Design and Experimental Evaluation of a Crypto-Agile Code Signing Framework for Post-Quantum Migration

Zheannetta Apple Haihando – 18223105
Program Studi Sistem dan Teknologi Informasi
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
zheannettaapha@gmail.com , 18223105@std.stei.itb.ac.id

***Abstract*—Ketergantungan sistem distribusi perangkat lunak modern terhadap algoritma kriptografi asimetris tertentu menimbulkan risiko keamanan yang kritis seiring dengan ancaman perkembangan komputer kuantum. Setiap upaya migrasi algoritma baru, seperti transisi menuju standar pasca-kuantum ML-DSA (FIPS 204), umumnya terhambat oleh rancangan sistem code signing eksisting yang kaku (tight coupling). Makalah ini menyajikan rancangan, implementasi, dan evaluasi eksperimental sebuah framework code signing yang bersifat *crypto-agile*, dirancang untuk memungkinkan migrasi algoritma tanda tangan digital tanpa mengubah logika aplikasi inti. *Framework* dibangun di atas lapisan abstraksi kriptografi (*Crypto Provider Layer*) dengan antarmuka `SignatureProvider` yang memisahkan operasi *signing* dan *verification* dari implementasi algoritma spesifik. *Proof-of-concept* dikembangkan menggunakan Spring Boot, Java Cryptography Architecture (JCA), dan Bouncy Castle dengan tiga *provider*: ECDSA Provider, Hybrid Provider, dan Simulated Provider. Hasil eksperimen menunjukkan bahwa pendekatan *crypto-agile* secara signifikan mereduksi kompleksitas dan risiko migrasi algoritma, di mana integrasi algoritma baru dapat dilakukan tanpa modifikasi pada lapisan layanan utama (*modified services* = 0), sementara *overhead* performa yang dihasilkan masih berada dalam batas operasional yang dapat diterima.**

***Keywords*—crypto agility; code signing; post-quantum cryptography; ECDSA; ML-DSA; digital signature; software supply chain; Spring Boot; JCA; Bouncy Castle**

I. Pendahuluan

A. Latar Belakang

Kepercayaan terhadap perangkat lunak yang didistribusikan secara digital bergantung pada satu mekanisme kriptografi fundamental: *code signing*. Melalui mekanisme ini, penerbit perangkat lunak menandatangani artefak distribusi menggunakan kunci privat, sementara pihak penerima memverifikasi tanda tangan tersebut untuk memastikan autentisitas dan integritas perangkat lunak sebelum dieksekusi. Pada banyak platform modern, mekanisme ini bersifat wajib—sistem operasi menolak atau memberikan peringatan keras terhadap perangkat lunak yang tidak memiliki tanda tangan valid, menjadikan *code signing* sebagai lapisan kepercayaan yang tidak dapat diabaikan dalam rantai distribusi perangkat lunak.

Implementasi *code signing* saat ini umumnya mengandalkan algoritma kriptografi kunci publik seperti ECDSA dan RSA, yang keamanannya bertumpu pada kesulitan komputasional masalah matematika tertentu. Kemajuan dalam komputasi kuantum mengancam fondasi tersebut secara serius—telah dibuktikan secara teoritis bahwa komputer kuantum berskala kriptografis mampu menyelesaikan masalah matematika yang mendasari kedua algoritma tersebut jauh lebih efisien dibandingkan komputasi klasik [8], sehingga algoritma yang saat ini digunakan tidak lagi dapat dianggap aman dalam jangka panjang.

B. Pernyataan Masalah

Merespons ancaman ini, NIST pada tahun 2024 secara resmi menerbitkan standar kriptografi pasca-kuantum pertama, termasuk ML-DSA sebagai standar utama tanda tangan digital pasca-kuantum [2]. Namun, migrasi menuju standar baru jauh lebih kompleks daripada sekadar mengganti algoritma. Sebagian besar sistem *code signing* yang ada dirancang dengan *tight coupling* antara logika aplikasi dan implementasi algoritma—operasi kriptografi dipanggil langsung tanpa lapisan abstraksi, sehingga setiap upaya pergantian algoritma berpotensi menyebar ke seluruh lapisan sistem dan menimbulkan risiko *breaking changes*, peningkatan *maintenance burden*, serta kompleksitas migrasi yang tinggi. Tantangan ini tidak dapat diselesaikan hanya dengan mengadopsi algoritma baru—diperlukan pendekatan arsitektural yang secara eksplisit mendukung perubahan tersebut secara berkelanjutan.

Konsep yang menjawab permasalahan ini adalah *crypto agility*: kemampuan sistem untuk beradaptasi terhadap perubahan algoritma kriptografi tanpa rekayasa ulang yang signifikan [4]. Meskipun demikian, sebagian besar penelitian terkait transisi PQC masih berfokus pada perbandingan performa algoritma atau *benchmarking* implementasi PQC tertentu [1], [10]—penelitian yang mengeksplorasi *crypto agility* sebagai solusi arsitektural untuk sistem *code signing*, termasuk cara mengukur efektivitasnya dalam mengurangi kompleksitas migrasi, masih relatif terbatas. Kesenjangan inilah yang menjadi motivasi utama penelitian ini.

C. Tujuan dan Kontribusi

Penelitian ini menyajikan rancangan, implementasi, dan evaluasi eksperimental sebuah *framework code signing* yang bersifat *crypto-agile*. *Framework* dibangun di atas *Crypto Provider Layer* dengan antarmuka `SignatureProvider` yang seragam, sehingga seluruh operasi *signing* dan *verification* berjalan tanpa ketergantungan langsung pada algoritma tertentu. *Proof-of-concept* dikembangkan menggunakan Spring Boot, JCA, dan Bouncy Castle dengan tiga *provider*: *ECDSA Provider* sebagai *baseline*, *Hybrid Provider* sebagai representasi transisi *hybrid*, dan *Simulated Provider* sebagai pembuktian integrasi penyedia baru.

Kontribusi utama penelitian ini mencakup:

1) Rancangan arsitektur *crypto-agile* berbasis *provider abstraction* dengan *Crypto Provider Layer* sebagai pemisah antara logika aplikasi dan implementasi kriptografi.
2) Mekanisme *algorithm switching* berbasis konfigurasi yang tidak memerlukan modifikasi pada *business logic* aplikasi.
3) Evaluasi kuantitatif performa sistem dan *migration effort* melalui tiga skenario migrasi terstruktur.

D. Ruang Lingkup dan Batasan

Framework yang dikembangkan tidak mengimplementasikan algoritma PQC aktual seperti ML-DSA, Falcon, atau SLH-DSA secara penuh. Fokus utama penelitian adalah pada fleksibilitas arsitektur dan kemudahan migrasi, bukan pada keamanan kriptografi algoritma PQC itu sendiri. *Simulated Provider* digunakan sebagai representasi integrasi penyedia baru untuk menguji fleksibilitas arsitektur *framework* secara internal, bukan untuk menyimulasikan performa atau ukuran data PQC secara riil.

II. Landasan Teori

A. Tanda Tangan Digital dan Code Signing

Tanda tangan digital merupakan mekanisme kriptografi yang memungkinkan pihak pengirim membuktikan keotentikan sebuah pesan atau dokumen secara matematis. Berbeda dengan tanda tangan fisik, tanda tangan digital tidak hanya membuktikan siapa yang menandatangani, tetapi juga menjamin bahwa konten yang ditandatangani tidak berubah sejak saat penandatanganan dilakukan. Tiga properti utama yang dijamin adalah *authenticity* (identitas penandatangan dapat diverifikasi), *integrity* (konten tidak dimodifikasi), dan *non-repudiation* (penandatangan tidak dapat menyangkal) [9].

Dalam konteks distribusi perangkat lunak, properti-properti ini diimplementasikan melalui mekanisme *code signing*. Secara umum, alur kerjanya melibatkan dua sisi: sisi penerbit (*publisher*) yang menandatangani artefak menggunakan kunci privat, dan sisi penerima (*verifier*) yang memvalidasi tanda tangan menggunakan kunci publik yang bersesuaian. Pada banyak platform modern, mekanisme ini telah menjadi prasyarat wajib—sistem operasi menolak atau memberikan peringatan keras terhadap perangkat lunak tanpa tanda tangan valid, menjadikan *code signing* sebagai lapisan kepercayaan fundamental dalam rantai distribusi [6].

Relevansi *code signing* semakin meningkat seiring dengan berkembangnya ancaman pada *software supply chain*. Serangan yang menargetkan tahap distribusi—bukan hanya kode sumber—menunjukkan bahwa kompromi pada artefak yang didistribusikan dapat berdampak sangat luas. Dalam konteks ini, *code signing* berfungsi sebagai mekanisme deteksi: artefak yang telah dimodifikasi oleh pihak tidak berwenang akan gagal proses verifikasi, memutus rantai kepercayaan sebelum perangkat lunak berbahaya berhasil dieksekusi [6].

B. ECDSA sebagai Algoritma Baseline

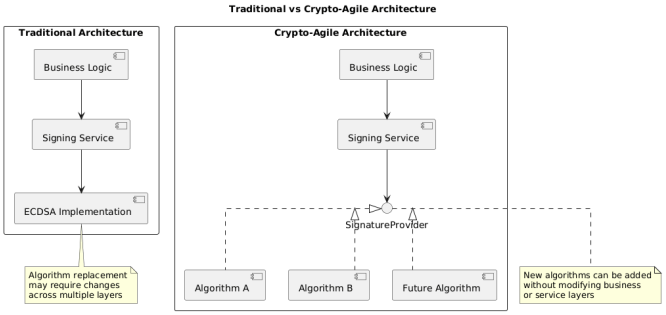
Elliptic Curve Digital Signature Algorithm (ECDSA) merupakan algoritma tanda tangan digital berbasis kriptografi kurva eliptis yang menjadi standar dominan pada sistem *code signing* modern. Dibandingkan pendahulunya RSA, ECDSA menawarkan keunggulan praktis yang signifikan: ukuran kunci yang jauh lebih kecil pada tingkat keamanan yang setara—kunci 256-bit pada kurva eliptis secara umum dianggap setara keamanannya dengan kunci RSA 3072-bit. Efisiensi ini menjadikan ECDSA pilihan natural untuk sistem *code signing* di mana ukuran artefak distribusi dan *overhead* verifikasi menjadi pertimbangan penting.

ECDSA telah ditetapkan dalam FIPS 186-5 dan didukung secara *native* oleh hampir seluruh infrastruktur kriptografi modern, termasuk *Java Cryptography Architecture* (JCA) dan Bouncy Castle. Ketersediaan implementasi yang luas dan rekam jejak yang panjang menjadikan ECDSA titik referensi yang tepat untuk mengukur performa dan kompleksitas migrasi dalam penelitian ini. Pada kurva *secp256r1*, ECDSA menghasilkan tanda tangan berukuran sekitar 70 byte—jauh lebih kompak dibandingkan RSA pada tingkat keamanan setara—namun keunggulan ini, bersama dengan seluruh keluarga algoritma berbasis masalah matematika klasik, berada di bawah ancaman serius dari kemajuan komputasi kuantum.

C. Ancaman Komputasi Kuantum dan Urgensi Migrasi

Keamanan ECDSA bertumpu pada kesulitan komputasional memecahkan masalah logaritma diskrit pada kurva eliptis—sebuah masalah yang secara praktis tidak dapat diselesaikan oleh komputer klasik dalam waktu yang masuk akal. Namun, telah dibuktikan secara teoritis bahwa komputer kuantum berskala kriptografis mampu menyelesaikan masalah ini secara efisien menggunakan algoritma Shor [8], sehingga seluruh algoritma kriptografi kunci publik berbasis masalah matematika klasik—termasuk RSA, DSA, dan ECDSA—tidak lagi dapat dianggap aman dalam jangka panjang.

Ancaman ini bukan sekadar wacana akademik. Komunitas keamanan mengenal skenario *harvest now, decrypt later*: pihak adversarial dapat mengumpulkan artefak bertanda tangan hari ini untuk dipalsukan di masa depan ketika komputer kuantum yang cukup kuat tersedia. Dalam konteks *code signing*, risiko



Gambar 1. Perbandingan arsitektur *code signing* tradisional dan arsitektur *crypto-agile* berbasis *provider abstraction*.

konkretnya adalah kemampuan penyerang untuk memalsukan tanda tangan artefak distribusi seolah berasal dari penerbit yang sah. Merespons urgensi ini, NIST pada 2024 secara resmi menerbitkan standar kriptografi pasca-kuantum pertama, termasuk FIPS 204 yang mendefinisikan ML-DSA berbasis masalah *module learning with errors* sebagai standar tanda tangan digital pasca-kuantum [3]. Diterbitkannya standar ini menegaskan bahwa transisi kriptografi bukan lagi pertanyaan "apakah", melainkan "bagaimana" dan "kapan" [2].

Namun justru di sinilah tantangan yang sesungguhnya muncul: sebagian besar sistem *code signing* yang ada saat ini belum dirancang dengan kemampuan untuk beradaptasi terhadap perubahan algoritma. Mengganti algoritma pada sistem seperti ini bukan tindakan lokal—ia berpotensi menyebar ke seluruh lapisan sistem dan menimbulkan *breaking changes* yang signifikan. Inilah yang membuat transisi menuju PQC menjadi masalah arsitektural, bukan sekadar masalah algoritmik.

D. Crypto Agility*: Definisi dan Prinsip Arsitektur*

Crypto agility merujuk pada kemampuan suatu sistem untuk beradaptasi terhadap perubahan algoritma atau parameter kriptografi tanpa memerlukan rekayasa ulang yang signifikan pada logika aplikasi [7], seperti yang diilustrasikan pada Gambar 1. Prinsip intinya adalah pemisahan yang tegas antara logika bisnis dan logika kriptografi: operasi kriptografi diabstraksi di balik antarmuka yang stabil, sehingga algoritma yang mendasarinya dapat diganti tanpa menyentuh kode di lapisan lain.

Pada sistem yang tidak *crypto-agile*, operasi kriptografi biasanya dipanggil secara langsung dari lapisan layanan, menciptakan *tight coupling* yang menjadikan setiap perubahan algoritma sebagai perubahan arsitektural berskala besar. Sebaliknya, sistem yang *crypto-agile* mengekspresikan kebijakan kriptografi—algoritma apa yang digunakan, parameter apa yang dipilih, kapan transisi dilakukan—pada lapisan konfigurasi, bukan dikodekan secara permanen di dalam implementasi [4]. Ini berarti migrasi algoritma cukup dilakukan dengan mendaftarkan provider baru dan mengubah satu parameter konfigurasi, tanpa menyentuh logika *signing* atau *verification* di lapisan atas.

Dari perspektif rekayasa perangkat lunak, *crypto agility* merupakan penerapan konkret dari prinsip *open/closed*—sistem terbuka terhadap perluasan (algoritma baru dapat ditambahkan), namun tertutup terhadap modifikasi (penambahan algoritma tidak mengubah komponen yang sudah ada). Kemampuan ini menjadi sangat kritis dalam konteks transisi pasca-kuantum, di mana standar terus berkembang dan organisasi mungkin perlu menjalankan beberapa algoritma secara paralel selama periode transisi [1].

E. Post-Quantum Migration Challenges

Meskipun standar PQC seperti ML-DSA telah tersedia, perjalanan menuju adopsi penuh masih dihadapkan pada serangkaian tantangan yang bersifat arsitektural, bukan semata-mata teknis. Penelitian terkini mengidentifikasi beberapa dimensi tantangan yang perlu dipahami sebelum merancang sistem yang benar-benar siap bermigrasi [10].

Kompatibilitas dan interoperabilitas. Algoritma PQC umumnya menghasilkan kunci dan tanda tangan yang jauh lebih besar dibandingkan pendahulunya—ukuran tanda tangan ML-DSA, misalnya, berkisar antara 2420 hingga 4595 byte tergantung level keamanan, dibandingkan $\sim$70 byte untuk ECDSA. Perbedaan ini berimplikasi langsung pada format paket distribusi, protokol transmisi, dan sistem verifikasi yang ada, yang mungkin belum dirancang untuk mengakomodasi ukuran tersebut.

Periode transisi *hybrid*. Selama masa peralihan, sistem mungkin perlu mendukung algoritma lama dan baru secara bersamaan—melayani klien yang belum diperbarui sekaligus memastikan keamanan terhadap ancaman kuantum bagi klien yang sudah diperbarui. Skema tanda tangan *hybrid*, di mana artefak ditandatangani dengan dua algoritma sekaligus, menjadi salah satu strategi yang direkomendasikan untuk mengelola periode ini [5].

***Maintainability* dan evolusi sistem.** Sistem yang tidak memiliki lapisan abstraksi kriptografi akan menghadapi *migration effort* yang tinggi setiap kali standar berubah atau kerentanan baru ditemukan. Tanpa mekanisme isolasi yang jelas, setiap perubahan algoritma berpotensi menimbulkan efek riak (*ripple effect*) ke seluruh basis kode yang bergantung pada implementasi spesifik tersebut.

Kesiapan arsitektur sebagai prasyarat. Tantangan-tangangan di atas mengarah pada satu kesimpulan: kesiapan untuk bermigrasi menuju PQC tidak dapat dinilai semata dari ketersediaan algoritma baru, melainkan dari sejauh mana arsitektur sistem yang ada mendukung perubahan tersebut dengan minimal gangguan. Inilah yang menjadikan pengukuran *migration effort*—bukan hanya performa algoritma—sebagai metrik yang relevan untuk mengevaluasi kesiapan sistem terhadap era pasca-kuantum [4], [7].

F. Provider Pattern *dalam Arsitektur Kriptografi*

Provider pattern merupakan salah satu pola arsitektur paling efektif untuk mewujudkan *crypto agility* dalam implementasi nyata. Secara struktural, pola ini menerapkan prinsip inversi dependensi: lapisan tingkat tinggi bergantung pada abstraksi

(antarmuka), bukan pada implementasi konkret. Dalam konteks kriptografi, sebuah antarmuka mendefinisikan kontrak operasi—*signing*, *verification*, *key generation*—sementara setiap algoritma diimplementasikan sebagai *provider* terpisah yang memenuhi kontrak tersebut. Pergantian algoritma cukup dilakukan dengan mendaftarkan *provider* baru; lapisan layanan tidak perlu diubah.

Java Cryptography Architecture (JCA) merupakan contoh referensi yang paling dikenal dari pola ini pada platform Java. JCA mendefinisikan antarmuka seperti `Signature` dan `KeyPairGenerator` yang memisahkan penggunaan kriptografi dari implementasinya—implementasi konkret disediakan melalui mekanisme *provider* yang dapat didaftarkan secara dinamis. Library Bouncy Castle, yang digunakan dalam penelitian ini, mengikuti model yang sama: ia mendaftarkan dirinya sebagai JCA *provider* tambahan, mengekspos algoritma tambahan (termasuk kandidat PQC eksperimental) melalui antarmuka yang identik dengan yang sudah ada [4].

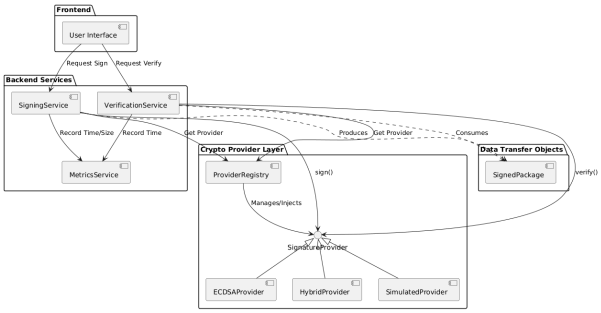
Dalam penelitian ini, prinsip *provider pattern* diadopsi dan diperluas melalui antarmuka `SignatureProvider` yang didefinisikan sendiri, di atas lapisan JCA dan Bouncy Castle. Ekstensi ini diperlukan karena JCA standar tidak menyediakan mekanisme untuk mengelola *multi-signature*—sebuah kebutuhan yang muncul pada skema *hybrid* di mana satu artefak ditandatangani oleh dua algoritma sekaligus. Dengan mendefinisikan kontrak abstraksi pada tingkat ini, *framework* yang dikembangkan dapat mengakomodasi baik algoritma tunggal maupun komposit tanpa mengubah logika layanan di atasnya.

III. Desain Sistem

A. Gambaran Umum Arsitektur Sistem

Arsitektur *framework* dirancang dengan prinsip pemisahan perhatian (*separation of concerns*) untuk mengurangi *coupling* antara logika aplikasi dan implementasi algoritma kriptografi. Tujuan utama arsitektur ini adalah memfasilitasi migrasi algoritma kriptografi dengan meminimalisasi modifikasi pada sistem inti. Tanggung jawab sistem didistribusikan ke dalam beberapa komponen utama, yaitu *Signing Service*, *Verification Service*, *Crypto Provider Layer*, dan *Provider Registry*. Komponen-komponen ini saling berinteraksi melalui antarmuka yang standar, sehingga mendukung prinsip *crypto agility*. Gambaran arsitektur ini divisualisasikan pada Gambar 2.

Secara konseptual, framework dibangun di atas *Provider Abstraction Pattern* yang menempatkan seluruh implementasi algoritma di belakang antarmuka `SignatureProvider`. Dengan pendekatan ini, lapisan layanan hanya berinteraksi dengan kontrak abstrak dan tidak memiliki ketergantungan langsung terhadap algoritma tertentu. Penambahan, penggantian, maupun penghapusan algoritma dapat dilakukan melalui implementasi provider baru tanpa memerlukan perubahan pada *SigningService* maupun *VerificationService*. Karakteristik ini menjadi fondasi utama *crypto agility* yang diusulkan dalam penelitian.



Gambar 2. Arsitektur *framework crypto-agile* yang diusulkan.

B. Crypto Provider Layer

Crypto Provider Layer bertujuan untuk memberikan abstraksi tunggal terhadap seluruh operasi kriptografi tingkat rendah. Tanggung jawab utama lapisan ini diemban oleh antarmuka `SignatureProvider`, yang memisahkan lapisan bisnis (*Signing Service* dan *Verification Service*) dari dependensi algoritma spesifik seperti JCA atau Bouncy Castle. Interaksi antarkomponen diatur menggunakan parameter dan nilai kembalian standar; tanda tangan direpresentasikan dalam format *map* berbasis JSON sehingga skema penandatanganan algoritma tunggal maupun komposit dapat diimplementasikan tanpa memerlukan perubahan struktur data pada antarmuka. Hubungan antara antarmuka `SignatureProvider` dan ketiga implementasi provider divisualisasikan pada Gambar 3.

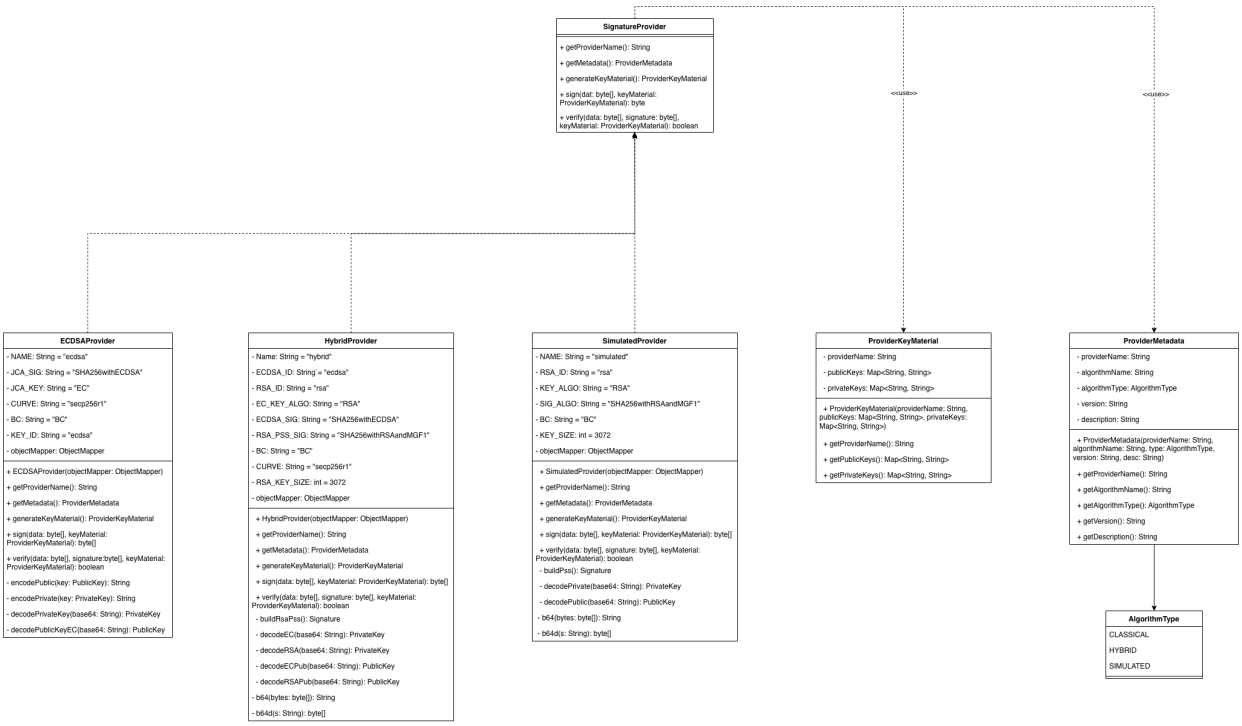
Secara implementatif, `SignatureProvider` mendefinisikan empat operasi standar yang harus dipenuhi oleh setiap implementasi: `generateKeyMaterial()`, `sign()`, `verify()`, dan `getMetadata()`. Kontrak yang seragam ini memungkinkan seluruh provider digunakan secara bergantian tanpa memerlukan perubahan pada lapisan layanan. Dengan demikian, integrasi algoritma baru cukup dilakukan melalui implementasi provider tambahan yang mengikuti kontrak yang sama—tanpa menyentuh *SigningService* maupun *VerificationService*.

C. *Implementasi* Provider

Tiga *provider* diimplementasikan untuk merepresentasikan tiga fase transisi migrasi yang berbeda, sekaligus untuk memvalidasi fleksibilitas rancangan *Provider Abstraction Pattern*. Setiap *provider* dianotasi dengan `@Component` sehingga Spring secara otomatis mendaftarkannya ke dalam *Provider Registry* saat aplikasi diinisialisasi.

Pertama, ***ECDSA Provider*** beroperasi menggunakan `SHA256withECDSA` pada kurva `secp256r1`. Komponen ini berfungsi sebagai *baseline* untuk mengukur performa dan *migration effort* sistem konvensional sebelum migrasi dilakukan.

Kedua, ***Hybrid Provider*** menjalankan algoritma ECDSA dan RSA-PSS-3072 secara paralel dalam satu siklus penandatanganan. Komponen ini dirancang sebagai representasi arsitektural fase transisi—bukan implementasi skema *hybrid signature* yang mengikuti standar kriptografis tertentu. Hasil penandatanganan direpresentasikan sebagai dua *signature* independen



Gambar 3. Abstraksi antarmuka `SignatureProvider` terhadap ketiga implementasi *provider*.

dalam satu struktur data, dan verifikasi menggunakan semantik logika **AND**: artefak hanya dianggap valid apabila **kedua** *signature* berhasil diverifikasi. Pendekatan ini menyimulasikan kebutuhan keamanan ganda selama masa peralihan teknologi.

Ketiga, ***Simulated Provider*** mengimplementasikan RSA-PSS-3072 sebagai representasi algoritma yang *unfamiliar*. Komponen ini tidak dimaksudkan sebagai implementasi algoritma *Post-Quantum Cryptography* (PQC) aktual, melainkan semata-mata untuk mengevaluasi *migration effort* arsitektural ketika sebuah algoritma baru diintegrasikan ke dalam *framework*. Keberadaan ketiga *provider* ini mendemonstrasikan bahwa penambahan algoritma baru cukup dilakukan dengan mengimplementasikan satu kelas *provider* baru—tanpa perubahan apa pun pada lapisan layanan.

D. Signing Service

Signing Service memiliki tujuan untuk mengorkestrasi proses penandatanganan artefak distribusi. Tanggung jawab komponen ini mencakup kalkulasi nilai *hash* (SHA-256) dari berkas artefak, resolusi instansiasi *provider* melalui *Provider Registry*, dan pembuatan `SignedPackage`. Layanan ini berinteraksi dengan `SignatureProvider` dan *MetricsService* tanpa menyimpan dependensi terhadap implementasi algoritma spesifik. Desain yang tidak bergantung pada *provider* tertentu ini meminimalkan dampak modifikasi (*ripple effect*) ketika terjadi transisi algoritma kriptografi, sehingga mengukuhkan prinsip *crypto agility*.

E. Verification Service

Tujuan utama *Verification Service* adalah memvalidasi integritas artefak dan keabsahan tanda tangannya. Komponen ini bertanggung jawab untuk merekonstruksi objek material kunci publik dari `SignedPackage`, mengidentifikasi nama *provider* yang digunakan, dan mendelegasikan perintah verifikasi kepada antarmuka `SignatureProvider`. Layanan ini saling berinteraksi dengan *Provider Registry* untuk resolusi dependensi dan dengan *MetricsService* untuk pencatatan metrik waktu. Layaknya *Signing Service*, komponen ini tidak bergantung pada detail implementasi *provider*.

Portabilitas artefak dicapai melalui penyimpanan metadata *provider* dan material verifikasi di dalam `SignedPackage`. Dengan pendekatan ini, proses verifikasi dapat direproduksi pada lingkungan lain selama *provider* yang sama tersedia pada *ProviderRegistry*. Desain tersebut memastikan konsistensi proses verifikasi tanpa menuntut ketergantungan langsung terhadap implementasi algoritma tertentu.

F. Provider Registry *dan* Algorithm Switching

Provider Registry memusatkan mekanisme resolusi dan manajemen *provider* kriptografi pada saat *runtime*. Komponen ini memanfaatkan mekanisme *Dependency Injection* Spring untuk mendeteksi secara otomatis seluruh kelas yang dianotasi `@Component` dan mengimplementasikan antarmuka `SignatureProvider`, kemudian menyimpannya dalam sebuah *map* yang diindeks berdasarkan nama *provider*. Ketika *Signing Service* atau *Verification Service* membutuhkan *provider* tertentu, mereka menyertakan nama *provider* sebagai parameter;

Provider Registry melakukan *lookup* berdasarkan nama tersebut dan mengembalikan instansiasi yang sesuai. Penggantian algoritma (*algorithm switching*) dengan demikian cukup dilakukan melalui perubahan satu nilai konfigurasi tanpa menyentuh logika layanan mana pun.

Kombinasi antara `SignatureProvider`, *Provider Registry*, serta layanan yang tidak bergantung pada implementasi algoritma membentuk mekanisme utama *crypto agility* pada *framework* yang diusulkan. Algoritma baru dapat diintegrasikan hanya dengan menambahkan satu kelas *provider* baru—tanpa perubahan pada *SigningService*, *VerificationService*, maupun *Provider Registry* itu sendiri. Karakteristik inilah yang menjadi dasar pengukuran *migration effort* pada Bab V.

G. MetricsService

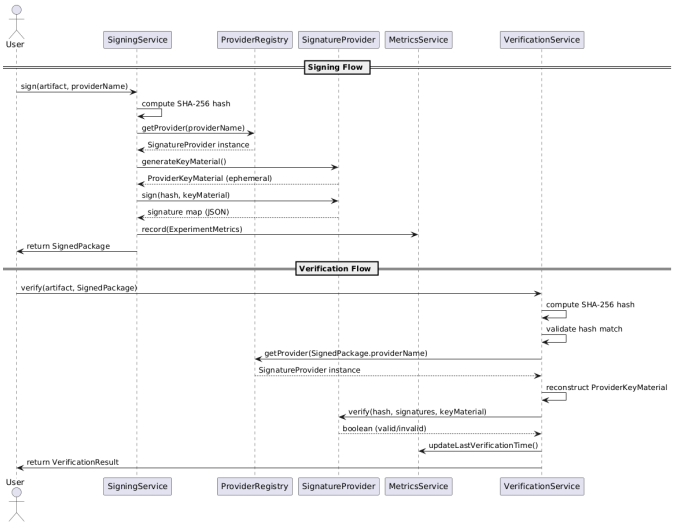
MetricsService dibangun secara khusus dengan tujuan memfasilitasi pengumpulan data observasi untuk evaluasi arsitektur. Tanggung jawab komponen ini difokuskan pada agregasi data kuantitatif yang terdiri atas: 1) durasi proses *signing* dan *verification* dalam satuan milidetik, 2) ukuran *public key* yang dihasilkan, dan 3) ukuran *signature* yang dihasilkan. *MetricsService* berinteraksi secara pasif dengan dipanggil oleh *Signing Service* dan *Verification Service* setiap kali satu siklus operasional selesai dikerjakan. Data yang direkam oleh komponen ini akan bertindak sebagai fondasi evaluasi eksperimental pada Bab V guna membuktikan efektivitas performa serta beban struktural yang harus ditanggung sistem di berbagai skenario migrasi.

Perlu dicatat bahwa *MetricsService* tidak terlibat dalam proses kriptografi inti. Komponen ini hanya berfungsi sebagai lapisan observasi yang mencatat parameter eksperimen secara non-intrusif. Dengan demikian, pengukuran performa dapat dilakukan tanpa memengaruhi perilaku operasional *provider* maupun layanan yang sedang dievaluasi.

H. Struktur Data Utama

Untuk mendukung pemisahan tanggung jawab antar komponen dan memastikan portabilitas artefak hasil penandatanganan, *framework* mendefinisikan empat struktur data khusus sebagai media pertukaran informasi antar layanan.

`SignedPackage` merupakan komponen sentral yang berfungsi sebagai artefak portabel hasil proses *signing*. Objek ini menyimpan *hash* artefak (SHA-256), nama *provider*, *public key*, serta *signature* yang diperlukan untuk verifikasi, sehingga proses verifikasi dapat direproduksi secara independen tanpa bergantung pada konteks *signing* sebelumnya. `ProviderKeyMaterial` mengenkapsulasi material kunci publik dan privat yang digunakan *provider* dalam satu siklus operasional. `ProviderMetadata` menyimpan identitas, tipe algoritma, dan deskripsi *provider*. `VerificationResult` membawa hasil validasi, durasi eksekusi, dan pesan status yang dikembalikan oleh *Verification Service*.



Gambar 4. Urutan proses *signing* dan *verification* pada *framework*.

I. Alur Operasional Sistem

Setelah komponen-komponen utama dijelaskan secara individual, langkah berikutnya adalah memahami bagaimana komponen tersebut berinteraksi selama proses *signing* dan *verification*. Interaksi tersebut divisualisasikan melalui diagram urutan pada Gambar 4.

Pada alur penandatanganan, sistem menerima artefak dan meneruskannya ke *Signing Service* untuk dihitung nilai *hash*-nya. Berdasarkan instruksi konfigurasi, *Signing Service* mengakses *Provider Registry* guna meresolusi *provider*. Selanjutnya, *provider* membangkitkan *ephemeral keys* dan meneken nilai *hash* tersebut. Seluruh beban operasional dicatat oleh *MetricsService*, sebelum *Signing Service* merilis `SignedPackage` sebagai produk akhir.

Pada alur verifikasi, artefak dan `SignedPackage` dievaluasi melalui komparasi nilai *hash* dasar. Apabila terverifikasi autentik, nama *provider* diekstraksi dari `SignedPackage` untuk menentukan spesifikasi verifikasi dari *Provider Registry*. Setelah kunci publik dirakit kembali, algoritma pembanding dari *provider* dijalankan. Proses verifikasi ini diakhiri dengan pendataan durasi oleh *MetricsService* dan pengembalian hasil uji validitas melalui objek `VerificationResult`. Rancangan sistem yang telah dijelaskan pada bab ini menjadi dasar implementasi *framework* yang dibahas pada Bab IV.

IV. Implementasi

A. Spesifikasi Teknologi (Tech Stack)

Sistem *Proof-of-Concept* (PoC) diimplementasikan menggunakan tumpukan teknologi modern yang tercantum pada Tabel I.

Backend aplikasi diorganisasikan menggunakan kerangka kerja Spring Boot yang mendelegasikan tugas pemrosesan ke lapisan pengontrol (*controllers*), layanan (*services*), registri (*registry*), dan penyedia (*providers*). Pemilihan Spring

Tabel I
SPESIFIKASI TEKNOLOGI IMPLEMENTASI POC

Komponen	Teknologi / Pustaka dan Keterangan
Platform Eksekusi	Java 23 (JDK 23), dengan kompatibilitas mundur hingga Java 17 (LTS) karena berbasis API standar JCA.
Framework Utama	Spring Boot 3.2.5 untuk penyediaan REST API, *dependency injection*, dan manajemen daur hidup komponen.
Build Tool	Maven 3.9 untuk manajemen dependensi dan otomasi proses kompilasi.
Kriptografi Inti	Java Cryptography Architecture (JCA) dikombinasikan dengan Bouncy Castle 1.78.1 (`bcprov-jdk18on`) sebagai *security provider*.
Klien Web (*Frontend*)	React 19 + Vite 8 menggunakan TypeScript, menyediakan antarmuka interaktif untuk modul unggah berkas, penandatanganan, verifikasi, dan visualisasi metrik.
Serialisasi Data	Jackson ObjectMapper untuk konversi data DTO dan pemetaan data biner Base64 ke format JSON.
Pengujian	JUnit 5 untuk verifikasi pengujian unit fungsional.

Boot sebagai *framework* utama didasarkan pada kebutuhan arsitektural untuk mengelola daur hidup komponen secara dinamis melalui wadah *Inversion of Control* (IoC) dan *Dependency Injection* (DI). Mekanisme *auto-discovery* pada Spring memungkinkan kelas penyedia konkret baru yang dianotasi dengan `@Component` dideteksi secara otomatis saat *booting* aplikasi tanpa memerlukan pendaftaran manual (*hardcoded wiring*) di *registry*, sehingga menjaga arsitektur tetap longgar (*loosely coupled*). Penggunaan Maven 3.9 memastikan seluruh dependensi seperti Bouncy Castle dan Jackson terintegrasi secara modular. Di sisi klien, aplikasi React 19 dibangun dengan Vite 8 untuk menyajikan antarmuka visual terpadu bagi pengguna.

B. Struktur Paket dan Pengorganisasian Kode

Untuk menjaga prinsip pemisahan perhatian (*separation of concerns*) dan mempermudah pemeliharaan, basis kode *backend* diorganisasikan ke dalam struktur paket modular sebagai berikut:

- `com.cryptoagile.codesigning.config`: Berisi konfigurasi sistem Spring, termasuk registrasi *security provider* Bouncy Castle secara global pada `CryptoConfig`.
- `com.cryptoagile.codesigning.controller`: Menyediakan *REST API endpoints* seperti `SigningController` (`POST /api/sign`), `VerificationController` (`POST /api/verify`), dan `MetricsController`.
- `com.cryptoagile.codesigning.crypto.provider`: Paket tempat antarmuka `SignatureProvider` didefinisikan, beserta kelas implementasi konkretnya (`ECDSAProvider`, `HybridProvider`, dan `SimulatedProvider`).
- `com.cryptoagile.codesigning.crypto.registry`: Mengandung `ProviderRegistry` yang bertanggung jawab melakukan penemuan otomatis (*auto-discovery*) dan resolusi dinamis terhadap penyedia kriptografi pada saat *runtime*.
- `com.cryptoagile.codesigning.service`: Lapisan bisnis yang independen dari detail algoritma kriptografi konkret, di antaranya `SigningService` dan `VerificationService` yang berinteraksi dengan antarmuka abstraksi kriptografi.
- `com.cryptoagile.codesigning.dto` dan `metrics`: Berisi kelas pemindahan data (*Data Transfer Objects*) seperti `SignedPackage` dan struktur penyimpanan data metrik eksperimen.

Pengorganisasian paket yang terisolasi ini memastikan bahwa penambahan algoritma baru sepenuhnya terlokalisasi di dalam paket `crypto.provider` tanpa memicu efek riak (*ripple effect*) modifikasi pada paket layanan (`service`) atau pengontrol (`controller`).

C. Implementasi Antarmuka SignatureProvider

Antarmuka `SignatureProvider` bertindak sebagai kontrak arsitektural utama yang menjamin terwujudnya *crypto agility*. Semua operasi kriptografi tingkat rendah diisolasi di balik antarmuka ini untuk mencegah *tight coupling* antara logika bisnis aplikasi dan implementasi spesifik algoritma. Definisi kontrak antarmuka diimplementasikan sebagai antarmuka Java standar, dengan metode-metode utama seperti yang ditunjukkan pada Listing 1.

```
1  public interface SignatureProvider {
2      String getProviderName();
3      ProviderMetadata getMetadata();
4      ProviderKeyMaterial generateKeyMaterial();
5      byte[] sign(byte[] data, ProviderKeyMaterial
           keyMaterial);
6      boolean verify(byte[] data, byte[] signature,
           ProviderKeyMaterial keyMaterial);
7  }
```

Listing 1. Definisi kontrak antarmuka `SignatureProvider`.

Metode `getProviderName()` mengembalikan pengenal unik dalam huruf kecil (seperti `"ecdsa"`). `getMetadata()` mengembalikan DTO `ProviderMetadata` yang menampung detail algoritma. Metode `generateKeyMaterial()` bertugas membangkitkan pasangan kunci efemeris untuk siklus tanda tangan tunggal. Metode `sign()` dan `verify()` menggunakan representasi biner dari peta JSON (`Map<String, String>`) sebagai format pertukaran tanda tangan untuk menampung skema penandatanganan tunggal maupun komposit secara fleksibel.

D. Implementasi Masing-masing Provider

1) ECDSA Provider*: ECDSA Provider* direalisasikan melalui kelas `ECDSAProvider` yang mengimplementasikan `SignatureProvider`. Komponen ini memanfaatkan API JCA

standar `java.security.Signature` dengan algoritma penandatanganan `"SHA256withECDSA"` dan generator pasangan kunci `KeyPairGenerator` untuk kurva standar NIST `secp256r1`. Bouncy Castle (`"BC"`) didaftarkan secara eksplisit sebagai penyedia keamanan dasar. *ECDSA Provider* menghasilkan tanda tangan tunggal berukuran sekitar 71 byte dan bertindak sebagai referensi pembanding (*baseline*) terhadap *overhead* performa dan ukuran tanda tangan pada skema transisi.

2) Hybrid Provider*: Hybrid Provider* diimplementasikan pada kelas `HybridProvider`. Komponen ini menyimulasikan strategi migrasi transisi dengan menjalankan dua algoritma penandatanganan secara independen terhadap nilai *hash* artefak yang sama. Algoritma pertama adalah ECDSA (`secp256r1`), sedangkan algoritma kedua menggunakan RSA-PSS dengan panjang kunci 3072-bit (`"SHA256withRSAandMGF1"` dengan parameter PSS terperinci).

Proses penandatanganan menghasilkan dua buah tanda tangan digital yang dimasukkan ke dalam JSON `Map<String, String>` dengan entri kunci `"ecdsa"` dan `"rsa"`. Proses verifikasi menerapkan semantik logika **AND**:

$$\text{Status Verifikasi} = \text{Validasi}_{\text{ECDSA}} \wedge \text{Validasi}_{\text{RSA-PSS}} \quad (1)$$

Artefak perangkat lunak hanya dinyatakan valid jika kedua tanda tangan digital tersebut berhasil diverifikasi secara independen. Jika salah satu atau kedua tanda tangan tidak valid, maka verifikasi gagal. Pendekatan hibrida ini meningkatkan tingkat kepercayaan selama masa transisi karena validitas artefak bergantung pada keberhasilan verifikasi kedua algoritma.

3) Simulated Provider*: Simulated Provider* direalisasikan dalam kelas `SimulatedProvider` menggunakan algoritma RSA-PSS-3072 (`"SHA256withRSAandMGF1"`). Penting untuk ditegaskan bahwa `SimulatedProvider` *bukan* merupakan implementasi algoritma kriptografi pasca-kuantum (PQC) riil seperti ML-DSA atau Falcon, dan tidak menyimulasikan secara presisi karakteristik ukuran data dari ML-DSA. Pemilihan algoritma klasik RSA-PSS dengan kunci 3072-bit ini semata-mata ditujukan untuk menghadirkan algoritma alternatif baru dengan ukuran kunci dan tanda tangan yang lebih besar dibandingkan ECDSA *baseline*, guna mengevaluasi *migration effort* dari sudut pandang perubahan arsitektural tanpa perlu menyentuh layanan inti sistem.

E. Provider Registry *dan Resolusi Algoritma*

Manajemen daur hidup penyedia kriptografi dipusatkan pada kelas `ProviderRegistry`. Registri ini memanfaatkan fitur pendeteksian otomatis (*auto-discovery*) dari Spring Framework. Seluruh kelas konkret yang mengimplementasikan antarmuka `SignatureProvider` dan ditandai dengan anotasi `@Component` secara otomatis disuntikkan oleh kontainer Spring ke dalam konstruktor `ProviderRegistry` sebagai `List<SignatureProvider>`.

Di dalam registri, list penyedia tersebut ditransformasikan menjadi peta yang tidak dapat dimodifikasi (`Map<String, SignatureProvider>`) dengan nama penyedia sebagai kunci pencarian. Mekanisme pergantian algoritma (*algorithm switching*) dicapai di *runtime* melalui metode `getProvider(String name)` dengan *lookup* berbasis string nama penyedia (case-insensitive). Karena modul layanan hanya meminta antarmuka abstrak dari *registry* ini, pergantian algoritma aktif dapat dilakukan secara instan tanpa memodifikasi kode kelas `SigningService` maupun `VerificationService`.

F. Implementasi Signing Service

Lapisan penandatanganan diorkestrasi oleh kelas `SigningService`. Layanan ini dirancang steril dari dependensi algoritma konkret. Alur eksekusi *run-time* penandatanganan dirinci sebagai berikut:

1) Sistem menerima berkas biner artefak melalui REST API *endpoint* `POST /api/sign`.
2) Nilai *hash* dari berkas artefak dikalkulasi menggunakan utilitas `HashUtil.sha256(fileBytes)` untuk menghasilkan nilai *hash* biner, serta representasi heksadesimalnya.
3) Instansiasi penyedia kriptografi diresolusi dari `ProviderRegistry` berdasarkan parameter `providerName`.
4) `SigningService` memanggil metode `generateKeyMaterial()` dari penyedia terpilih untuk menghasilkan pasangan kunci efemeris pembantu.
5) Operasi penandatanganan dijalankan dengan memanggil `provider.sign(digest, keyMaterial)`, dan waktu eksekusinya diukur menggunakan `System.currentTimeMillis()`.
6) Hasil penandatanganan dide-serialisasi dari bentuk biner JSON menjadi peta data `signatures`.
7) Data ukuran biner dari tanda tangan dan kunci publik dikalkulasi untuk pencatatan eksperimen.
8) Metrik waktu dan ukuran diteruskan ke kelas `MetricsService`.
9) Objek `SignedPackage` dirakit dan dikembalikan sebagai respons JSON.

G. Implementasi Verification Service

Proses verifikasi integritas dan keabsahan tanda tangan dikelola oleh kelas `VerificationService`. Layanan ini menerima berkas artefak dan objek JSON `SignedPackage`. Alur eksekusi verifikasi aktual berjalan sebagai berikut:

1) Nilai *hash* berkas artefak dihitung ulang dan dicocokkan dengan *field* `artifactHash` dari `SignedPackage`. Jika terdeteksi perbedaan, proses langsung dihentikan dengan pesan kegagalan modifikasi artefak (*hash mismatch*).
2) Nama penyedia diidentifikasi dari paket data, dan instansiasi penyedia diresolusi dari `ProviderRegistry`.
3) Objek `ProviderKeyMaterial` direkonstruksi menggunakan data kunci publik yang diekstraksi dari `SignedPackage`, sedangkan peta kunci privat secara eksplisit diisi dengan peta kosong (`Collections.emptyMap()`) untuk memastikan

bahwa operasi verifikasi berjalan secara aman tanpa ketergantungan pada kunci privat.
4) Peta tanda tangan digital dari paket data diserialisasi kembali ke format biner JSON untuk mematuhi kontrak antarmuka penyedia.
5) Operasi verifikasi dijalankan dengan memanggil `provider.verify(digest, sigBytes, keyMaterial)` dan durasi eksekusinya dicatat.
6) Nilai waktu verifikasi diperbarui pada modul `MetricsService`, dan status hasil akhir dibungkus dalam objek `VerificationResult` untuk dikirimkan kembali ke klien.

Sifat portabilitas paket tanda tangan dicapai karena `SignedPackage` mengemas seluruh material verifikasi (kunci publik dan tanda tangan) secara mandiri, sehingga proses verifikasi dapat direproduksi secara sukses pada *node* distribusi lain yang memiliki *registry* penyedia yang sama.

H. Struktur Data dan Data Transfer Object *(DTO)*

Operasi pertukaran data antar komponen *runtime backend* serta transmisi data antara *backend* dan *frontend* difasilitasi oleh empat kelas model representasi data terstruktur berikut:

- `SignedPackage`: Menjadi produk akhir dari alur penandatanganan. Menyimpan *field* `artifactHash` (SHA-256 heksadesimal), `providerName` (nama penyedia), `algorithmType` (tipe klasifikasi algoritma), `timestamp` (penanda waktu ISO-8601), `signingTimeMs` (durasi tanda tangan), `signatures` (peta data tanda tangan ter-encode Base64), dan `publicKeys` (peta kunci publik pembentuk format DER ter-encode Base64).
- `VerificationResult`: Menampung status validitas akhir verifikasi (`valid` bertipe `boolean`), `providerName`, `verificationTimeMs` (durasi verifikasi), dan pesan diagnostik `diagnosticMessage`.
- `ProviderKeyMaterial`: Struktur internal yang membawa material pasangan kunci efemeris yang terdiri dari *field* `providerName`, peta `publicKeys`, dan peta `privateKeys`.
- `ProviderMetadata`: Menyimpan informasi statis penyedia kriptografi seperti `providerName`, `algorithmName`, tipe `algorithmType` (berupa *enum* `CLASSICAL`, `HYBRID`, atau `SIMULATED`), versi penyedia (`version`), dan deskripsi fungsional (`description`).

Seluruh objek tersebut ditransformasikan dari representasi kelas memori Java ke representasi teks JSON menggunakan pustaka Jackson untuk dipertukarkan melalui *endpoint* API HTTP REST.

*I. Implementasi Pengumpulan Metrik (*MetricsService*)*

Untuk mendukung evaluasi kuantitatif arsitektur sistem yang menjadi fokus pembahasan pada Bab V, modul pengumpulan metrik diimplementasikan pada kelas `MetricsService`. Komponen ini bertindak sebagai penyimpanan metrik sementara di dalam memori (*in-memory store*).

Keamanan *thread* (*thread-safety*) dicapai dengan menggunakan koleksi `CopyOnWriteArrayList<ExperimentMetrics>` untuk menampung setiap catatan eksperimen (`ExperimentMetrics`) yang berisi nama penyedia, durasi penandatanganan, durasi verifikasi, ukuran tanda tangan digital, dan ukuran kunci publik.

Metrik penandatanganan direkam secara pasif oleh panggilan dari `SigningService`, sedangkan metrik durasi verifikasi diperbarui ke dalam catatan yang bersesuaian oleh `VerificationService`. Kelas `MetricsService` mengekspos data agregat rata-rata per penyedia kriptografi melalui DTO `MetricsResponse` serta data metrik migrasi statis (`MigrationMetrics` yang menampung jumlah kelas, layanan, dan konfigurasi yang dimodifikasi per skenario) ke pengontrol REST API `MetricsController`.

J. Keterbatasan Implementasi PoC

Implementasi sistem PoC dirancang di bawah batasan teknis dan fungsional spesifik sebagai berikut:

- **Penyimpanan Metrik Sementara**: Tidak ada integrasi *database*. Seluruh data metrik eksperimen disimpan dalam memori RAM, sehingga data akan terhapus ketika aplikasi *backend* dihentikan atau di-restart.
- **Ketiadaan Infrastruktur Kunci Publik (PKI)**: Sistem tidak mengimplementasikan validasi rantai sertifikat (*certificate chains*) atau otorisasi dari Otoritas Sertifikat (*Certificate Authority*). Keputusan desain untuk menggunakan pasangan kunci efemeris yang dibangkitkan secara dinamis pada PoC ini diambil sebagai *trade-off* arsitektural untuk mengukur durasi murni pemrosesan kriptografi (*signing* dan *verification*) tanpa adanya gangguan latensi dari operasi I/O basis data, penyimpanan *keystore* fisik, atau panggilan jaringan ke *Hardware Security Module* (HSM).
- **Ketiadaan Modul Autentikasi dan Otorisasi**: REST API *endpoints* pada jalur pengontrol `/api/sign` dan `/api/verify` dapat diakses secara publik tanpa proses pemeriksaan hak akses pengguna atau token autentikasi.
- **Sifat Simulasi PQC**: Tidak menggunakan algoritma PQC riil seperti ML-DSA, Falcon, maupun SLH-DSA yang sesungguhnya. Algoritma RSA-PSS-3072 digunakan sebagai pengganti simulasi perilaku arsitektur.
- **Batasan Lingkungan Uji**: Seluruh evaluasi performa dikerjakan pada mesin komputasi lokal terisolasi tanpa menyertakan latensi transmisi jaringan internet sesungguhnya.

Secara keseluruhan, implementasi berhasil merealisasikan seluruh rancangan arsitektur yang dijelaskan pada Bab III. Seluruh komponen utama, termasuk *Provider Registry*, `SignatureProvider`, serta mekanisme *algorithm switching* berhasil diimplementasikan dan siap dievaluasi secara empiris pada Bab V. Keberhasilan implementasi tersebut selanjutnya dievaluasi secara kuantitatif melalui serangkaian eksperimen yang dijelaskan pada Bab V.

V. Evaluasi Eksperimental

A. Metodologi dan Lingkungan Eksperimen

Evaluasi dilakukan dalam lingkungan terkontrol untuk meminimalkan variabel pengganggu yang dapat memengaruhi keakuratan data performa. Spesifikasi lingkungan pengujian dirancang sebagai berikut:

- **Perangkat Keras**: Prosesor Intel Core i7 generasi ke-12, memori 16 GB DDR5, dan media penyimpanan SSD.
- **Perangkat Lunak**: Sistem operasi macOS Sonoma.
- **Platform Eksekusi**: Java 23 (JDK 23) dengan menggunakan kerangka *backend* Spring Boot 3.2.5.

Pengujian diotomatisasi dengan menggunakan pengontrol API `MetricsController` pada rute `POST /api/metrics/run-experiment`. Pengukuran dilakukan sebanyak 1.000 iterasi untuk setiap penyedia kriptografi. Untuk menyingkirkan bias inisialisasi awal akibat kompilasi *Just-In-Time* (JIT) pada Java Virtual Machine (JVM), dilakukan fase pemanasan (*JVM warm-up*) sebanyak 100 iterasi sebelum pencatatan data eksperimen dimulai. Hasil akhir dari 1.000 iterasi dilaporkan sebagai nilai rata-rata (*mean*) waktu eksekusi.

B. Desain Eksperimen

Eksperimen dirancang untuk mengevaluasi efektivitas arsitektur *crypto-agile* di bawah kondisi pembebanan yang konsisten. Karakteristik utama dari desain eksperimen ini meliputi:

- **Artefak Uji yang Identik**: Setiap penyedia kriptografi diuji menggunakan berkas artefak masukan yang sama berupa data biner acak berukuran tetap sebesar 1 KB.
- **Konsistensi Lingkungan**: Seluruh skenario pengujian dijalankan pada lingkungan perangkat keras dan perangkat lunak yang identik untuk menjamin validitas perbandingan.
- **Pengulangan Statistik**: Pengulangan sebanyak 1.000 iterasi dilakukan untuk meminimalkan dampak fluktuasi kinerja CPU jangka pendek (*jitter*) sehingga dapat menghasilkan data performa yang stabil.

C. Skenario Migrasi

Tiga skenario migrasi dirancang untuk merepresentasikan fase transisi realistis dari sistem penandatanganan konvensional menuju sistem yang siap bermigrasi ke algoritma masa depan:

- **Skenario A – *Baseline* (*ECDSA Provider*)**: Menggunakan penyedia `ECDSAProvider` untuk mewakili kondisi sistem *code signing* konvensional sebelum adanya langkah migrasi.
- **Skenario B – *Hybrid Migration* (*Hybrid Provider*)**: Mensimulasikan fase transisi aktif di mana algoritma klasik dan algoritma kedua dijalankan secara paralel melalui `HybridProvider`. Hal ini bertujuan untuk mempertahankan tingkat kepercayaan dan kompatibilitas klien selama proses migrasi berlangsung.
- **Skenario C – *Simulated Provider Migration* (*Simulated Provider*)**: Menggambarkan integrasi algoritma baru ke dalam kerangka kerja *crypto-agile* melalui kelas `SimulatedProvider` berbasis RSA-PSS-3072.

Penting untuk ditegaskan bahwa `SimulatedProvider` **bukan** merupakan implementasi dari algoritma kriptografi pasca-kuantum (PQC) riil seperti ML-DSA, Falcon, maupun SLH-DSA, serta tidak menyimulasikan ukuran data aktual dari standar tersebut. Fokus dari skenario ini adalah mengevaluasi kelayakan arsitektural dalam mengintegrasikan penyedia baru tanpa perlu melakukan modifikasi pada kode layanan inti.

D. Metrik Pengujian

Evaluasi terhadap kerangka kerja dilakukan dengan membagi indikator pengukuran menjadi dua kelompok utama:

1) **Metrik Performa (Metrik Pendukung)**: Digunakan untuk mengukur dampak operasional sistem yang terdiri atas: waktu penandatanganan (*signing time*) dalam milidetik (ms), waktu verifikasi (*verification time*) dalam milidetik (ms), ukuran tanda tangan (*signature size*) dalam byte (B), dan ukuran kunci publik (*public key size*) dalam byte (B).
2) **Metrik Arsitektural / Migration Effort (Metrik Utama)**: Menjadi fokus utama penelitian untuk mengukur kesiapan migrasi arsitektural. Metrik ini diekspresikan sebagai jumlah komponen sistem yang perlu dimodifikasi ketika terjadi penambahan atau pergantian algoritma baru. Indikator yang diukur meliputi: jumlah kelas layanan yang dimodifikasi (*modified services*), jumlah komponen bisnis utama yang dimodifikasi (*modified business components*), jumlah kelas penyedia kriptografi baru yang ditambahkan (*modified providers*), dan jumlah file konfigurasi registry yang diubah (*modified configuration files*).

E. Hasil Eksperimen

1) Hasil Performa: Pengukuran performa waktu eksekusi dan ukuran muatan kunci/tanda tangan dirangkum dalam Tabel II.

Tabel II
Hasil Pengukuran Performa dan Ukuran *Payload*

Skenario	Teken (ms)	Verifikasi (ms)	Ttd (B)	Kunci (B)
Sk. A (ECDSA)	1,20	0,92	71	91
Sk. B (Hybrid)	11,31	1,83	455	513
Sk. C (Simulated)	8,18	0,37	384	422

Berdasarkan Tabel II, Skenario A (ECDSA) menunjukkan waktu eksekusi tercepat dan *payload* paling kompak. Pada Skenario B (Hybrid), terjadi kenaikan waktu *signing* menjadi 11,31 ms serta penambahan ukuran *signature* menjadi 455 byte dan ukuran kunci publik menjadi 513 byte karena keterlibatan dua algoritma secara simultan. Skenario C (Simulated) mencatatkan durasi *signing* sebesar 8,18 ms, ukuran *signature* 384 byte, dan ukuran kunci publik 422 byte akibat kompleksitas matematis operasi eksponensiasi modular pada RSA-PSS-3072.

2) Hasil Migration Effort: Pengukuran kuantitatif terhadap *migration effort* arsitektural disajikan pada Tabel III.

Tabel III
HASIL PENGUKURAN KESIAPAN MIGRASI (*Migration Effort*)

Transisi	Serv. Mod	Bisnis Mod	Prov. Add	Konfig. Mod
Sk. A $\rightarrow$ B	0	0	1	1
Sk. A $\rightarrow$ C	0	0	1	1

Data pada Tabel III membuktikan bahwa penambahan algoritma baru (baik untuk skema hibrida maupun simulasi algoritma baru) sama sekali tidak menyentuh lapisan bisnis maupun komponen layanan aplikasi (*modified services* = 0, *modified business components* = 0). Perubahan hanya terjadi secara terlokalisasi pada penambahan berkas kelas penyedia baru (*modified providers* = 1) dan satu baris perubahan konfigurasi penanda nama algoritma aktif (*modified configuration files* = 1).

Hasil ini menunjukkan bahwa manfaat utama pendekatan *crypto-agile* tidak terletak pada peningkatan performa, melainkan pada kemampuan mengisolasi dampak perubahan algoritma ke satu titik implementasi. Temuan ini konsisten dengan prinsip *crypto agility*, *provider pattern*, *open/closed principle*, dan *separation of concerns* yang telah dibahas pada Bab II. Hasil ini memberikan validasi empiris terhadap konsep *crypto agility* yang telah dijelaskan pada Bab II. Keberhasilan pencatatan *migration effort* yang rendah secara konkret menunjukkan bahwa *Provider Pattern*, *Open/Closed Principle*, dan *Separation of Concerns* bukan hanya sekadar konsep teoritis, melainkan properti arsitektural yang benar-benar terwujud dan terukur pada implementasi aktual *framework* yang dibangun.

Berdasarkan teori pada Bab II dan analisis konseptual arsitektur tradisional yang *tightly-coupled*, penambahan algoritma baru tanpa abstraksi *crypto-agile* akan memicu efek riak (*ripple effect*). Pengembang dipaksa melakukan modifikasi langsung pada kelas layanan utama seperti `SigningService` dan `VerificationService`, mengubah kelas pembungkus data (DTO), serta memodifikasi setidaknya 2 komponen bisnis utama lainnya, yang dapat mengakibatkan tingginya risiko regresi kode.

F. Analisis Hasil

1) Analisis Performa: Overhead waktu eksekusi pada penandatanganan Skenario B (11,31 ms) dan Skenario C (8,18 ms) dipicu oleh karakteristik komputasi RSA-PSS-3072 yang lebih berat dibandingkan dengan kriptografi kurva eliptis (ECDSA secp256r1). Operasi eksponensiasi modular dengan basis kunci besar (3072-bit) membutuhkan siklus *clock* CPU yang lebih banyak. Pada proses verifikasi, peningkatan waktu eksekusi relatif lebih kecil (1,83 ms pada Skenario B dan 0,37 ms pada Skenario C) karena operasi verifikasi RSA secara matematis jauh lebih cepat dibandingkan dengan operasi penandatanganannya. Fenomena verifikasi Skenario C yang lebih cepat dibandingkan Skenario A (0,92 ms) secara matematis disebabkan oleh sifat operasi kunci publik RSA-PSS yang menggunakan eksponen publik kecil (umumnya $e = 65537$). Operasi ini hanya membutuhkan sedikit operasi perkalian modular (sekitar 17 kali), berbeda dengan verifikasi ECDSA pada Skenario A yang melibatkan perkalian titik kurva eliptik yang lebih kompleks secara komputasi. Karakteristik ini didukung pula oleh tingkat optimasi pustaka kriptografi dasar (Bouncy Castle) yang digunakan dalam lingkungan eksekusi PoC.

Kenaikan ukuran *payload* tanda tangan dan kunci publik pada Skenario B (455 byte untuk tanda tangan dan 513 byte untuk kunci publik) serta Skenario C (384 byte untuk tanda tangan dan 422 byte untuk kunci publik) merupakan akibat langsung dari panjang representasi matematis kunci RSA-PSS-3072. Meskipun terjadi peningkatan waktu dan ukuran *payload*, *trade-off* ini sepenuhnya dapat diterima (*acceptable*) dalam ekosistem distribusi perangkat lunak. Proses *code signing* umumnya dilakukan sebagai bagian dari alur *build* integrasi kontinu (CI/CD) yang berjalan secara asinkron, sehingga tambahan waktu beberapa milidetik tidak mengganggu pengalaman pengguna akhir.

2) Analisis Migration Effort: Analisis terhadap data *migration effort* menunjukkan efektivitas pola desain arsitektur yang diusulkan. Kunci keberhasilan isolasi dampak perubahan ini bersandar pada tiga pilar utama:

- **Provider Abstraction Pattern & SignatureProvider Interface**: Berkat pemisahan kontrak antarmuka, kelas layanan utama seperti `SigningService` dan `VerificationService` hanya berinteraksi dengan tipe data abstrak `SignatureProvider`. Detail internal dari algoritma baru (seperti parameter JCA atau pustaka Bouncy Castle) terbungkus rapi di dalam kelas penyedia konkret.
- **Mekanisme Registry Dinamis**: Penggunaan `ProviderRegistry` menghilangkan kebutuhan deklarasi kelas secara manual atau instansiasi langsung melalui operator `new`. Pemanfaatan injeksi dependensi Spring secara otomatis mendaftarkan penyedia baru ke dalam registry saat aplikasi *booting*.
- **Open/Closed Principle**: Kepatuhan terhadap prinsip ini terbukti secara empiris. Sistem terbuka terhadap ekstensi (menambahkan `SimulatedProvider` baru) tetapi tertutup terhadap modifikasi (tidak ada perubahan pada logika layanan pengkalkulasi hash atau pembentuk paket data).

G. Diskusi Validitas Hasil

1) Validitas Internal:

- **Efek JVM *Warm-up***: Karakteristik eksekusi Java Virtual Machine (JVM) yang melakukan kompilasi dinamis (JIT) dapat menyebabkan waktu eksekusi iterasi awal menjadi sangat lambat. Hal ini diatasi dengan melakukan 100 iterasi pemanasan sebelum pencatatan data dilakukan.
- **CPU Jitter**: Proses latar belakang pada sistem operasi dapat menginterupsi jalannya pengujian. Dampak ini

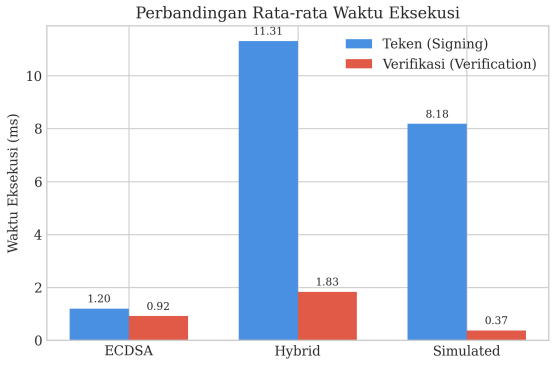
ditekan dengan mengambil rata-rata dari 1.000 iterasi untuk menekan fluktuasi pengukuran komputasi dan menjamin kestabilan data rata-rata yang diambil.

2) Validitas Eksternal:

- **Lingkungan Produksi**: Pengujian PoC ini dilakukan pada satu *node* komputasi lokal. Pada implementasi nyata di lingkungan produksi, faktor konkurensi tinggi, akses penyimpanan jaringan, dan latensi *gateway* HTTP REST dapat mendominasi total waktu respons.
- **Ketiadaan PKI Riil**: Pada implementasi komersial, proses verifikasi harus memvalidasi seluruh rantai sertifikat (*certificate chains*) hingga ke *root* CA terpercaya. Langkah validasi sertifikat ini akan menambah *overhead* waktu verifikasi secara signifikan.
- **Simulasi Karakteristik PQC**: `SimulatedProvider` menggunakan RSA-PSS-3072 untuk merepresentasikan algoritma masa depan. Pada kenyataannya, algoritma PQC standar seperti ML-DSA-65 memiliki ukuran tanda tangan yang jauh lebih besar (∼3.300 byte). Meskipun demikian, karena arsitektur data `SignedPackage` menggunakan pemetaan biner ter-encode Base64 yang dinamis, peningkatan ukuran ini tidak akan merusak struktur data sistem melainkan hanya akan meningkatkan *overhead* penyimpanan dan transmisi data.

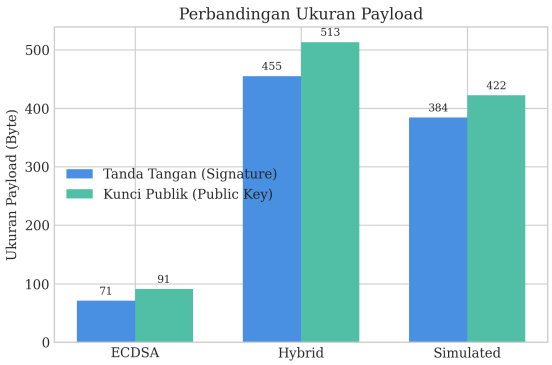
H. Visualisasi Hasil Eksperimen

Visualisasi komparatif data performa dan ukuran *payload* antar skenario disajikan pada Gambar 5 dan Gambar 6.



Gambar 5. Grafik batang perbandingan rata-rata waktu eksekusi *signing* dan *verification* (dalam ms) antar skenario.

Secara keseluruhan, evaluasi eksperimental ini membuktikan secara empiris bahwa kerangka kerja *code signing* yang dirancang berhasil mencapai target *crypto agility* tingkat tinggi. Integrasi algoritma baru dapat dituntaskan hanya dengan perubahan terlokalisasi pada lapisan *provider*, sementara logika bisnis aplikasi tetap stabil dan aman dari risiko regresi kode, dengan konsekuensi *overhead* performa yang masih dalam batas toleransi operasional. Temuan ini menjadi landasan analisis yang lebih mendalam pada Bab VI.



Gambar 6. Grafik batang perbandingan ukuran *signature* dan *public key* (dalam byte) antar skenario.

VI. Diskusi

A. *Efektivitas Pendekatan* Crypto Agility

Hasil evaluasi arsitektural menunjukkan bahwa *crypto agility* bukan sekadar pilihan operasional, melainkan struktur fondasi yang menentukan bagaimana sistem beradaptasi terhadap perubahan standar keamanan di masa depan. Melalui pemisahan yang tegas antara logika aplikasi dan mekanisme kriptografi menggunakan *Provider Registry* dan antarmuka `SignatureProvider`, efek riak (*ripple effect*) dari perubahan algoritma berhasil dieliminasi sepenuhnya. Hal ini dibuktikan secara empiris dari nilai metrik utama *migration effort* di mana perubahan pada modul layanan dan logika bisnis bernilai nol (*modified services* = 0). Temuan ini kontras dengan karakteristik arsitektur sistem tradisional yang tidak fleksibel (*tightly-coupled*) seperti diuraikan oleh Paul dan Stübs [7] serta Alnahawi [4]. Pada sistem tradisional, ketiadaan lapisan abstraksi kriptografi memaksa setiap integrasi algoritma baru memicu perubahan yang menyebar langsung ke komponen inti (*modified services* > 0) dan komponen bisnis utama (*modified business components* > 0), yang secara drastis meningkatkan risiko regresi keamanan.

Signifikansi temuan ini terletak pada kemampuan sistem untuk mengisolasi kerentanan arsitektural. Perubahan algoritma kriptografi dalam sistem tradisional sering kali diperlakukan sebagai masalah implementasi tingkat rendah. Namun, ketika detail algoritma konkret bocor ke lapisan bisnis, setiap upaya migrasi akan memicu modifikasi massal yang rentan terhadap galat. Desain *crypto-agile* yang dievaluasi dalam penelitian ini membuktikan bahwa dengan membatasi dependensi hanya pada kontrak abstrak, stabilitas struktur sistem dapat dipertahankan secara utuh. Keberhasilan menjaga stabilitas ini jauh lebih penting daripada biaya latensi operasional milidetik, karena stabilitas kode secara langsung menekan biaya pemeliharaan (*maintenance cost*) dan memperkecil risiko regresi keamanan selama masa transisi.

B. Migration Effort *sebagai Metrik Arsitektur*

Dalam literatur transisi kriptografi pasca-kuantum (PQC), mayoritas penelitian berfokus pada evaluasi performa komputasi kriptografis murni seperti *throughput*, latensi penandatanganan, dan ukuran muatan kunci publik atau tanda tangan digital. Meskipun metrik performa tersebut penting untuk menguji kelayakan operasional pada perangkat dengan sumber daya terbatas, metrik tersebut memiliki keterbatasan dalam menggambarkan kesiapan organisasi dalam mengadopsi standar baru secara menyeluruh. Penelitian ini memperkenalkan perspektif yang berbeda dengan menjadikan *migration effort* struktural sebagai metrik evaluasi utama.

Metrik *migration effort* mengalihkan fokus dari optimasi algoritmik tingkat rendah menuju pengukuran objektif terhadap "biaya perubahan sistem" (*cost of change*). Bagi organisasi skala besar dengan basis kode yang masif, biaya terbesar dari migrasi kriptografi bukan berasal dari konsumsi CPU, melainkan dari upaya rekayasa perangkat lunak dan pengujian ulang sistem yang terdampak oleh perubahan tanda tangan metode. Dengan menyajikan kerangka pengukuran kuantitatif terhadap jumlah komponen yang dimodifikasi, penelitian ini membuktikan bahwa *migration effort* dapat menjadi indikator yang andal untuk menilai kelayakan desain perangkat lunak dari perspektif skalabilitas migrasi dan kemampuan adaptasi jangka panjang.

C. Keterbatasan dan Ancaman Validitas

Terdapat beberapa batasan metodologis yang memengaruhi tingkat generalisasi dari temuan penelitian ini. Pertama, ketiadaan Infrastruktur Kunci Publik (PKI) yang terintegrasi penuh menyebabkan proses verifikasi pada sistem PoC ini belum sepenuhnya merepresentasikan alur kerja lingkungan produksi. Pada *deployment* produksi nyata, proses verifikasi tanda tangan digital harus memvalidasi seluruh rantai sertifikat (*certificate chain*) hingga ke *root* CA tepercaya, memeriksa pencabutan sertifikat (CRL/OCSP), dan memverifikasi atribut *timestamping*. Ketiadaan rantai validasi sertifikat ini berimplikasi pada nilai rata-rata waktu verifikasi yang diperoleh dalam eksperimen ini, di mana *overhead* verifikasi sesungguhnya di lingkungan produksi dipastikan akan jauh lebih besar.

Kedua, penggunaan algoritma klasik RSA-PSS-3072 sebagai representasi `SimulatedProvider` membatasi generalisasi performa terhadap skema PQC riil. Meskipun parameter panjang kunci dan ukuran tanda tangan dirancang lebih besar daripada ECDSA untuk menyimulasikan dampak muatan data, RSA-PSS memiliki kompleksitas matematis dan karakteristik penggunaan memori yang berbeda dibandingkan dengan standar PQC seperti ML-DSA (FIPS 204) atau Falcon. Oleh karena itu, metrik performa waktu eksekusi dalam penelitian ini tidak boleh diinterpretasikan secara langsung sebagai kinerja empiris dari migrasi PQC aktual, melainkan harus dipandang sebagai simulasi kelayakan arsitektural sistem.

D. Implikasi terhadap Migrasi PQC Nyata

Rancangan *framework crypto-agile* ini memberikan panduan praktis yang berharga bagi arsitek perangkat lunak dalam menghadapi standardisasi kriptografi pasca-kuantum yang dinamis. Ketika pustaka kriptografi pasca-kuantum (seperti implementasi ML-DSA dan Falcon pada pustaka Bouncy Castle) telah stabil dan siap digunakan di lingkungan produksi, organisasi dapat mengintegrasikannya tanpa perlu mendesain ulang sistem. Integrasi tersebut dilakukan cukup dengan membuat satu kelas konkret penyedia baru (misalnya `MLDSAProvider`) yang mengimplementasikan antarmuka `SignatureProvider` dan mendaftarkannya ke dalam `ProviderRegistry`.

Pendekatan abstraksi penyedia ini secara langsung memitigasi risiko keterikatan pada vendor (*vendor lock-in*) atau algoritma tunggal. Jika di masa mendatang salah satu standar algoritma PQC terbukti rentan terhadap serangan kriptanalisis baru, sistem dapat beralih ke algoritma alternatif hanya dengan mengubah konfigurasi string nama penyedia aktif, tanpa perlu melakukan penulisan ulang kode pada layanan utama. Selain itu, kemampuan *framework* untuk menjalankan `HybridProvider` memungkinkan organisasi menerapkan strategi penyebaran hibrida (*hybrid deployment*) selama masa transisi. Dengan menerapkan verifikasi ganda (ECDSA klasik dan PQC), organisasi dapat menjamin kompatibilitas mundur dengan klien lama sekaligus meningkatkan tingkat kepercayaan keamanan terhadap ancaman komputer kuantum secara bertahap.

VII. KESIMPULAN

A. Ringkasan Kontribusi

Makalah ini menyajikan rancangan, implementasi, dan evaluasi eksperimental sebuah *crypto-agile code signing framework* untuk memfasilitasi transisi aman menuju kriptografi pasca-kuantum. Melalui penyediaan antarmuka `SignatureProvider` yang memisahkan logika aplikasi dari detail algoritma, *framework* berhasil membuktikan kelayakan desain arsitektur yang fleksibel.

Tiga penyedia konkret diimplementasikan: *ECDSA Provider* sebagai *baseline*, *Hybrid Provider* sebagai skema transisi hibrida, dan *Simulated Provider* sebagai representasi integrasi algoritma baru ke dalam *framework*. Temuan utama penelitian ini menunjukkan bahwa pendekatan *crypto-agile* berhasil meminimalkan *migration effort* secara signifikan dengan nilai *modified services* = 0, membuktikan bahwa migrasi dapat dilakukan tanpa modifikasi pada logika bisnis aplikasi inti. Meskipun terdapat *overhead* latensi akibat ukuran kunci yang lebih besar, *trade-off* tersebut sepenuhnya dapat diterima demi stabilitas struktur sistem dan isolasi dampak perubahan algoritma. Kesiapan arsitektural ini menjadi kontribusi utama dalam mengurangi kompleksitas pemeliharaan kode dan risiko kegagalan sistem selama masa migrasi kriptografi global.

B. Saran Penelitian Lanjutan

Berdasarkan keterbatasan dan temuan yang diidentifikasi, terdapat beberapa arah penelitian lanjutan yang menjanjikan:

- **Integrasi algoritma PQC aktual**: Menggantikan *Simulated Provider* dengan implementasi ML-DSA atau Falcon aktual menggunakan Bouncy Castle PQC API yang semakin matang [3].
- **Pengujian pada infrastruktur produksi**: Mendeploy *framework* pada sistem distribusi perangkat lunak skala produksi untuk mengukur performa dan *migration effort* dalam kondisi nyata [10].
- **Otomasi analisis dependensi kriptografi**: Mengembangkan alat analisis statis yang dapat secara otomatis mengidentifikasi titik-titik *coupling* kriptografi dalam basis kode yang ada [4].

Video Link at YouTube

Tautan video demonstrasi pengujian fungsional dan eksperimen *framework* di YouTube:

https://youtu.be/GRBLL3vAQPk

Acknowledgment

Penulis mengucapkan terima kasih kepada dosen pengampu mata kuliah II4021 Kriptografi, Program Studi Sistem dan Teknologi Informasi, Institut Teknologi Bandung, atas bimbingan dan arahan selama proses penulisan makalah ini.

References

[1] K. F. Hasan et al., "A framework for migrating to post-quantum cryptography: Security dependency analysis and case studies," *IEEE Access*, vol. 12, pp. 23427–23450, 2024.

[2] D. Alagic et al., "Status report on the third round of the NIST post-quantum cryptography standardization process," NIST Interag. Rep. 8413, Nat. Inst. Standards and Technology, 2022.

[3] National Institute of Standards and Technology, "Module-lattice-based digital signature standard," Fed. Inf. Process. Stand. Publ. 204, Aug. 2024.

[4] N. Alnahawi et al., "ELCA: Introducing enterprise-level cryptographic agility for a post-quantum era," Cryptology ePrint Archive, Rep. 2023/1539, 2023.

[5] Y. Kwon et al., "Compact hybrid signature for secure transition to post-quantum era," *IEEE Access*, 2024.

[6] A. Kalu et al., "Why software signing (still) matters: Trust boundaries in the software supply chain," arXiv:2510.04964, 2025.

[7] S. Paul and M. Stübs, "Problems and new approaches for crypto-agility in operational technology," Cryptology ePrint Archive, Rep. 2024/1386, 2024.

[8] D. Ott and C. Peikert, "Identifying research challenges in post-quantum cryptography migration and cryptographic agility," arXiv:1909.07353, 2019.

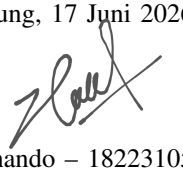
[9] B. Schneier, *Applied Cryptography: Protocols, Algorithms, and Source Code in C*, 2nd ed. New York, NY, USA: John Wiley & Sons, 1996.

[10] C. Näther et al., "Migrating software systems towards post-quantum cryptography: A systematic literature review," arXiv:2404.12854, 2024.

Pernyataan

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, 17 Juni 2026



Zheannetta Apple Haihando – 18223105