Implementasi dan Analisis Sistem Verifikasi Integritas Mod Minecraft Menggunakan SHA-256 dan Tanda Tangan Digital ECDSA

Alma Felicia Vielrizki - 18223112
Program Studi Sistem dan Teknologi Informasi
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: almavielrizki@gmail.com , 18223112@std.stei.itb.ac.id

Abstract*—Mod Minecraft didistribusikan dalam bentuk arsip Java (.jar) yang dieksekusi langsung oleh Java Virtual Machine, sehingga modifikasi tidak sah pada file tersebut dapat membahayakan sistem pengguna secara langsung. Insiden malware Fractureiser pada tahun 2023 membuktikan ancaman ini bersifat nyata, menyusup ke salah satu modpack populer dengan lebih dari 4,6 juta unduhan di CurseForge. Mekanisme verifikasi yang sudah ada, seperti penandatanganan JAR pada Forge, terbukti tidak mendeteksi perubahan konten file. Makalah ini merancang, mengimplementasikan, dan menguji sebuah sistem verifikasi integritas dua lapis yang menggabungkan fungsi hash SHA-256 untuk deteksi perubahan konten dengan tanda tangan digital ECDSA pada kurva secp256k1 untuk verifikasi otentisitas sumber. Sistem diuji terhadap empat skenario serangan: injeksi byte, injeksi kelas Java palsu, pemalsuan tanda tangan, dan replay attack menggunakan manifes kedaluwarsa. Hasil eksperimen menunjukkan sistem berhasil mendeteksi seluruh skenario serangan dengan waktu verifikasi rata-rata di bawah 6 milidetik untuk file berukuran hingga 2 MB. Perbandingan performa menunjukkan ECDSA menghasilkan tanda tangan yang signifikan lebih ringkas dibanding RSA-2048 dengan waktu pembuatan kunci yang jauh lebih cepat. Pengujian avalanche effect pada SHA-256 mengonfirmasi bahwa modifikasi satu bit menghasilkan perubahan rata-rata mendekati 50% dari total bit hash, sesuai sifat teoritis fungsi hash kriptografis yang baik.

Keywords——integritas file, SHA-256, ECDSA, tanda tangan digital, keamanan mod, Minecraft.

I. Pendahuluan

A. Latar Belakang

Minecraft adalah permainan video sandbox yang dikembangkan oleh Mojang Studios dan dirilis secara resmi pada tahun 2011. Hingga April 2025, permainan ini telah terjual lebih dari 350 juta salinan di seluruh dunia, menjadikannya permainan video terlaris sepanjang masa [1]. Dengan lebih dari 200 juta pemain aktif bulanan pada tahun 2025 [2], Minecraft telah melampaui definisi permainan konvensional dan berkembang menjadi sebuah platform kreatif yang digunakan secara luas, bahkan dalam konteks pendidikan dan pengembangan perangkat lunak komunitas.

Salah satu faktor utama yang mempertahankan relevansi Minecraft adalah ekosistem modifikasi (mod) yang sangat aktif. Platform distribusi mod terkemuka seperti CurseForge dan Modrinth menjadi pusat bagi komunitas pengembang mod global. CurseForge sendiri mencatat lebih dari 800 juta unduhan mod per bulan dengan 11 juta pengguna aktif [3], sementara total unduhan mod Minecraft di platform tersebut telah melampaui 100 miliar pada awal tahun 2026 [4]. Mod-mod ini umumnya didistribusikan dalam bentuk arsip Java (Java Archive / .jar) yang dieksekusi langsung oleh Java Virtual Machine bersama proses utama Minecraft, sehingga memiliki akses yang sangat luas terhadap sumber daya sistem komputer pengguna.

Ekosistem *mod* Minecraft memiliki struktur yang unik dibandingkan dengan platform modifikasi permainan lainnya. Tidak seperti sistem mod pada permainan seperti The Elder Scrolls V: Skyrim yang umumnya menggunakan file data terpisah atau Kerbal Space Program yang memiliki API mod terisolasi, *mod* Minecraft berjalan dalam ruang memori yang sama dengan proses utama permainan. Hal ini berarti bahwa kode berbahaya dalam sebuah *mod* dapat mengakses tidak hanya data permainan, tetapi juga file sistem, kredensial pengguna, dan bahkan perangkat keras yang terhubung. Tingkat akses ini setara dengan menjalankan program *arbitrer* tanpa batasan hak istimewa, menjadikan verifikasi keamanan sebagai kebutuhan kritis yang sering diabaikan oleh pengguna awam.

Karakteristik eksekusi langsung ini membawa konsekuensi keamanan yang serius. Karena tidak ada mekanisme sandboxing yang ketat antara mod dan sistem operasi host, sebuah file .jar yang telah dimanipulasi secara jahat memiliki kemampuan setara dengan program biasa yang dijalankan oleh pengguna. Celah ini telah dieksploitasi secara nyata. Pada Juni 2023, insiden keamanan berskala besar terjadi ketika malware bernama Fractureiser ditemukan tertanam dalam sejumlah proyek mod yang diunggah ke CurseForge dan BukkitDev [5]. Malware ini beroperasi dalam empat tahap eksekusi dan mampu melakukan berbagai tindakan berbahaya, pertama, menyebarkan dirinya sendiri ke seluruh file .jar yang ditemukan di sistem, mencuri cookies dan kredensial yang tersimpan di peramban web, mengalihkan alamat dompet kripto yang tersalin di clipboard, serta mengekstraksi kredensial akun Microsoft, Minecraft, dan Discord [6][7].

Urgensi permasalahan ini semakin meningkat seiring dengan berkembangnya tren *modpack*, yaitu kumpulan puluhan hingga ratusan *mod* yang dikemas dan didistribusikan sebagai satu kesatuan. *Modpack* memungkinkan pengguna menginstal ratusan berkas .jar secara otomatis dalam satu sesi, yang berarti satu berkas yang terkompromi di dalam *modpack* dapat menyebar ke seluruh basis pengguna *modpack* tersebut tanpa sepengetahuan siapa pun. Fenomena ini memperluas permukaan serangan secara eksponensial. Investigasi pasca-insiden Fractureiser mengungkapkan bahwa malware tersebut aktif menyebarkan dirinya ke berkas .jar lain di sistem yang terinfeksi, termasuk berkas dari modpack yang tidak terkait, sehingga satu pengguna yang terinfeksi berpotensi menjadi vektor penyebaran ke pengguna lain melalui mekanisme berbagi konfigurasi *modpack*. Realitas ini menegaskan bahwa solusi keamanan yang diperlukan bukan sekadar pemindaian terpusat, melainkan mekanisme verifikasi yang dapat dijalankan secara independen oleh setiap pengguna di sisi klien.

Insiden ini mengungkap kerentanan mendasar pada rantai distribusi mod Minecraft yaitu tidak adanya mekanisme verifikasi integritas file yang memadai dari sisi pengguna akhir. Meskipun platform seperti Modrinth telah mewajibkan pencantuman nilai hash SHA-1 dan SHA-512 dalam spesifikasi format paket mereka [8], dan Forge menyediakan mekanisme penandatanganan JAR, penelitian menunjukkan bahwa sistem penandatanganan JAR milik Forge tidak secara efektif mendeteksi perubahan pada konten file di dalam arsip [9]. Selain itu, tidak satu pun sistem yang ada saat ini menggabungkan verifikasi integritas konten dengan verifikasi otentisitas sumber menggunakan tanda tangan digital kriptografis modern. Kondisi ini menjadi motivasi utama penelitian ini.

B. *Rmusan Masalah*

- Bagaimana cara mendeteksi secara otomatis apakah file .jar mod Minecraft telah dimodifikasi dari versi aslinya?
- Apakah penambahan tanda tangan digital ECDSA pada sistem hash memberikan perlindungan yang lebih kuat dibandingkan hash checking saja?
- Bagaimana perbandingan performa antara RSA-2048 dan ECDSA dalam konteks verifikasi mod?

A. *Tujuan*

- Merancang sistem verifikasi integritas dua lapis (SHA-256 + ECDSA) untuk file mod Minecraft
- Mengimplementasikan sistem sebagai *proof-of-concept tool* berbasis Python
- Menguji sistem terhadap empat skenario serangan dengan injeksi *byte*, injeksi *class*, pemalsuan *signature*, dan *replay attack*

B. *Batasan Masalah*

- Sistem diimplementasikan sebagai *standalone tool.*
- Pengujian dilakukan pada file .jar mod *open-source* yang diunduh dari Modrinth.
- Tidak mencakup deteksi cheat runtime.

II. Tinjauan Pustaka

A. *Fungsi Hash SHA-256*

SHA-256 adalah fungsi hash kriptografis yang menghasilkan keluaran berukuran tetap 256-bit dari masukan berukuran sembarang, mengikuti struktur Merkle–Damgård sesuai standar FIPS 180-4 [10]. Fungsi hash yang baik memiliki sifat *collision resistance*, yaitu sulit menemukan dua masukan berbeda yang menghasilkan keluaran hash yang sama, serta avalanche effect, yaitu perubahan sekecil satu bit pada masukan menghasilkan perubahan signifikan pada keluaran. SHA-256 dipilih dalam penelitian ini karena MD5 dan SHA-1 telah memiliki *collision attack* praktis yang terdokumentasi, sehingga tidak lagi direkomendasikan untuk aplikasi yang menuntut keamanan integritas.

B. *Tanda Tangan Digital ECDSA*

Elliptic Curve Digital Signature Algorithm (ECDSA) adalah skema tanda tangan digital berbasis kriptografi kurva eliptik sesuai standar FIPS 186-5 [11]. Berbeda dengan RSA yang keamanannya bergantung pada kesulitan faktorisasi bilangan besar, ECDSA bergantung pada kesulitan masalah logaritma diskret pada kurva eliptik (ECDLP). Pada tingkat keamanan yang setara, ECDSA membutuhkan ukuran kunci yang jauh lebih kecil dibandingkan RSA, sehingga relevan untuk skenario yang menuntut signature ringkas dan komputasi cepat [12]. Penelitian ini menggunakan kurva secp256k1, kurva yang juga digunakan secara luas pada Bitcoin.

C. *Struktur File JAR Minecraft*

File .jar adalah arsip ZIP standar yang berisi berkas bytecode Java (.class), metadata pada direktori META-INF, serta aset pendukung lain. Karena formatnya identik dengan ZIP, file .jar dapat dibuka, dibaca, maupun dimodifikasi menggunakan pustaka pemrosesan ZIP standar tanpa memerlukan parser khusus.

D. *Sistem Verifikasi yang Sudah Ada*

Forge menyediakan mekanisme penandatanganan JAR bawaan, namun sebuah laporan resmi pada repositori GitHub MinecraftForge menunjukkan bahwa mekanisme ini tidak memvalidasi ulang isi file setelah proses ekstraksi, sehingga modifikasi konten internal .jar tidak terdeteksi meskipun signature tetap tampak valid [10]. Di sisi lain, format modpack Modrinth (.mrpack) mewajibkan pencatuman hash SHA-1 dan SHA-512 untuk setiap berkas [9], namun pendekatan ini hanya memverifikasi integritas konten tanpa memverifikasi otentisitas sumber, karena tidak melibatkan tanda tangan digital. Penelitian ini memosisikan kontribusinya pada titik yang belum dijawab oleh kedua pendekatan tersebut, yaitu penggabungan verifikasi integritas konten dan verifikasi otentisitas sumber dalam satu sistem yang sama.

E. *Keamanan Rantai Pasok Perangkat Lunak*

Permasalahan yang dihadapi ekosistem *mod* Minecraft dapat dipahami dalam konteks yang lebih luas sebagai bagian dari ancaman keamanan rantai pasok perangkat lunak *(software supply chain security)*. Serangan rantai pasok menargetkan proses distribusi perangkat lunak itu sendiri, bukan kerentanan pada kode yang dijalankan pengguna.

Laporan dari National Institute of Standards and Technology (NIST) mendefinisikan rantai pasok perangkat lunak mencakup seluruh entitas yang terlibat dalam pengembangan, pengemasan, distribusi, dan pemeliharaan perangkat lunak, termasuk pihak ketiga yang kodenya diintegrasikan ke dalam produk akhir.. Insiden SolarWinds pada tahun 2020 dan kompromi repositori PyPI yang berulang menunjukkan bahwa serangan rantai pasok mampu mengakibatkan dampak yang jauh lebih luas dibandingkan eksploitasi kerentanan konvensional, karena memanfaatkan kepercayaan yang telah terbentuk antara pengguna dan sumber distribusi resmi. Dalam konteks mod Minecraft, *platform* distribusi seperti CurseForge berperan sebagai titik kepercayaan sentral yang apabila dikompromi, baik melalui kompromi akun pengembang maupun infrastruktur *platform*, akan langsung mempengaruhi seluruh basis penggunanya.

III. Metodologi Dan Rancangan Sistem

A. Arsitektur Sistem

Sistem dirancang dengan dua peran utama, yaitu *Publisher* dan *Verifier*. Publisher menghitung hash SHA-256 dari berkas .jar, menandatangani hash tersebut menggunakan kunci privat ECDSA, lalu menerbitkan manifes yang berisi hash, signature, dan kunci publik. *Verifier* menerima berkas .jar beserta manifesnya, menghitung ulang hash secara lokal, membandingkannya dengan manifes, kemudian memverifikasi signature menggunakan kunci publik sebelum memutuskan status ACCEPT atau REJECT.

B. Format Manifes JSON

Manifes dirancang dalam format JSON agar mudah dibaca dan diproses lintas platform. Berikut contoh manifes nyata yang dihasilkan sistem untuk salah satu berkas mod pada dataset pengujian.

```
{
  "mod_name": "examplemod_appleskin.jar",
  "sha256": "62b34262c0ad02e9d2b31a4bc653
  ba22472c57c91a8052ad3ef6016545b9162e",
  "signature": "304502200c20f64dce74d442
    4682239427a04a4fab858c7957e7528d62be...",
  "algorithm": "ECDSA-secp256k1",
  "public_key": "-----BEGIN PUBLIC KEY---",
  "issued_at": "2026-06-10T10:00:00Z",
  "expires_at": "2027-06-10T10:00:00Z"
}
```

Figure 1. Contoh manifes hasil implementasi sistem.

C. Alur Verifikasi

Proses verifikasi pada sisi *Verifier* mengikuti empat langkah berurutan. Pertama, manifes dimuat dari sumber yang dipercaya. Kedua, hash SHA-256 dihitung ulang dari berkas .jar lokal dan dibandingkan dengan nilai pada manifes. Ketidaksesuaian akan langsung menghasilkan REJECT dengan alasan hash tidak sama. Ketiga, jika hash sesuai, *signature* pada manifes diverifikasi menggunakan kunci publik yang tertera. Kegagalan verifikasi akan menghasilkan REJECT dengan alasan *invalid signature*. Keempat, sistem memeriksa apakah manifes telah melewati batas waktu kadaluarsa, jika ya, status REJECT dengan alasan manifest expired. Berkas dinyatakan ACCEPT hanya jika seluruh tahap tersebut terlewati.

D. Skenario Serangan yang Diuji

Empat skenario dirancang untuk menguji ketahanan sistem. Skenario A merupakan baseline menggunakan berkas asli dengan manifes valid, yang seharusnya menghasilkan ACCEPT. Skenario B1 menyisipkan rangkaian byte acak pada akhir berkas .jar untuk mensimulasikan korupsi atau injeksi data. Skenario B2 menyisipkan sebuah berkas .class palsu ke dalam struktur ZIP berkas .jar untuk mensimulasikan injeksi *payload* berbahaya yang lebih realistis. Skenario C memasangkan *signature* dari manifes mod lain ke berkas target untuk mensimulasikan pemalsuan signature. Skenario D memasangkan manifes yang telah melewati tanggal kadaluarsanya pada berkas yang sebenarnya valid, untuk mensimulasikan *replay attack* menggunakan manifes lama.

IV. Implementasi

A. Environment dan Dataset

Sistem diimplementasikan menggunakan Python 3.11 dengan pustaka hashlib untuk hashing, cryptography untuk operasi RSA dan ECDSA, serta zipfile untuk manipulasi struktur arsip .jar. Dataset pengujian terdiri atas lima berkas .jar dengan struktur ZIP yang menyerupai mod nyata (mengandung direktori META-INF, berkas mods.toml, dan berkas .class), berukuran 49,58 KB hingga 1968,26 KB untuk merepresentasikan variasi ukuran mod ringan hingga mod kompleks seperti pada mod gameplay berskala besar.

B. Implementasi Hashing

Fungsi hashing membaca berkas secara bertahap (chunked reading) sebesar 64 KB per iterasi untuk menjaga efisiensi memori pada berkas besar, sebagaimana ditunjukkan pada potongan kode berikut.

```
def hash_file_sha256(path, chunk_size=65536):
   h = hashlib.sha256()
   with open(path, "rb") as f:
        while True:
               chunk = f.read(chunk_size)
               if not chunk:
               break
               h.update(chunk)
   return h.hexdigest()
```

Figure 2. Potongan kode hash

C. Implementasi Signing dan Verifikasi

Pembuatan kunci ECDSA menggunakan kurva secp256k1, dengan proses *signing* dan *verification* dilakukan terhadap *digest* SHA-256 yang telah dihitung sebelumnya agar konsisten dengan hash yang tercatat pada manifes.

```
private_key = ec.generate_private_key(
   ec.SECP256K1())

signature = private_key.sign(
   digest_bytes,
   ec.ECDSA(utils.Prehashed(
        hashes.SHA256())))
```

Figure 3. Potongan kode kunci ECDSA

D. Implementasi Simulasi Serangan

Simulasi injeksi *byte* dilakukan dengan menambahkan 256 *byte* acak pada akhir berkas, sementara simulasi injeksi kelas menggunakan mode *append* pada modul zipfile untuk menyisipkan berkas .class baru ke dalam struktur arsip tanpa merusak entri ZIP yang sudah ada.

```
def attack_class_injection(src, dst):
   shutil.copyfile(src, dst)
   with zipfile.ZipFile(dst, "a") as zf:
       zf.writestr(
            "com/example/mod/Injected.class",
              random.randbytes(4096))
```

Figure 4. Potongan kode injeksi

V. Hasil Dan Analisis

A. Hasil Verifikasi Skenario Serangan

Tabel I menunjukkan hasil pengujian keempat skenario serangan terhadap berkas examplemod_appleskin.jar. Seluruh skenario serangan berhasil terdeteksi dan ditolak oleh sistem sesuai dengan hasil yang diharapkan, dengan waktu verifikasi keseluruhan berada pada rentang sub milidetik.

TABLE I.

Skenario	Hasil	Alasan	Waktu (ms)
A - *Baseline*	ACCEPT	OK	0.8842
B1 - Byte Injection	REJECT	Hash mismatch	0.2267
B2 - Class Injection	REJECT	Hash mismatch	0.2430
C - Signature Forgery	REJECT	Invalid signature	0.7716
D - Replay Attack	REJECT	Manifest expired	0.6913

a. Hasil verifikasi empat skenario serangan

Skenario B1 dan B2 sama-sama terdeteksi pada tahap pemeriksaan hash, menunjukkan bahwa SHA-256 sensitif terhadap perubahan sekecil apapun pada konten berkas, baik berupa penambahan *byte* di luar struktur ZIP maupun penyisipan entri baru ke dalam struktur ZIP itu sendiri. Skenario C terdeteksi pada tahap verifikasi *signature* meskipun hash berkas tidak berubah, membuktikan bahwa lapisan *signature* memberikan perlindungan tambahan terhadap kasus di mana penyerang berusaha memalsukan metadata verifikasi tanpa mengubah berkas itu sendiri. Skenario D menunjukkan pentingnya pemeriksaan masa berlaku manifes untuk mencegah penggunaan kembali manifes lama pada konteks yang tidak sesuai.

B. Perbandingan Performa RSA-2048 dan ECDSA

Tabel II menyajikan hasil benchmark dari 20 kali pengulangan untuk masing-masing algoritma terhadap berkas yang sama.

TABLE II.

Metrik	RSA-2048	ECDSA	Rasio
Key Generation (ms)	62.4268	0.4305	0.8842
Signing (ms)	1.0972	0.4507	0.2267
Verification (ms)	0.0621	0.4168	0.2430
Ukuran signature (B)	256	70	0.7716
Ukuran public key (B)	451	174	0.6913

b. Perbandingan performa RSA-2048 dengan ECDSA SECP256K1

Hasil menunjukkan ECDSA menghasilkan ukuran signature 3,7 kali lebih ringkas dan ukuran kunci publik 2,6 kali lebih ringkas dibandingkan RSA-2048, dengan waktu pembuatan kunci 145 kali lebih cepat. Hal ini relevan untuk skenario distribusi manifes berskala besar, mengingat manifes yang lebih ringkas mengurangi overhead jaringan ketika diterbitkan untuk ribuan berkas *mod*. Sebaliknya, RSA-2048 menunjukkan waktu verifikasi yang lebih singkat dibandingkan ECDSA pada implementasi ini, sehingga pemilihan algoritma pada praktiknya perlu mempertimbangkan apakah operasi yang lebih sering dilakukan adalah *signing* atau *verification*.

C. Avalanche Effect pada SHA-256

Pengujian dilakukan dengan membalik satu bit pada sepuluh posisi *byte* acak dari berkas examplemod_appleskin.jar, lalu mengukur jumlah bit hash SHA-256 yang berubah dari total 256 bit. Hasil pengujian ditunjukkan pada Tabel III.

TABLE III.

Posisi Byte	Bit Berubah	Presentasi
40888	135	52.73%
75470	127	49.61%
49192	126	49.22%
51990	116	45.31%
25995	120	46.88%
9961	121	47.27%
77607	123	48.05%
31830	132	51.56%
13356	138	53.91%
39529	131	51.17%

c. Hasil pengujian *avalanche effect* SHA-256

Rata-rata perubahan bit dari sepuluh pengujian tercatat sebesar 126,90 bit dari 256 bit, atau setara 49,57%. Nilai ini sangat mendekati nilai ideal 50% yang diharapkan dari fungsi hash kriptografis yang baik, mengonfirmasi bahwa perubahan satu bit pada berkas .jar menghasilkan perubahan menyeluruh pada nilai hash sehingga modifikasi sekecil apapun tidak dapat lolos dari pemeriksaan integritas.

D. Waktu Verifikasi Terhadap Ukuran Berkas

Tabel IV menunjukkan rata-rata waktu verifikasi *end-to-end* dari 10 kali pengulangan untuk setiap berkas pada dataset.

TABLE IV.

Mod	Ukuran(kb)	Waktu Verifikasi (ms)
appleskin	79.02	0.7174
jei	49.58	0.6041
jade	197.09	1.0010
optifine	492.29	1.8037
create	1968.26	5.8374

d. Waktu verifikasi berdasarkan ukuran berkas

Waktu verifikasi meningkat secara mendekati linear terhadap ukuran berkas, sebagaimana diharapkan karena komponen hashing mendominasi waktu komputasi untuk berkas besar. Bahkan untuk berkas berukuran hampir 2 MB, waktu verifikasi tetap berada di bawah 6 milidetik, menunjukkan bahwa sistem ini layak digunakan dalam praktik tanpa menimbulkan keterlambatan yang berarti pada proses instalasi mod.

E. Perbandingan dengan Sistem yang Sudah Ada

TABLE V.

Fitur	Sistem Ini	Forge Sign.	Mod Auth.	Modrinth
Hash verification	Ya (SHA-256)	Tidak konsisten	Ya	Ya (SHA-512)
Digital signature	Ya (ECDSA)	Ya (tidak efektif)	Tidak	Tidak
Verifikasi sumber	Ya	Tidak	Tidak	Tidak
Anti-replay	Ya	Tidak	Tidak	Tidak

e. Perbandingan fitur dengan sistem yang sudah ada

F. Keterbatasan Penelitian

Pada penelitian ini terdapat beberapa keterbatasan. Pertama, sistem berjalan sebagai *standalone tool* dan belum terintegrasi langsung ke dalam *mod loader* Forge atau Fabric, sehingga masih memerlukan langkah verifikasi manual dari pengguna. Kedua, distribusi kunci publik pada penelitian ini masih dilakukan secara langsung melalui manifes tanpa infrastruktur kunci publik terpusat, sehingga rentan terhadap skenario di mana penyerang mendistribusikan kunci publik palsunya sendiri bersamaan dengan berkas yang telah dimodifikasi. Ketiga, sistem ini berfokus pada deteksi modifikasi pasca-publikasi dan tidak melindungi dari skenario *mod* berbahaya yang sejak awal dipublikasikan oleh pengembang yang tidak bertanggung jawab.

VI. KESIMPULAN

Penelitian ini merancang dan mengimplementasikan sistem verifikasi integritas dua lapis untuk *mod* Minecraft, menggabungkan fungsi hash SHA-256 untuk deteksi perubahan konten dengan tanda tangan digital ECDSA untuk verifikasi keaslian sumber. Sistem terbukti mendeteksi seluruh empat skenario serangan yang diuji yaitu injeksi *byte*, injeksi kelas Java, pemalsuan *signature*, dan *replay attack* dengan tingkat deteksi 100% pada eksperimen yang dilakukan.

Temuan kuantitatif menunjukkan ECDSA secp256k1 menghasilkan signature 3,7 kali lebih ringkas dan proses pembuatan kunci 145 kali lebih cepat dibandingkan RSA-2048, menjadikannya pilihan yang lebih sesuai untuk skenario distribusi manifes berskala besar. Pengujian *avalanche effect* mengonfirmasi sifat teoritis SHA-256 dengan rata-rata perubahan 49,57% bit hash akibat modifikasi satu bit pada berkas, dan waktu verifikasi end-to-end tetap di bawah 6 milidetik bahkan untuk berkas berukuran hampir 2 MB.

Implikasi praktis dari penelitian ini mencakup rekomendasi untuk pengembang platform distribusi *mod* dan *mod loader*. Pertama, integrasi verifikasi ini ke dalam proses instalasi *mod loader* akan memberikan perlindungan otomatis tanpa memerlukan tindakan tambahan dari pengguna. Kedua, platform distribusi seperti CurseForge dan Modrinth dapat mengadopsi format manifes terstandarisasi yang mencakup hash dan tanda tangan digital, memungkinkan verifikasi di sisi pengguna tanpa perubahan infrastruktur yang besar. Ketiga, pengembang *mod* didorong untuk menandatangani karya mereka menggunakan kunci privat yang aman, dengan mekanisme pemulihan kunci yang hilang atau kompromi. Rekomendasi tambahan mencakup penggunaan mekanisme pembaruan otomatis untuk daftar kunci publik yang dicabut, serta periode kadaluarsa manifes yang masuk akal untuk menyeimbangkan keamanan dengan kenyamanan pengguna. Dengan adopsi yang meluas, sistem verifikasi seperti yang diusulkan dalam penelitian ini dapat secara signifikan mengurangi risiko serangan rantai pasok pada ekosistem mod Minecraft.

Pengembangan lanjutan yang disarankan meliputi integrasi langsung sistem ini pada mod loader Forge atau Fabric, pembangunan infrastruktur kunci publik terpusat untuk distribusi kunci yang lebih terpercaya.

REFERENCES

[1] Mojang Studios, "Minecraft Annual 2026," April 2025.

[2] Host Havoc, "Minecraft User and Revenue Statistics," 2026.

[3] CurseForge, "CurseForge — Mods & Addons Leading Community," 2026.

[4] Sportskeeda, "Minecraft mods reach 100 billion downloads on CurseForge, marking a huge milestone," Feb. 2026. [Online].

[5] Prism Launcher, "[MALWARE WARNING] 'fractureiser' malware in many popular Minecraft mods and modpacks," Jun. 2023. [Online].

[6] fractureiser Investigation Team, "fractureiser malware — User Documentation," GitHub Repository, Jun. 2023.

[7] Kaspersky, "Fractureiser attacks Minecraft players," Kaspersky Official Blog, Jun. 2023. [Online].

[8] Modrinth, "Modrinth modpack format (.mrpack) specification," Modrinth

[9] Darkhax, "Jar Signing doesn't check the integrity of the actual files," MinecraftForge GitHub Issue #7500, Nov. 2020. [Online].

[10] National Institute of Standards and Technology, “Secure Hash Standard (SHS),” FIPS PUB 180-4, Aug. 2015.

[11] National Institute of Standards and Technology, “Digital Signature Standard (DSS),” FIPS PUB 186-5, Feb. 2023.

[12] D. Johnson, A. Menezes, and S. Vanstone, “The Elliptic Curve Digital Signature Algorithm (ECDSA),” International Journal of Information Security, vol. 1, pp. 36–63, 2001.

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, 19 Juni 2026

Alma Felicia Vielrizki 18223112