Available: https://docs.authlib.org/

Statement

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 19 Juni 2026

Sendi Putra Alicia 18223063