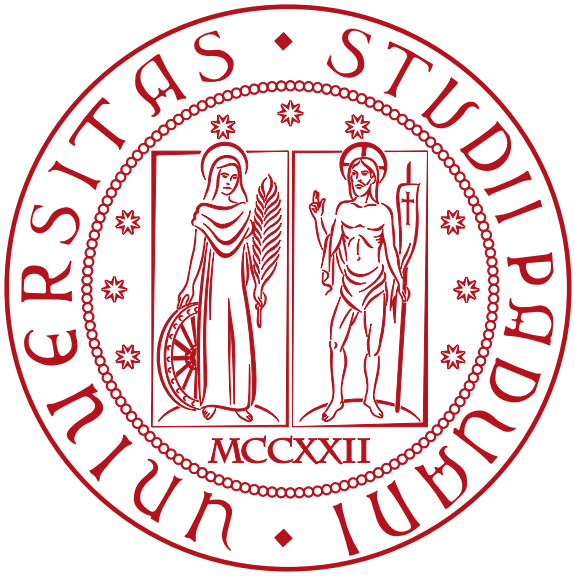
Università degli studi di Padova

Dipartimento di Matematica "Tullio Levi-Civita"

Corso di Laurea in Informatica



Analisi di sicurezza e integrità crittografica di un sistema di versionamento distribuito

Tesi di laurea triennale

Relatore

Prof. Tullio Vardanega

Laureando

Michele Stevanin

Matricola 2101741

ANNO ACCADEMICO 2025-2026

«Chi non conosce la storia è condannato a ripeterla,
ma chi la conosce è condannato a guardare gli altri che la ripetono.»

— Primo Levi

Sommario

Questa relazione descrive il lavoro svolto durante lo stage curricolare presso Zucchetti S.p.A., della durata di trecentoquattro ore, sotto la supervisione del tutor aziendale Gregorio Piccoli e del tutor accademico Prof. Tullio Vardanega.

Lo stage ha riguardato l'analisi della sicurezza di *RVC*$_G$, un sistema di versionamento distribuito sviluppato internamente da Zucchetti S.p.A. come alternativa a *Git*$_G$. A differenza dei sistemi tradizionali, *RVC*$_G$ non richiede un server centrale per la verifica di autenticità dei *commit*$_G$: i *commit*$_G$ vengono distribuiti come archivi firmati, navigabili direttamente tramite filesystem. L'integrità dei contenuti è garantita attraverso la verifica crittografica degli *hash*$_G$ di ogni *commit*$_G$, con l'obiettivo di permettere a qualsiasi utente di accertare autonomamente l'autenticità della *repository*$_G$ ricevuta, indipendentemente dalla fonte.

Il lavoro ha previsto lo studio delle tecnologie crittografiche alla base del sistema — chiavi *SSH*$_G$, firma-digitale e strumenti di cifratura — seguito dalla definizione di un modello di sicurezza formale che identifica i requisiti di un sistema di versionamento distribuito sicuro, dalla progettazione di una gerarchia di fiducia e di livelli di sicurezza configurabili, e dalla simulazione di scenari di attacco con diversi livelli di accesso. Infine, sono stati implementati miglioramenti nel codice sorgente di *RVC*$_G$ per colmare le vulnerabilità individuate e avvicinare il sistema al modello di sicurezza proposto.

Organizzazione del testo

Il primo capitolo presenta l'azienda ospitante, introduce il progetto *RVC*$_G$ e illustra le motivazioni che mi hanno portato a scegliere questo stage, con gli obiettivi definiti, la pianificazione e l'analisi dei rischi;

Il secondo capitolo descrive l'organizzazione del lavoro durante il tirocinio, l'ambiente di sviluppo, gli strumenti utilizzati e l'approccio metodologico seguito;

Il terzo capitolo illustra le tecnologie e i concetti teorici alla base del progetto, dalla crittografia-asimmetrica e *SSH*$_G$ fino all'architettura di *RVC*$_G$ e al confronto con *Git*$_G$

Il quarto capitolo definisce il modello di sicurezza che un sistema di versionamento distribuito moderno dovrebbe soddisfare, analizzando le proprietà fondamentali richieste, la distinzione tra *commit*$_G$ ordinari e amministrativi, la gerarchia di fiducia, i livelli di sicurezza configurabili, la gestione del ciclo di vita delle identità e il meccanismo di Redazione Trasparente, concludendo con un'analisi del divario rispetto allo stato iniziale di *RVC*$_G$

Il quinto capitolo presenta gli scenari di attacco simulati in ambiente controllato, con diversi livelli di accesso — da attaccante esterno fino a compromissione della chiave del capo progetto — e analizza la propagazione degli errori nella catena degli *hash*$_G$

Il sesto capitolo descrive i miglioramenti implementati nel codice sorgente di *RVC*$_G$, tra cui il sistema di configurazione dinamico, la firma *SSH*$_G$ integrata, la verifica dell'integrità della catena e i meccanismi di controllo delle identità autorizzate;

Il settimo capitolo riassume i risultati raggiunti, valuta il grado di soddisfacimento degli obiettivi prefissati e propone una riflessione personale sull'esperienza e sugli sviluppi futuri.

Convenzioni tipografiche

Durante la stesura del testo sono state adottate le seguenti convenzioni tipografiche:

- Gli acronimi, le abbreviazioni e i termini di uso non comune vengono definiti nel glossario, situato alla fine del documento (p. 61);
- I termini presenti nel glossario sono indicati con la notazione: *RVC*$_G$;
- I termini in lingua straniera non di uso comune o appartenenti al gergo tecnico sono evidenziati in *corsivo*;
- I nomi di funzioni, variabili o comandi sono scritti con carattere `monospaziato`;
- I riferimenti bibliografici sono indicati con il numero identificativo della fonte, es. [1];
- I blocchi di codice sorgente sono rappresentati nel seguente modo:

```cpl
proc ReportInfo(FileManifestPersistent fm, bool files:=false)
  if fm.tag <> nil
    ? '    tag:', fm.tag
  end
  if fm.prev <> nil
    ? '   prev:', fm.prev
  end
  if fm.merge <> nil
    ? '  merge:', fm.merge
  end
end
```

Esempio di funzione CPL estratta dal sorgente di *RVC*$_G$.

Indice

Elenco delle Figure

Elenco delle Tabelle

Elenco dei Codici Sorgente

Capitolo 1

Introduzione

In questo capitolo viene descritta l'azienda ospitante, introdotto il progetto e illustrate le motivazioni che hanno portato alla scelta di questo stage.

1.1 L'azienda

Zucchetti S.p.A. è un'azienda italiana con sede principale a Lodi che produce soluzioni software, hardware e servizi per aziende, professionisti e associazioni di categoria. Le origini risalgono al 1977, quando lo studio del commercialista Domenico «Mino» Zucchetti realizzò il primo software in Italia per l'elaborazione automatica delle dichiarazioni dei redditi. L'anno successivo, nel 1978, fu fondata formalmente l'azienda.

Oggi il gruppo conta più di 8.000 dipendenti, di cui 2.000 dedicati alla Ricerca e Sviluppo, e serve oltre 700.000 clienti tramite sedi distribuite in tutta Italia e più di 1.650 partner. A livello internazionale è presente con proprie società in diversi paesi europei e oltreoceano, con una rete di oltre 350 partner in 50 paesi. Zucchetti si posiziona oggi come la prima *software house* in Italia per fatturato.

Lo stage è stato svolto presso la sede di Padova e di Noventa Padovana (PD), sotto la supervisione del tutor aziendale Gregorio Piccoli.

Logo di Zucchetti S.p.A.

1.2 Il progetto

Il progetto ha riguardato l'analisi della sicurezza di *RVC*$_G$, un sistema di versionamento distribuito sviluppato internamente da Zucchetti S.p.A. come alternativa a *Git*$_G$. A differenza dei sistemi tradizionali, *RVC*$_G$ non richiede un server centrale: i *commit* vengono distribuiti come archivi firmati, navigabili direttamente tramite filesystem. L'integrità dei contenuti è garantita attraverso la verifica crittografica degli *hash*$_G$ di ogni *commit*, con l'obiettivo di permettere a qualsiasi utente di accertare autonomamente l'autenticità della *repository* ricevuta, indipendentemente dalla fonte di distribuzione.

L'autenticazione e la firma dei *commit* avvengono tramite chiavi *SSH*$_G$, rendendo ogni modifica crittograficamente attribuibile al suo autore. Questo approccio distingue *RVC*$_G$ non solo per l'architettura distribuita, ma anche per le garanzie di autenticità che offre rispetto ai sistemi di versionamento convenzionali.

La versione di *RVC*$_G$ fornita per lo stage è una versione di sviluppo deliberatamente priva di alcuni meccanismi di sicurezza, con l'obiettivo di permettere uno studio autonomo delle vulnerabilità e la progettazione di soluzioni originali. Questo approccio ha consentito di affrontare il problema della sicurezza senza vincoli architetturali predefiniti, producendo analisi e implementazioni indipendenti.

1.3 Scelta del progetto

Ho scelto questo progetto per l'interesse verso la sicurezza informatica e la crittografia applicata, temi che avevo già incontrato durante il percorso universitario ma che non avevo mai avuto l'occasione di approfondire in un contesto reale. L'analisi di un sistema in uso aziendale, con l'obiettivo di individuare vulnerabilità concrete e proporre miglioramenti, rappresenta un'opportunità difficilmente replicabile in ambito accademico.

La natura del lavoro — che combina studio teorico delle tecnologie crittografiche, progettazione di modelli di sicurezza e simulazione pratica di scenari di attacco — mi ha convinto che fosse il progetto più adatto per concludere il percorso triennale con un contributo originale e tangibile.

1.4 Obiettivi dello stage

Gli obiettivi dello stage sono stati definiti in accordo con il tutor aziendale e classificati secondo tre livelli di priorità:

- **Obbligatori** (O): obiettivi il cui soddisfacimento è vincolante per la riuscita del progetto;
- **Desiderabili** (D): obiettivi non strettamente necessari ma dal riconoscibile valore aggiunto;
- **Facoltativi** (F): obiettivi che rappresentano un ulteriore contributo non competitivo.

Codice	Descrizione
O01	Studio delle tecnologie *SSH*$_G$, *AGE*$_G$ e *RVC*$_G$
O02	Analisi del sistema *RVC*$_G$: architettura, formato dei *commit*$_G$ e individuazione delle vulnerabilità
O03	Definizione di un modello di sicurezza formale per sistemi di versionamento distribuito, con requisiti classificati per priorità e confronto con lo stato iniziale di *RVC*$_G$
O04	Valutazione del grado di soddisfacimento del modello di sicurezza proposto a seguito degli interventi migliorativi implementati
O05	Produzione della documentazione tecnica e della relazione finale
D01	Progettazione della gerarchia di fiducia e dei livelli di sicurezza configurabili per *repository*$_G$ distribuite
D02	Simulazione di scenari di attacco con diversi livelli di accesso e analisi delle vulnerabilità individuate
D03	Implementazione dei miglioramenti prioritari nel codice sorgente *CPL*$_G$ di *RVC*$_G$
F01	Progettazione del meccanismo di Redazione Trasparente per la gestione di contenuto illegale o sensibile nella *repository*$_G$
F02	Analisi delle possibilità di adozione di una struttura monorepo o polirepo in *RVC*$_G$

Obiettivi dello stage.

1.5 Pianificazione

Il lavoro è stato organizzato su otto settimane per un totale di 304 ore, suddivise tra studio delle tecnologie, analisi della sicurezza, sviluppo e documentazione.

Settimana	Ore	Attività
1	32	Studio di *SSH*$_G$, crittografia-asimmetrica, firma-digitale e *AGE*$_G$
2	40	Studio di *RVC*$_G$: architettura, formato dei *commit*$_G$
3	40	Analisi del codice sorgente *CPL*$_G$ e individuazione delle vulnerabilità
4	40	Simulazione scenari di attacco senza credenziali e con chiave compromessa
5	40	Implementazione miglioramenti: configurazione, firma *SSH*$_G$, verifica integrità
6	32	Progettazione della gerarchia di fiducia, gestione delle identità e verifica della catena completa
7	40	Simulazione attacchi avanzati e progettazione del meccanismo di Redazione Trasparente
8	40	Completamento e revisione della relazione finale

Pianificazione del periodo di stage.

1.6 Analisi dei rischi

Prima dell'avvio del progetto è stata condotta un'analisi preventiva dei rischi, al fine di individuare le possibili criticità e predisporre le opportune contromisure.

Descrizione	**Contromisura**	**Probabilità** **Impatto**
Codice sorgente di *RVC*$_G$ non disponibile nella fase iniziale, con impossibilità di analisi *white-box*$_G$	Analisi *black-box*$_G$ tramite osservazione del comportamento esterno e dei file prodotti	Alta Alto
Linguaggio *CPL*$_G$ proprietario senza documentazione pubblica, con curva di apprendimento elevata	Studio della documentazione interna fornita dall'azienda e confronto diretto col tutor	Alta Medio
Comportamenti silenziosi del sistema in caso di errore, che rendono difficile il debug	Aggiunta sistematica di istruzioni di debug nel codice sorgente durante l'analisi	Media Alto
Vulnerabilità individuate già risolte nella versione interna, rendendo il lavoro ridondante	Verifica periodica col tutor aziendale sull'allineamento tra la versione di test e quella interna	Media Medio

Analisi preventiva dei rischi.

6.

Capitolo 2

Descrizione dello stage

In questo capitolo viene descritta l'organizzazione del lavoro durante il tirocinio, l'ambiente di sviluppo utilizzato, gli strumenti adottati e l'approccio metodologico seguito.

2.1 Organizzazione del lavoro

Lo stage si è svolto in presenza presso le sedi di Zucchetti S.p.A., con rare eccezioni in modalità *smart working* nei periodi in cui l'azienda era impegnata in un trasferimento di sede. La prima settimana si è tenuta presso gli uffici di via Giovanni Cittadella a Padova; le settimane successive presso la sede principale di Noventa Padovana (PD), dove ho trascorso la maggior parte del tirocinio.

L'orario di lavoro era strutturato su due fasce giornaliere: dalle 9:00 alle 13:00 e dalle 14:00 alle 18:00, per un totale di otto ore al giorno. Questa organizzazione ha permesso di alternare sessioni di studio teorico al mattino con attività pratiche nel pomeriggio, sfruttando la pausa di mezzogiorno per consolidare i concetti appresi.

2.1.1 Rapporto con i tutor

Il confronto con i tutor ha avuto modalità diverse a seconda del ruolo.

Il **tutor aziendale**, Gregorio Piccoli, era disponibile quasi quotidianamente per chiarimenti, revisioni del lavoro svolto e indicazioni sui passi successivi. Le riunioni non seguivano una cadenza fissa ma avvenivano su richiesta, ogni volta che si presentava un problema tecnico rilevante o si raggiungeva un risultato degno di discussione. Questo approccio ha favorito un dialogo continuo e ha permesso di adattare il percorso in modo flessibile all'avanzamento del lavoro.

Il **tutor accademico**, il Professor Prof. Tullio Vardanega, è stato invece coinvolto con cadenza settimanale tramite scambio di email. Le comunicazioni riguardavano principalmente l'avanzamento dello stage rispetto alle attese previste e la verifica dell'allineamento con gli obiettivi didattici dello stage.

2.2 Ambiente di sviluppo

Per tutta la durata dello stage ho utilizzato un portatile fornito dall'azienda con sistema operativo Windows. La scelta degli strumenti da installare è stata lasciata alla mia discrezione, senza vincoli imposti dall'azienda, il che ha permesso di configurare un ambiente di lavoro ottimale per le esigenze del progetto.

L'accesso ai sistemi aziendali includeva anche i modelli linguistici (*LLM*) disponibili localmente nell'infrastruttura di Zucchetti, utilizzati come supporto durante le fasi di studio e sviluppo.

2.2.1 Strumenti utilizzati

Gli strumenti principali adottati durante lo stage sono stati:

Visual Studio Code come editor di testo principale, utilizzato sia per la scrittura e modifica del codice sorgente *CPL*$_G$ di *RVC*$_G$, sia per la stesura della documentazione in Typst. L'editor è stato configurato con estensioni per la sintassi *CPL*$_G$ e per la compilazione live dei documenti Typst.

GitHub per il versionamento della tesi e dei file di lavoro personali. Il *repository*$_G$ della relazione finale è ospitato su GitHub con pubblicazione automatica del PDF tramite *GitHub Actions* e *GitHub Pages*.

Typst come sistema di composizione tipografica per la stesura della relazione finale. Typst è un'alternativa moderna a LaTeX che permette di produrre documenti PDF di qualità professionale con una sintassi più accessibile.

Terminale Windows (cmd e PowerShell) per l'esecuzione dei comandi *RVC*$_G$, la gestione dei file e le operazioni di debug. La maggior parte delle operazioni con *RVC*$_G$ avviene tramite interfaccia a riga di comando.

PuTTY come client *SSH*$_G$ per Windows, utilizzato per la gestione delle chiavi *SSH*$_G$ tramite il componente `puttygen`.

Ssh-keygen (OpenSSH per Windows) per la generazione delle chiavi *Ed25519*$_G$, la firma crittografica dei file e la verifica delle firme *SSH*$_G$. Questo strumento è centrale nel meccanismo di sicurezza implementato in *RVC*$_G$.

AGE$_G$ come strumento di cifratura moderno, studiato durante lo stage in preparazione alla fase di progettazione delle *repository* cifrate prevista negli obiettivi desiderabili.

2.3 Approccio metodologico

Il percorso di lavoro ha seguito un andamento alternato tra fasi teoriche e fasi pratiche, con transizioni determinate dall'avanzamento della comprensione del sistema e dalla disponibilità del materiale.

Nella **fase iniziale** l'attenzione era rivolta allo studio delle tecnologie crittografiche — chiavi *SSH*$_G$, firma-digitale, *AGE*$_G$ — attraverso documentazione ufficiale, esempi pratici al terminale e confronto con il tutor aziendale. Parallelamente ho iniziato a esplorare *RVC*$_G$ dall'esterno, analizzandone il comportamento tramite i comandi disponibili e studiando il formato dei file prodotti.

Una volta ottenuto accesso ai **sorgenti *CPL*$_G$**, l'approccio è cambiato: dall'analisi esterna (*black-box*) si è passati all'analisi interna (*white-box*), con lettura sistematica del codice, identificazione delle vulnerabilità e progettazione degli interventi migliorativi. Questa fase ha richiesto anche lo studio del linguaggio *CPL*$_G$, per il quale non esiste documentazione pubblica — la comprensione è avvenuta tramite lettura del codice esistente e consultazione della documentazione interna fornita dall'azienda.

La **fase di implementazione** ha proceduto in parallelo con l'analisi: man mano che venivano individuate vulnerabilità o carenze, venivano progettate e implementate le relative soluzioni, testandole su una *repository* di simulazione appositamente creata.

2.3.1 Ambiente di simulazione

Per riprodurre scenari di attacco in modo controllato e reversibile, ho configurato un ambiente di test dedicato composto da:

- Una **directory di lavoro** (`simulazione/`) contenente i file di un progetto fittizio su cui eseguire le operazioni *RVC*$_G$.
- Una ***repository*$_G$ locale** (`repo/`) dove vengono archiviati i *commit*, separata dalla directory di lavoro.
- Un file `.git/repository.info` che collega la directory di lavoro alla *repository*$_G$, seguendo la convenzione di *RVC*$_G$.
- Un file `allowed_signers` nel formato OpenSSH, contenente le chiavi pubbliche degli autori autorizzati, utilizzato manualmente per i test di verifica delle firme tramite `ssh-keygen`.

Questo ambiente ha permesso di simulare scenari realistici — inclusi la manomissione dei file di *commit*, la compromissione delle chiavi *SSH*$_G$ e la verifica della propagazione degli errori nella catena degli *hash*$_G$ — senza rischiare di danneggiare dati reali.

Capitolo 3

Tecnologie e fondamenti teorici

In questo capitolo sono illustrate le tecnologie e i concetti teorici alla base del progetto. Partendo dalla crittografia-asimmetrica, vengono descritti SSH$_G$, AGE$_G$ e i sistemi di controllo versione, fino ad arrivare all'architettura di RVC$_G$.

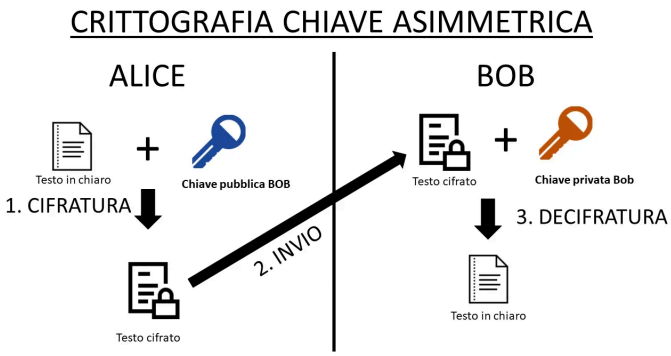
3.1 Crittografia asimmetrica e firma digitale

La crittografia è la disciplina che studia le tecniche per proteggere le informazioni rendendole illeggibili a chi non è autorizzato. Nella sua forma classica, detta *crittografia simmetrica*, mittente e destinatario condividono una stessa chiave segreta per cifrare e decifrare i messaggi. Questo approccio presenta un problema fondamentale: come si trasmette la chiave in modo sicuro prima ancora di poter comunicare in modo sicuro?

La risposta a questo problema arrivò negli anni "70 con la *crittografia asimmetrica*, detta anche *crittografia a chiave pubblica*. L"idea è elegante: ogni utente genera una coppia di chiavi matematicamente collegate tra loro. La **chiave-pubblica** può essere distribuita liberamente a chiunque. La **chiave-privata** deve rimanere segreta e non lasciare mai il dispositivo del proprietario. Le due chiavi sono collegate da una relazione matematica tale per cui ciò che viene cifrato con una può essere decifrato solo con l'altra, ma è computazionalmente impossibile ricavare la chiave-privata a partire da quella pubblica.

Questo meccanismo permette due operazioni fondamentali:

- **Cifratura**: chiunque può cifrare un messaggio usando la chiave-pubblica del destinatario. Solo il destinatario, possedendo la chiave-privata corrispondente, potrà decifrarlo.
- **Firma-digitale**: il mittente cifra un messaggio con la propria chiave-privata. Chiunque, usando la chiave-pubblica del mittente, può verificare che il messaggio provenga effettivamente da lui e non sia stato alterato.



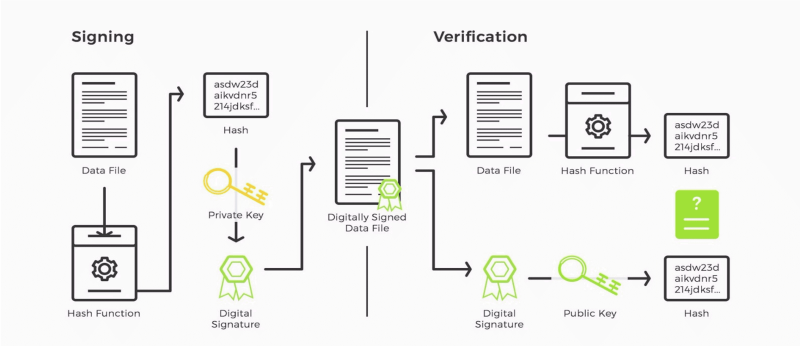
Flusso trasferimento messaggio con uso di chiave pubblica e privata

3.1.1 La firma digitale in dettaglio

La firma-digitale non cifra il messaggio intero — sarebbe inefficiente. Funziona invece in questo modo:

1. Il mittente calcola l'*hash* del messaggio, ovvero una sua impronta digitale di dimensione fissa prodotta da una funzione matematica non invertibile.
2. Il mittente cifra quell'hash con la propria chiave-privata. Il risultato è la firma-digitale.
3. Il destinatario riceve messaggio e firma. Decifra la firma usando la chiave-pubblica del mittente, ottenendo l'hash originale.
4. Il destinatario calcola autonomamente l'hash del messaggio ricevuto e confronta i due valori. Se coincidono, la firma è valida: il messaggio proviene dal mittente dichiarato e non è stato modificato.

La sicurezza di questo meccanismo si basa su due proprietà: la resistenza alle collisioni delle funzioni di $\underline{hash_G}$ (è praticamente impossibile trovare due messaggi diversi con lo stesso $\underline{hash_G}$) e l'impossibilità computazionale di produrre una firma valida senza possedere la chiave-privata.



Flusso firma di un documento con chiave pubblica e privata

3.1.2 Curve ellittiche: Ed25519

Gli algoritmi crittografici moderni si basano su problemi matematici computazionalmente difficili. RSA, il primo algoritmo a chiave-pubblica diffuso, si basa sulla difficoltà di fattorizzare numeri molto grandi. Gli algoritmi moderni basati su *curve ellittiche* offrono lo stesso livello di sicurezza con chiavi molto più corte, risultando più veloci ed efficienti. ***Ed25519***$_G$ è un algoritmo di firma-digitale basato sulla curva ellittica Curve25519. Produce chiavi di 256 bit e firme di 512 bit, con un livello di sicurezza equivalente a RSA con chiavi da 3000 bit. È l'algoritmo raccomandato oggi per la firma *SSH*$_G$ ed è quello utilizzato in questo progetto.

Symmetric Key Size (bits)	RSA and Diffie-Hellman Key Size (bits)	Elliptic Curve Key Size (bits)
80	1024	160
112	2048	224
128	3072	256
192	7680	384
256	15360	521

Comparazione dimensione tra tipologie di chiavi

3.2 SSH e ssh-keygen

SSH$_G$ è un protocollo di rete che permette di stabilire connessioni sicure tra due macchine attraverso una rete non sicura. Nato come sostituto sicuro di Telnet e rsh, *SSH*$_G$ cifra tutto il traffico e autentica sia il server che il client, impedendo attacchi di tipo *man-in-the-middle*.

3.2.1 Autenticazione con chiavi

SSH$_G$ supporta due modalità principali di autenticazione: tramite password e tramite coppia di chiavi. L'autenticazione con chiavi è considerata più sicura e viene raccomandata in tutti i contesti professionali. Il funzionamento è il seguente:

1. L'utente genera una coppia di chiavi con `ssh-keygen`.
2. La chiave-pubblica viene copiata sul server remoto, nella cartella `~/.ssh/authorized_keys`.
3. Al momento della connessione, il server invia una sfida cifrata con la chiave-pubblica dell'utente.
4. Il client dimostra di possedere la chiave-privata corrispondente risolvendo la sfida.
5. La connessione viene stabilita senza che la chiave-privata abbia mai lasciato il client.

3.2.2 ssh-keygen per la firma di file

Oltre alla gestione delle chiavi *SSH*$_G$, `ssh-keygen` offre una funzionalità meno nota ma molto utile: la firma e verifica di file arbitrari. Questo è il meccanismo che *RVC*$_G$ utilizza per firmare crittograficamente i *commit*.

Il comando per firmare un file è:

```
1 ssh-keygen -Y sign -n <namespace> -f <chiave_privata>
  <file>
```
Bash

Il parametro `-n` specifica il *namespace*, una stringa che identifica il contesto d'uso della firma e previene attacchi di riutilizzo — una firma prodotta con *namespace*$_G$ `git` non può essere spacciata per una firma con *namespace*$_G$ `file`.

Il comando per verificare una firma è:

```
1 ssh-keygen -Y verify -n <namespace> -f <allowed_signers> -
  I <identità> -s <firma> < <file>
```
Bash

Il file `allowed_signers` è un elenco di identità autorizzate con le rispettive chiavi pubbliche, nel formato:

```
1 identita@esempio.com ssh-ed25519 AAAA...chiave...
```

Se la firma è valida e l'identità è presente nel file, il comando restituisce `Good "file" signature for <identità>`.

3.3 AGE

AGE$_G$ (*Actually Good Encryption*) è uno strumento moderno per la cifratura di file, progettato con l'obiettivo di essere semplice, sicuro e componibile. A differenza di PGP, che nel corso degli anni ha accumulato una complessità notevole, *AGE*$_G$ offre un'interfaccia minimale con poche opzioni ben definite.

AGE$_G$ supporta tre modalità di cifratura:

- **Chiave-pubblica**: il file viene cifrato con la chiave-pubblica del destinatario e può essere decifrato solo con la corrispondente chiave-privata.
- **Chiave *SSH*$_G$**: *AGE*$_G$ può utilizzare le stesse chiavi *SSH*$_G$ già esistenti (incluse le chiavi *Ed25519*$_G$) come chiavi di cifratura, senza bisogno di generare nuove chiavi dedicate.
- ***Passphrase***$_G$: il file viene cifrato con una *passphrase*$_G$, usando la funzione di derivazione `scrypt` per proteggersi da attacchi a dizionario.

Un file cifrato con *AGE*$_G$ contiene nell'intestazione le informazioni necessarie per la decifratura (il tipo di chiave usata e la chiave di sessione cifrata), seguite dal contenuto

cifrato con ChaCha20-Poly1305, un algoritmo moderno che garantisce sia riservatezza che integrità.

Nel contesto di questo progetto, *AGE*$_G$ è stato studiato come tecnologia di riferimento per la fase successiva del lavoro, che prevede la progettazione di *repository* cifrate con distribuzione dei permessi di accesso. In questa fase *AGE*$_G$ permette di cifrare il contenuto di una *repository* in modo che solo gli utenti autorizzati — identificati dalle loro chiavi pubbliche — possano decifrarla, senza dover condividere nessuna chiave segreta.

```
age-encryption.org/v1
-> ssh-ed25519 Yu405A MJl34qS7ezoxXy9FQjl5TVe1BLjrz1hYEpxVxaLqFy8
bs84Hjf93u1AyipJZ50oe2NxvEeZk1EHkY1/uDZW5pI
--- RxYyA9D3tPSdSdvlcGsVOp/0oOT9XMWCJ1ffjI7mEcs
‡□□□□Í,<.'›0¹êôx9□(NÎÿ
□P§Sø†UùË□Pµ[□=tî,…¾i'»@IÒþÙ#ëÇÜtlF„ô□¼x□□|J3]Fb“°Juž{§bŽ#□Àg□é¸{Œ□ÍT-□□·□F½9*ÃÒ
```

Esempio di un file cifrato con AGE

3.4 Sistemi di controllo versione

3.4.1 Storia e evoluzione

I sistemi di controllo versione (*Version Control System*, *VCS*$_G$) nascono dalla necessità pratica di tenere traccia delle modifiche al codice sorgente nel tempo. Prima della loro diffusione, era comune affidare il versionamento a convenzioni manuali: copie di file con date nel nome, cartelle numerate, archivi compressi. Questo approccio era fragile, difficile da gestire in team e privo di qualsiasi garanzia di integrità.

Il primo sistema formale di controllo versione ampiamente adottato fu **SCCS** (*Source Code Control System*), sviluppato da Bell Labs nel 1972. SCCS introduceva il concetto di *delta*: invece di salvare ogni versione completa del file, memorizzava solo le differenze rispetto alla versione precedente, riducendo significativamente lo spazio occupato.

RCS (*Revision Control System*), sviluppato nel 1982, migliorò SCCS rendendolo più accessibile e introducendo il concetto di *branching*, ossia la possibilità di sviluppare linee di codice parallele a partire da un punto comune.

Entrambi questi sistemi operavano però a livello di singolo file e su singola macchina. Il passo successivo fu l'introduzione dei sistemi *centralizzati*.

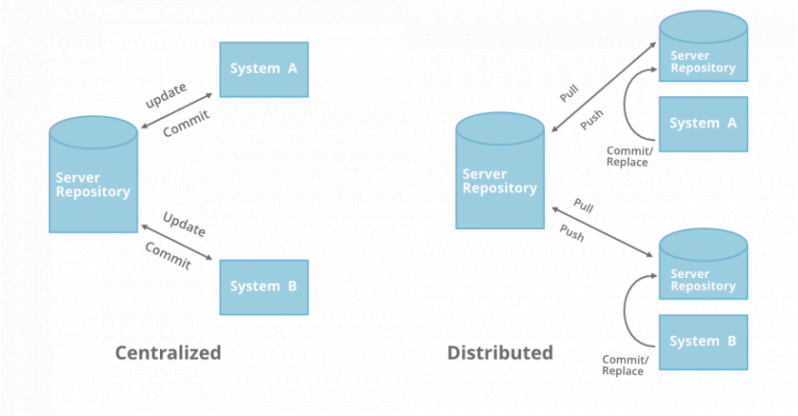
3.4.1.1 Sistemi centralizzati

CVS (*Concurrent Versions System*), negli anni "90, portò il controllo versione in rete. Per la prima volta più sviluppatori potevano lavorare contemporaneamente sullo stesso

progetto, con un server centrale che coordinava le modifiche. Il modello era semplice: il server custodisce l"intera storia del progetto; i client effettuano *checkout* (scaricano una versione) e *commit* (inviano le modifiche).

Subversion (SVN), rilasciato nel 2000, nacque esplicitamente come sostituto migliorato di CVS, correggendo molte delle sue limitazioni tecniche. SVN trattava l'intera struttura del progetto come un'unità atomica: un *commit*$_G$ poteva riguardare più file contemporaneamente, con la garanzia che o tutte le modifiche venivano salvate o nessuna.

Il limite fondamentale dei sistemi centralizzati era però strutturale: la presenza di un singolo punto di fallimento. Se il server non era raggiungibile, nessuno poteva fare *commit*$_G$. Se il server veniva perso, si perdeva l'intera storia del progetto.



Controllo versioni centralizzato vs distribuito

3.4.1.2 Sistemi distribuiti

Git$_G$, sviluppato da Linus Torvalds nel 2005 per gestire lo sviluppo del kernel Linux, rivoluzionò il campo introducendo un modello completamente distribuito. In *Git*$_G$ non esiste un server centrale: ogni sviluppatore possiede una copia completa dell'intera storia del progetto. I *commit*$_G$ avvengono localmente e possono essere sincronizzati con altri *repository*$_G$ in un secondo momento.

Questa architettura offre vantaggi significativi: si può lavorare offline, la storia del progetto è replicata su ogni macchina riducendo il rischio di perdita dei dati, e il *branching* è diventato un'operazione economica e centrale nel flusso di lavoro.

Mercurial, rilasciato nello stesso anno di *Git*$_G$, adottò un approccio simile ma con un'interfaccia considerata più accessibile. Oggi *Git*$_G$ domina il mercato, ma l'ecosistema dei *VCS*$_G$ distribuiti rimane vivo con strumenti come Fossil e Pijul.

3.4.2 Git in dettaglio

In *Git*$_G$ ogni oggetto — *blob* (contenuto di un file), *tree* (struttura di una directory), *commit*, *tag* — è identificato da un *hash*$_G$ SHA del suo contenuto. Questo significa che l'identità di ogni oggetto è determinata dal suo contenuto: due oggetti con lo stesso contenuto hanno lo stesso *hash*$_G$, e qualsiasi modifica produce un *hash*$_G$ diverso.

Un *commit* *Git*$_G$ contiene: il riferimento all'albero dei file in quello stato, il riferimento al *commit*$_G$ precedente (*parent*), i metadati dell'autore e del committer, il messaggio di *commit*$_G$. Questa struttura crea una catena crittograficamente collegata: modificare un *commit*$_G$ invalida tutti i *commit*$_G$ successivi, poiché i loro *hash*$_G$ dipendono dal *hash*$_G$ del precedente.

Git$_G$ supporta la firma crittografica dei *commit*$_G$ tramite GPG o *SSH*$_G$. Tuttavia questa funzionalità è opzionale e deve essere abilitata esplicitamente — non fa parte del flusso di lavoro standard.

3.4.3 RVC a confronto con Git

RVC$_G$ condivide con *Git*$_G$ il modello distribuito ma si differenzia in aspetti fondamentali di architettura e sicurezza.

Caratteristica	***Git***$_G$	***RVC***$_G$
Struttura	*Repository* con oggetti indicizzati	File ZIP navigabili su filesystem
Identificazione *commit*$_G$	*Hash*$_G$ SHA dell'oggetto *commit*$_G$	Timestamp codificato in base36
Firma *commit*$_G$	Opzionale (GPG o *SSH*$_G$)	Integrata nell'architettura
Server centrale	Non richiesto ma comune	Non richiesto per design
Repository$_G$ multiple	Un remote alla volta tipicamente	Più *repository*$_G$ sincronizzate nativamente
Linguaggio	C	*CPL*$_G$

Confronto tra Git e RVC.

La differenza più significativa riguarda la sicurezza: mentre in *Git*$_G$ la firma è un'opzione che il singolo sviluppatore può scegliere di abilitare o meno, in *RVC*$_G$ è parte del modello

stesso. Ogni *commit*$_G$ produce un file `.sig` che contiene gli *hash*$_G$ crittografici del contenuto e della catena precedente, costruendo una struttura analoga a una *blockchain*: modificare un *commit*$_G$ invalida tutti quelli successivi perché l'hash cumulativo non corrisponde più. Il modello di sicurezza proposto nel Sezione 4 estende questa struttura con campi aggiuntivi per supportare la gerarchia di fiducia, i livelli di sicurezza configurabili e la gestione degli incidenti.

3.5 RVC: architettura e funzionamento

3.5.1 Struttura della repository

Una *repository* *RVC*$_G$ è una semplice cartella sul filesystem, senza strutture dati complesse o indici da mantenere. Ogni *commit* è rappresentato da due file:

- Un archivio **ZIP** contenente il *commit* del progetto nella versione corrispondente, incluso il file `.FileManifest` che descrive lo stato di tutti i file tracciati.
- Un file **.sig** contenente i metadati del *commit*$_G$ e la firma *SSH*$_G$, la cui apposizione è opzionale nella versione iniziale del sistema.

I file seguono una convenzione di denominazione che codifica la struttura della storia:

```
1 <progetto>.<id>.<idPrecedente>.{autore}+tag.zip
2 <progetto>.<id>.sig
```

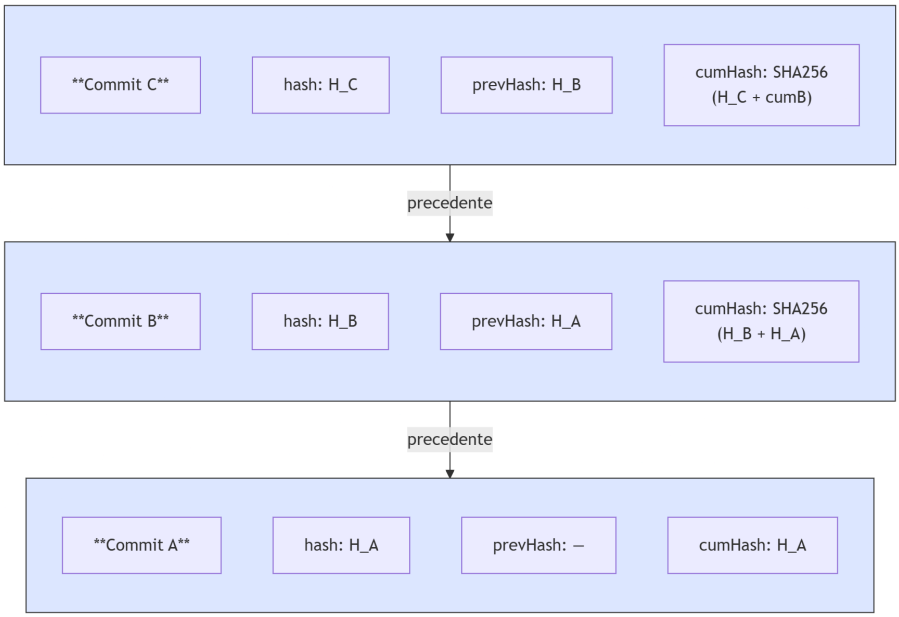
L'`id` è un timestamp codificato in base36 (cifre 0-9 e lettere A-Z), che permette l'ordinamento cronologico dei *commit*$_G$ semplicemente confrontando i nomi dei file. Il riferimento al *commit*$_G$ precedente è incorporato nel nome del file ZIP, rendendo la struttura della storia navigabile senza alcun indice aggiuntivo.

3.5.2 Il file .sig e la blockchain degli hash

Il file `.sig` è il cuore del sistema di sicurezza di *RVC*$_G$. Contiene in formato binario proprietario i seguenti campi:

- `author`: il nome dell'autore del *commit*$_G$
- `comment`: il messaggio del *commit*$_G$
- `fn`: il nome del file ZIP corrispondente
- `id`: l'identificativo del *commit*$_G$
- `prevId`: l'identificativo del *commit*$_G$ precedente
- `hash`: lo SHA256 del file ZIP di questo *commit*$_G$
- `prevHash`: lo SHA256 del file ZIP del *commit*$_G$ precedente
- `cumulativeHash`: lo SHA256 della concatenazione dell'hash attuale con il `cumulativeHash` del *commit*$_G$ precedente

Il `cumulativeHash` è la chiave della sicurezza: ogni *commit*$_G$ incorpora crittograficamente l'intera storia precedente. Verificare che il `cumulativeHash` di un *commit*$_G$ sia corretto significa verificare implicitamente che tutti i *commit*$_G$ precedenti siano integri.



Struttura del file .sig e la catena degli hash cumulativi

Dopo i metadati, il file `.sig` contiene una firma *SSH*$_G$ nel formato standard:

```
1  -----BEGIN SSH SIGNATURE-----
2  ...
3  -----END SSH SIGNATURE-----
```

Questa firma attesta che l'autore dichiarato ha effettivamente prodotto il *commit*$_G$, rendendo ogni modifica crittograficamente attribuibile.

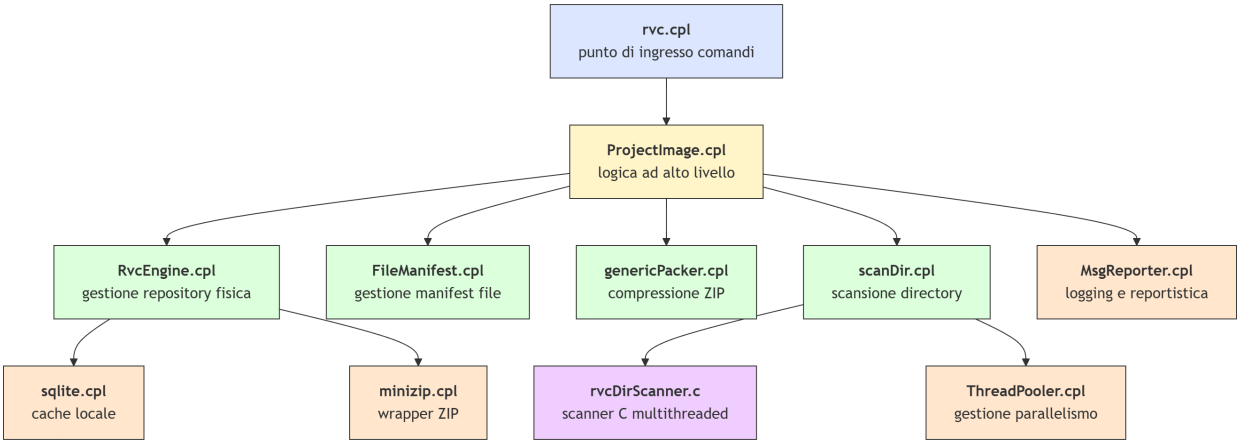
Questi sono i campi presenti nella versione attuale di *RVC*$_G$. Il modello di sicurezza proposto nel Sezione 4 estende questa struttura con campi aggiuntivi — tra cui `security_level`, `allowed_signers`, `branch_status`, `recipients` e `redacted` — necessari per supportare la gerarchia di fiducia e i livelli di sicurezza configurabili.

3.5.3 Il linguaggio CPL

RVC$_G$ è scritto in ***CPL***$_G$ (*CodePainter Language*), un linguaggio proprietario sviluppato da Zucchetti S.p.A. *CPL*$_G$ è un linguaggio interpretato tipizzato staticamente, con supporto a classi, moduli e gestione dei file. Viene eseguito tramite un interprete (`cpl.exe`) che supporta sia interpretazione diretta che compilazione *JIT*$_G$.

Le caratteristiche principali che distinguono *CPL*$_G$ dai linguaggi comuni includono la sintassi di assegnazione con `:=`, l'assenza dell'istruzione `return` esplicita (si usa invece la variabile implicita `result`), la dichiarazione obbligatoria di tutte le variabili in cima alla funzione prima di qualsiasi blocco di codice, e la distinzione tra `func` (funzione con valore di ritorno) e `proc` (procedura senza valore di ritorno).

Il codice sorgente di *RVC*$_G$ è organizzato in diversi moduli *CPL*$_G$, ciascuno con responsabilità ben definite: `ProjectImage.cpl` contiene la logica ad alto livello, `RvcEngine.cpl` gestisce la *repository*$_G$ fisica, `FileManifest.cpl` gestisce il *manifest*$_G$ dei file tracciati.



Struttura dei moduli principali di RVC

3.5.4 Flusso di un commit

Quando un utente esegue `rvc commit`, il sistema esegue i seguenti passi:

1. Legge il file `.FileManifest` nella directory di lavoro per conoscere lo stato corrente del progetto.
2. Scansiona la directory e calcola le differenze rispetto allo stato precedente.
3. Crea un archivio ZIP con i file modificati e il nuovo `.FileManifest`.
4. Calcola lo SHA256 dell'archivio ZIP.
5. Recupera l'hash e il `cumulativeHash` del *commit*$_G$ precedente dal suo file `.sig`.
6. Calcola il nuovo `cumulativeHash` come SHA256 della concatenazione dell'hash attuale con il `cumulativeHash` precedente.
7. Crea il file `.sig` con tutti i metadati.
8. Esegue `ssh-keygen -Y sign` per firmare il file `.sig` e accoda la firma al file.
9. Copia i due file nella *repository*$_G$.

Capitolo 4

Requisiti e modello di sicurezza

In questo capitolo viene definito il modello di sicurezza che un sistema di versionamento distribuito moderno dovrebbe soddisfare. Partendo dalle proprietà fondamentali richieste, vengono analizzate le scelte architetturali, i trade-off e le motivazioni che portano alla definizione di un sistema gerarchico di fiducia, di livelli di sicurezza configurabili e di requisiti formali. Il capitolo si conclude con un'analisi del divario che confronta il modello ideale con lo stato iniziale di RVC$_G$, identificando per ciascun requisito il grado di soddisfacimento nella versione del sistema fornita dall'azienda all'avvio dello stage, prima di qualsiasi intervento migliorativo. Data la centralità di questi aspetti all'interno del progetto, il capitolo risulta volutamente più esteso rispetto agli altri, così da consentire un'analisi dettagliata ed esaustiva delle problematiche considerate e delle soluzioni adottate.

4.1 Sicurezza strutturale nei sistemi di versionamento distribuito

Un sistema di versionamento distribuito è considerato sicuro quando garantisce, per ogni operazione e indipendentemente dal canale di distribuzione, le seguenti proprietà fondamentali.

- **Integrità** — il contenuto di ogni *commit*$_G$ non può essere alterato senza che la modifica sia rilevabile. Questa proprietà è garantita da funzioni di *hash*$_G$ crittografiche: dato un archivio ZIP, la sua impronta SHA256 è univoca e deterministica. Qualsiasi modifica — anche di un singolo byte — produce un *hash*$_G$ completamente diverso. Un sistema che garantisce l'integrità permette a chiunque di verificare che il contenuto ricevuto sia identico a quello prodotto dall'autore, senza dover contattare nessuna fonte autoritativa.
- **Autenticità** — ogni *commit*$_G$ è crittograficamente attribuibile all'autore che lo ha prodotto. L'autenticità è garantita dalla firma-digitale: l'autore firma il *commit*$_G$ con la propria chiave-privata e chiunque può verificare la firma usando la corrispondente chiave-pubblica. Senza autenticità, un sistema può garantire che i dati non siano stati modificati durante il trasferimento, ma non può garantire chi li abbia prodotti originariamente.

- **Non ripudio** — un autore non può negare di aver prodotto un *commit*$_G$ firmato con la propria chiave-privata. Il non ripudio è una conseguenza diretta della firma-digitale: poiché solo il possessore della chiave-privata può produrre una firma valida, la presenza di una firma valida è prova crittografica della paternità. Questa proprietà è rilevante in contesti contrattuali e legali — se un fornitore di software distribuisce codice firmato, non può successivamente sostenere di non averlo prodotto.
- **Ordine verificabile** — la sequenza temporale delle modifiche è verificabile e non manipolabile retroattivamente. Questa proprietà è la più complessa da garantire in un sistema distribuito. Non è sufficiente che ogni *commit*$_G$ sia integro e autentico — è necessario anche che la loro sequenza sia crittograficamente vincolata, in modo che inserire, rimuovere o riordinare *commit*$_G$ sia rilevabile. Questa proprietà è garantita dalla struttura a catena degli *hash*$_G$: ogni *commit*$_G$ incorpora l'hash del precedente, rendendo impossibile modificare un elemento della catena senza invalidare tutti quelli successivi — con l'unica eccezione del meccanismo di Redazione Trasparente descritto nella Sezione 4.7, riservato esclusivamente alla chiave master.

4.1.1 Confronto con i sistemi esistenti

In *Git*$_G$ l'integrità è garantita dalla catena di *hash*$_G$ SHA256 — modificare un *commit*$_G$ invalida tutti quelli successivi. Autenticità e non ripudio sono invece opzionali: la firma crittografica tramite *SSH*$_G$ o GPG deve essere abilitata esplicitamente e non fa parte del flusso di lavoro standard. L'ordine temporale non è verificabile in senso assoluto — *Git*$_G$ non verifica i timestamp e permette la creazione di *commit*$_G$ con date arbitrarie:

```bash
GIT_AUTHOR_DATE="2020-01-01T00:00:00" git commit -m "commit retrodatata"
```

Il risultato è un *commit*$_G$ inserito nella storia con una data arbitraria, indistinguibile da un *commit*$_G$ legittimo. Questa non è una vulnerabilità ma una scelta progettuale documentata — *Git*$_G$ garantisce l'ordine relativo tramite la catena di *hash*$_G$, non l'ordine temporale assoluto.

Disponibilità e correttezza non sono la stessa proprietà: un sistema può essere raggiungibile e simultaneamente produrre risultati errati. Un sistema distribuito che incorpora le garanzie crittografiche descritte sopra non dipende dalla correttezza del server per verificare l'integrità dei propri dati. La distinzione tra disponibilità e correttezza di un sistema centralizzato è emersa concretamente il 23 aprile 2026, quando un bug nella funzionalità di merge queue di GitHub ha causato la generazione silenziosa di *commit*$_G$ errati per 2.092 pull request in 658 *repository*$_G$. Le modifiche precedentemente unite sono state ripristinate dai merge successivi senza alcun avviso. La piattaforma era tecnicamente

operativa durante tutto l'incidente — gli sviluppatori potevano ancora fare push, aprire pull request e fare merge. Il fatto che il merge stesse corrompendo silenziosamente il codice non risultava come incidente nella dashboard di stato.

4.1.2 Applicazione in RVC

RVC$_G$ garantisce l'integrità tramite il `cumulativeHash` — ogni *commit*$_G$ incorpora crittograficamente l'intera storia precedente. La firma *SSH*$_G$ garantisce autenticità e non ripudio. L'ordine temporale presenta la stessa limitazione di *Git*$_G$ — gli identificativi sono timestamp codificati in base36 basati sull'orologio della macchina. Il `cumulativeHash` garantisce però l'ordine **relativo**: anche con timestamp errati o manipolati, la sequenza crittografica è verificabile e non manipolabile senza invalidare l'intera catena. Questa limitazione viene documentata come scelta consapevole — per il contesto d'uso di *RVC*$_G$ la garanzia di ordine relativo è sufficiente e la complessità aggiuntiva di un sistema di timestamping certificato non è giustificata.

Per mitigare parzialmente questa limitazione, il modello proposto prevede che l'identificativo di ogni *commit*$_G$ sia composto da due componenti: il timestamp codificato in base36 per mantenere l'ordinamento cronologico visivo, e una porzione dell'hash del contenuto per garantire l'unicità crittografica. Un identificativo nella forma `0Q6JTD7XVZ_A3F2B1C4` — dove la prima parte è il timestamp e la seconda sono i primi otto caratteri dell'hash SHA256 dell'archivio ZIP — rende praticamente impossibile la collisione intenzionale mantenendo la leggibilità e l'ordinamento per nome file. Questa modifica è identificata come requisito nel modello ideale e discussa nell'analisi del divario nella Sezione 4.8.

4.2 Requisiti di sicurezza formali

A partire dalle proprietà definite nella sezione precedente, è possibile derivare un insieme di requisiti formali che un sistema di versionamento distribuito sicuro deve soddisfare. Ogni requisito è classificato per priorità obbligatorio (O) o desiderabile (D) e verrà ripreso nell'analisi del divario per valutare lo stato iniziale di *RVC*$_G$.

4.2.1 Integrità e ordine verificabile

L'integrità è la proprietà fondamentale di qualsiasi sistema di versionamento — senza di essa non è possibile fidarsi del contenuto ricevuto. In un sistema distribuito questa garanzia non può dipendere dalla fiducia nel canale di distribuzione ma deve essere incorporata nei dati stessi. Ogni *commit*$_G$ deve portare con sé la prova crittografica della propria integrità, e la struttura della catena deve rendere rilevabile qualsiasi tentativo

di modifica retroattiva. Il problema dell'ordine temporale assoluto, come discusso nella sezione precedente, viene trattato come limitazione documentata piuttosto che come requisito da risolvere completamente — il sistema deve però garantire l'ordine relativo e rendere non manipolabili gli identificativi dei *commit*$_G$.

Codice	Descrizione	Priorità
RS01	Ogni *commit*$_G$ deve essere verificabile tramite *hash*$_G$ crittografico del proprio contenuto	O
RS02	La catena degli *hash*$_G$ deve essere verificabile in modo che qualsiasi modifica a un *commit*$_G$ invalidi tutti i successivi	O
RS03	Gli identificativi dei *commit*$_G$ devono essere univoci e non manipolabili tramite timestamp arbitrari	O
RS04	Il sistema deve documentare esplicitamente le proprie limitazioni in termini di ordine temporale assoluto	O

Requisiti di integrità e ordine verificabile.

4.2.2 Autenticità e non ripudio

Garantire l'integrità del contenuto non è sufficiente se non è possibile stabilire chi lo ha prodotto. L'autenticità richiede che ogni *commit*$_G$ sia crittograficamente attribuibile al suo autore tramite firma-digitale. Il non ripudio è una conseguenza diretta di questa scelta — un autore non può negare di aver prodotto un *commit*$_G$ firmato con la propria chiave-privata. La radice di fiducia dell'intera *repository*$_G$ è il primo *commit*$_G$, firmato dall'amministratore con la propria chiave-privata. Poiché l'autenticità di tutti i *commit*$_G$ successivi dipende dalla verificabilità di questa firma, il primo *commit*$_G$ è un requisito di autenticità prima ancora che di integrità — senza di esso non è possibile stabilire da chi provenga la catena di autorizzazioni che legittima ogni singola firma successiva.

Codice	Descrizione	Priorità
RS05	Il primo *commit*$_G$ di ogni *repository*$_G$ deve costituire una radice di fiducia verificabile autonomamente	O
RS06	Il sistema deve supportare e imporre la firma-digitale tramite chiave *SSH*$_G$ per ogni *commit*$_G$ nei progetti configurati con livello di sicurezza maggiore o uguale a 1	O

Requisiti di autenticità e non ripudio.

4.2.3 Gestione delle identità

In un sistema multi-utente la gestione delle identità è il meccanismo che traduce le garanzie crittografiche in un modello organizzativo concreto. Non è sufficiente che le firme siano tecnicamente valide — è necessario che il sistema definisca chi è autorizzato a firmare, come vengono gestiti i cambi di personale e chi ha il potere di modificare questi elenchi. La gerarchia a tre livelli — amministratore, responsabile e dipendente — riflette la struttura organizzativa tipica di un'azienda software e permette di delegare la gestione dei permessi mantenendo un controllo centralizzato sulla radice di fiducia. La revoca deve essere immediata e non richiedere operazioni straordinarie — è sufficiente aggiornare il file di autorizzazione nel *commit*$_G$ successivo.

Codice	Descrizione	Priorità
RS07	Il sistema deve supportare una gerarchia di fiducia a tre livelli: amministratore, responsabile e dipendente	O
RS08	I permessi di scrittura devono essere configurabili per progetto tramite un file di autorizzazione versionato	O
RS09	La revoca di un'identità deve essere efficace dal *commit*$_G$ successivo alla modifica del file di autorizzazione	O
RS10	La successione di un responsabile deve essere gestita esclusivamente dall'amministratore del sistema	D

Requisiti di gestione delle identità.

4.2.4 Sicurezza configurabile

Non tutti i progetti richiedono lo stesso livello di protezione. Un prototipo interno ha esigenze diverse da un modulo che gestisce dati sensibili di un cliente. Imporre lo stesso livello di sicurezza a tutti i progetti sarebbe eccessivamente restrittivo per alcuni e insufficiente per altri. Il modello proposto prevede livelli di sicurezza configurabili per progetto, con il vincolo che il livello non possa essere abbassato nel tempo — una scelta che previene attacchi che cercano di degradare le garanzie di sicurezza di un progetto già avviato. Per i progetti che richiedono la massima riservatezza, il contenuto dei *commit*$_G$ può essere cifrato con *AGE*$_G$, rendendo il codice leggibile solo agli utenti autorizzati.

Codice	Descrizione	Priorità
RS11	Il sistema deve supportare livelli di sicurezza configurabili per progetto, non abbassabili nel tempo	D
RS12	Il contenuto dei *commit*$_G$ deve poter essere cifrato con *AGE*$_G$ per progetti riservati	D

Requisiti di sicurezza configurabile.

4.2.5 Gestione dei branch

I *branch*$_G$ sono uno strumento fondamentale nello sviluppo software parallelo, ma introducono scenari di sicurezza che vanno gestiti esplicitamente. Un *branch*$_G$ può diventare inutile al termine di una funzionalità, oppure può risultare compromesso a seguito di un *commit*$_G$ fraudolento o non autorizzato. In entrambi i casi il sistema deve fornire un meccanismo formale per dichiarare lo stato del *branch*$_G$, senza cancellare la storia — che rimane immutabile e verificabile, salvo il meccanismo di Redazione Trasparente descritto nella Sezione 4.7 — ma aggiungendo un commit firmato che ne attesti la chiusura o la compromissione. Questo approccio mantiene la tracciabilità completa degli eventi, inclusa la prova della compromissione stessa.

Codice	Descrizione	Priorità
RS13	I *branch*$_G$ compromessi devono poter essere chiusi con un *commit*$_G$ firmato che ne attesti la compromissione	D

Requisiti di gestione dei branch.

I requisiti obbligatori definiscono le proprietà minime senza le quali il sistema non può essere considerato sicuro per il contesto d'uso descritto. I requisiti desiderabili estendono il modello con funzionalità che aumentano significativamente il livello di sicurezza, ma la cui assenza non compromette le garanzie fondamentali.

I requisiti RS01, RS02 e RS03 corrispondono alla proprietà di integrità e alla gestione dell'ordine verificabile. RS05 e RS06 garantiscono le proprietà di autenticità e non ripudio. RS07, RS08, RS09 e RS10 definiscono il modello di gestione delle identità e dei permessi. RS11 e RS12 estendono il modello con funzionalità di sicurezza configurabile. RS04 e RS13 affrontano rispettivamente la documentazione delle limitazioni note e la gestione degli incidenti sui *branch*$_G$.

4.3 Gerarchia di fiducia

In un sistema di versionamento distribuito la fiducia non può essere delegata a un server centrale — deve essere incorporata nella struttura stessa dei dati. Il modello proposto definisce una gerarchia a tre livelli operativi più un livello esterno di sola lettura, ciascuno con responsabilità e permessi ben definiti. La gerarchia è asimmetrica: ogni livello superiore può esercitare i poteri del livello inferiore, ma non viceversa.

4.3.1 Commit ordinari e commit amministrativi

Il modello distingue due categorie di *commit*$_G$ in base al contenuto dello ZIP.

Commit$_G$ **ordinario** — crea o modifica esclusivamente file del progetto, senza toccare nessun file speciale. Può essere firmato da qualsiasi dipendente presente in `allowed_signers`.

Commit$_G$ **amministrativo** — crea, modifica o elimina almeno uno dei seguenti file speciali: `allowed_Dipendenti`, `.rvc_policy` o `.rvc_branch_status`. Tutti i *commit*$_G$ del progetto `_rvc_root` sono amministrativi per definizione.

La verifica dei *commit*$_G$ amministrativi avviene prima che il *commit*$_G$ venga prodotto, ed è il punto cruciale per la segregazione dei permessi tra progetti diversi. Il motore confronta il contenuto dello ZIP con quello del *commit*$_G$ precedente e, se rileva modifiche ai file speciali, richiede che la firma appartenga all'amministratore o al responsabile del progetto. Per verificare che il firmatario sia il legittimo responsabile di quel progetto, il motore esegue un controllo congiunto: verifica che la chiave del firmatario appartenga a un responsabile (ovvero sia presente nel file `allowed_Responsabili` di `_rvc_root`) e contemporaneamente che sia già presente all'interno del file `allowed_Dipendenti` del progetto stesso. Questo doppio vincolo impedisce a un responsabile di alterare le policy o i permessi di un progetto assegnato a un altro responsabile. L'amministratore (identificato

invece dalla presenza della sua chiave nel file `allowed_Dipendenti` di `_rvc_root`) è esente dal controllo locale e ha facoltà di produrre *commit*$_G$ amministrativi su qualsiasi progetto. Se la verifica fallisce, il *commit*$_G$ viene rifiutato prima ancora della generazione del file `.sig`.

Nel caso specifico del primo *commit*$_G$ assoluto di un nuovo progetto, non esistendo uno stato precedente, il motore applica un'eccezione logica: accetta la creazione dei file speciali verificando che il firmatario sia un responsabile in `_rvc_root` e che si sia auto-incluso nel file `allowed_Dipendenti` appena creato.

Il primo *commit*$_G$ di qualsiasi progetto è sempre amministrativo — crea `allowed_Dipendenti` e `.rvc_policy` per la prima volta. Di conseguenza solo il responsabile o l'amministratore può inizializzare un progetto.

Questa verifica preventiva ha una conseguenza diretta sulla catena di fiducia: se un *commit*$_G$ esiste ed è crittograficamente valido, chiunque lo verifichi può assumere che il firmatario avesse i permessi necessari al momento della produzione. Non è quindi necessario accedere al contenuto dello ZIP per verificare la legittimità di una modifica ai file speciali — è sufficiente verificare che il commit sia crittograficamente valido e che il firmatario fosse autorizzato secondo `_rvc_root`. Questa proprietà vale per tutti i livelli di sicurezza incluso il livello 4, dove la verifica avviene prima della cifratura sul contenuto in chiaro.

I *commit*$_G$ amministrativi richiedono sempre la verifica dell'identità del firmatario, indipendentemente dal livello di sicurezza del progetto. Ai livelli 0 e 1 non esiste il file `allowed_Dipendenti` e quindi non esiste la figura del responsabile di progetto — l'unico soggetto autorizzato a produrre *commit*$_G$ amministrativi è l'amministratore, che firma con la propria chiave operativa. Questa regola garantisce che i file speciali siano sempre protetti da una firma verificabile anche nei progetti con il livello di sicurezza più basso, dove i *commit*$_G$ ordinari non richiedono firma o non verificano l'identità. Il progetto `_rvc_root` segue sempre le stesse regole indipendentemente dal livello — tutti i suoi *commit*$_G$ sono amministrativi e devono essere firmati dall'amministratore.

4.3.2 Amministratore (CapoProgetto)

L'amministratore è la radice assoluta di fiducia dell'intera *repository*$_G$. Questo ruolo può essere ricoperto da un singolo individuo o da un gruppo direttivo (ad esempio i fondatori o i direttori tecnici). A livello crittografico, il sistema si basa su una netta separazione: esiste un'unica chiave master conservata offline su un dispositivo air-gapped o in una cassaforte fisica, e una o più chiavi operative (una per ciascun amministratore autorizzato) utilizzate per le operazioni quotidiane su macchine connesse. La separazione tra la chiave master e

le chiavi operative limita la finestra di rischio in caso di compromissione: se una chiave operativa viene rubata o esposta, la chiave master interviene per revocarla e nominarne una nuova senza invalidare le altre chiavi operative o perdere il controllo della *repository*$_G$.

La prima operazione alla creazione di una *repository*$_G$ è produrre il file `allowed_Responsabili` — l'elenco delle chiavi pubbliche dei responsabili autorizzati — e firmarlo con la chiave-privata operativa. Questo file e la sua firma costituiscono il primo *commit*$_G$ della *repository*$_G$ e la radice di fiducia da cui deriva tutta la catena di verifica successiva. L'amministratore ha accesso in lettura e scrittura a tutti i progetti della *repository*$_G$ a qualsiasi livello di sicurezza, incluso il livello 4 con contenuto cifrato — la sua chiave pubblica è sempre inclusa tra i destinatari autorizzati.

4.3.3 Responsabile di progetto

Il responsabile gestisce uno o più progetti all'interno della *repository*$_G$. Il ruolo viene assegnato dall'amministratore tramite inclusione nel file `allowed_Responsabili` — non può essere auto-assegnato né delegato a un altro responsabile. Per ogni progetto gestito, il responsabile mantiene il file `allowed_Dipendenti`, che elenca le chiavi pubbliche dei dipendenti autorizzati a committare. Questo file è versionato all'interno dello ZIP di ogni *commit*$_G$ del progetto, fa parte del contenuto hashato e firmato, e la sua storia è completamente tracciabile.

Il responsabile aggiunge o rimuove dipendenti dal proprio progetto in autonomia tramite *commit*$_G$ amministrativi — modificando il file `allowed_Dipendenti` e firmando i *commit*$_G$ con la propria chiave-privata. L'amministratore può sovrascrivere il file `allowed_Dipendenti` di qualsiasi progetto in qualsiasi momento — il suo potere non è vincolato dalla struttura gerarchica. Se un responsabile lascia l'azienda o viene rimosso dal ruolo, l'amministratore nomina un sostituto aggiornando il file `allowed_Responsabili`. Fino alla nomina del sostituto il progetto entra in stato di attesa: i dipendenti esistenti possono continuare a committare, ma non è possibile aggiungere nuovi dipendenti né modificare i permessi esistenti.

Il responsabile può alzare il livello di sicurezza del proprio progetto in qualsiasi momento producendo un *commit*$_G$ firmato che aggiorna il file `.rvc_policy`. Il livello non può essere abbassato — questa operazione viene rifiutata dal motore indipendentemente dall'identità del firmatario. L'amministratore ha lo stesso potere su qualsiasi progetto della *repository*$_G$.

4.3.4 Dipendente

Il dipendente è autorizzato a produrre *commit*$_G$ ordinari su un progetto se e solo se la propria chiave-pubblica è presente nel campo `allowed_signers` del *commit*$_G$ più recente

di quel progetto. I permessi sono definiti per progetto — un dipendente autorizzato su ProgettoA non ha accesso a ProgettoB, anche se entrambi appartengono alla stessa *repository*$_G$. Non esistono permessi per *branch*$_G$: un dipendente autorizzato su un progetto può committare su qualsiasi *branch*$_G$ di quel progetto.

I dipendenti non possono produrre *commit*$_G$ amministrativi — qualsiasi tentativo di creare, modificare o eliminare un file speciale (`allowed_Dipendenti`, `.rvc_policy`, `.rvc_branch_status`) viene rifiutato dal motore indipendentemente dalla presenza della firma. Questa restrizione vale per tutti i livelli di sicurezza.

La revoca è operativa dal *commit*$_G$ successivo alla modifica del file `allowed_Dipendenti`: il dipendente rimosso non può produrre *commit*$_G$ validi sul progetto. I *commit*$_G$ prodotti prima della revoca rimangono validi in quanto firmati da un'identità che era autorizzata al momento della firma — la storia del progetto è immutabile e ogni modifica ai permessi è tracciata nella catena.

4.3.5 Cliente o Guest

Il cliente o guest riceve la *repository*$_G$ e verifica autonomamente autenticità e integrità del contenuto, senza dipendere dall'infrastruttura del produttore. La verifica parte dalla chiave-pubblica operativa dell'amministratore, ottenuta tramite un canale indipendente dalla *repository*$_G$ stessa — ad esempio il sito ufficiale del produttore distribuito tramite HTTPS. Con questa chiave il cliente verifica la firma sul file `allowed_Responsabili` del primo *commit*$_G$ e da lì risale crittograficamente all'intera catena di autorizzazioni e *commit*$_G$.

Il cliente opera in sola lettura e non ha permessi di scrittura sulla *repository*$_G$. Quando riceve un aggiornamento, verifica che i nuovi *commit*$_G$ si colleghino correttamente alla catena già in suo possesso. Nei progetti di livello 4 la chiave-pubblica *AGE*$_G$ del cliente deve essere registrata tra i destinatari autorizzati (`recipients`) per poter decifrare il contenuto. Questo meccanismo sfrutta la separazione architetturale dei permessi: il cliente è autorizzato alla lettura tramite *AGE*$_G$, ma la sua assenza dal file `allowed_Dipendenti` gli impedisce crittograficamente di produrre *commit*$_G$ validi, proteggendo l'integrità dello sviluppo. La verifica della catena e delle firme rimane comunque possibile anche per i non autorizzati senza decifrare, poiché l'hash nel file `.sig` è calcolato sul contenuto cifrato.

4.3.6 Inizializzazione della repository

La creazione di una nuova *repository*$_G$ segue questa procedura:

1. L'amministratore genera la singola coppia di chiavi master con `ssh-keygen -t ed25519` e conserva la chiave-privata master su un dispositivo offline.
2. Gli amministratori generano le proprie coppie di chiavi operative sui rispettivi computer di lavoro. Usando la chiave-privata master, vengono firmate tutte le chiavi pubbliche operative autorizzate.
3. Viene creato il primo *commit*$_G$ del progetto `_rvc_root`. Questo *commit*$_G$ è fondamentale perché inizializza lo stato del motore e deve contenere:
   - Il file `master.pub` (la chiave-pubblica master in chiaro).
   - I file di certificato `.sig` (le firme crittografiche della master sulle chiavi operative).
   - Il file `allowed_Dipendenti` contenente l'elenco di tutte le chiavi pubbliche operative.
   - Il file `allowed_Responsabili` (inizialmente vuoto o con i primi nominati).
4. Questo primo *commit*$_G$ viene firmato con la chiave master stessa, stabilendo l'ancora di fiducia interna al sistema.
5. Infine, la chiave-pubblica master viene pubblicata su un canale indipendente (es. sito web HTTPS). Il cliente verifica che la `master.pub` dentro la *repository*$_G$ coincida con quella sul sito web, validando a cascata l'intera catena.

4.3.7 Compromissione di una chiave operativa

La compromissione di una chiave operativa è lo scenario critico del modello. Si utilizza la chiave master — conservata offline — per revocare esclusivamente la chiave operativa compromessa, lasciando intatte le eventuali altre chiavi operative valide. La procedura è la seguente:

1. Viene recuperato il dispositivo offline contenente la chiave-privata master.
2. Se necessario, il soggetto compromesso genera una nuova coppia di chiavi operativa, la cui parte pubblica viene firmata con la chiave master per creare un nuovo certificato di delega.
3. Viene prodotto uno speciale *commit*$_G$ amministrativo su `_rvc_root` che aggiorna il file `allowed_Dipendenti` (inserendo la nuova chiave e/o rimuovendo la vecchia compromessa) e aggiorna i certificati.
4. Questo *commit*$_G$ di revoca viene firmato eccezionalmente con la **chiave-privata master**.
5. Il motore di *RVC*$_G$ riceve il *commit*$_G$. Poiché la chiave master non è elencata in `allowed_Dipendenti`, il motore procederebbe a rifiutarlo. Tuttavia, prima di emettere il rifiuto definitivo, il motore verifica la firma del *commit*$_G$ contro il file `master.pub` registrato in modo immutabile nel *commit*$_G$ iniziale di `_rvc_root`. Se la firma comba-

cia, il motore riconosce l'autorità suprema della chiave master e accetta il *commit*$_G$ altrimenti lo rifiuta.

4.3.8 Inizializzazione di un progetto

La creazione di un nuovo progetto all'interno di una *repository*$_G$ esistente segue percorsi diversi a seconda del livello di sicurezza scelto. Il primo *commit*$_G$ di qualsiasi progetto è sempre un *commit*$_G$ amministrativo, ma i file da generare e i soggetti autorizzati cambiano in base al contesto.

Per i **progetti ai livelli di sicurezza 2, 3 e 4**, l'inizializzazione è gestita dal responsabile nominato, senza necessità di intervento dell'amministratore. La procedura è la seguente:

1. Il responsabile crea il file `allowed_Dipendenti` con le chiavi pubbliche dei dipendenti autorizzati. In questa fase inaugurale, il motore richiede tassativamente che il responsabile includa la propria chiave-pubblica nel file, in modo da stabilire il vincolo di appartenenza al progetto discusso in precedenza.
2. Il responsabile crea il file `.rvc_policy` con il livello di sicurezza scelto (da 2 a 4) e, per il livello 4, la lista iniziale dei destinatari autorizzati alla decifratura.
3. Il responsabile produce il primo *commit*$_G$ del progetto, firmato con la propria chiave-privata. Il motore accetta il *commit*$_G$ verificando che il firmatario sia in `_rvc_root` e si sia auto-incluso nel nuovo `allowed_Dipendenti`.

Per i **progetti ai livelli di sicurezza 0 e 1**, non esistendo il file `allowed_Dipendenti` né la figura del responsabile, l'inizializzazione può essere eseguita esclusivamente dall'amministratore. L'amministratore produce un primo *commit*$_G$ contenente unicamente il file `.rvc_policy` (che dichiara il livello 0 o 1) e la firma con la propria chiave operativa. Qualsiasi tentativo da parte di un responsabile o di un dipendente di inizializzare un progetto a questi livelli viene rifiutato dal motore.

Il livello di sicurezza definito nel primo *commit*$_G$ non può essere abbassato — può essere alzato in qualsiasi momento tramite un *commit*$_G$ amministrativo firmato dal responsabile o dall'amministratore. Questa scelta elimina la possibilità di degradare le garanzie di sicurezza di un progetto già avviato.

4.3.9 File amministrativi della repository

In *RVC*$_G$ ogni *commit*$_G$ appartiene a un progetto — non esiste il concetto di *commit*$_G$ globale della *repository*$_G$. I file amministrativi `allowed_Responsabili` e la sua firma devono però risiedere nella *repository*$_G$ in modo verificabile e versionato, indipendentemente da qualsiasi progetto specifico.

Il modello proposto risolve questo problema definendo un progetto riservato con nome convenzionale `_rvc_root`, dedicato esclusivamente all'amministrazione della *repository*$_G$. Al suo interno risiede il file `allowed_Responsabili` e la sua firma. Anche `_rvc_root` contiene al suo interno un proprio file `allowed_Dipendenti`. In questo file è presente unicamente la chiave-pubblica operativa dell'amministratore. Solo l'amministratore è quindi autorizzato a committare su questo progetto, seguendo la medesima struttura logica di tutti gli altri.

Il nome `_rvc_root` è riservato per convenzione del modello. Per prevenire conflitti, il motore verifica all'inizializzazione che questo nome non sia già in uso e lo riserva automaticamente — qualsiasi tentativo di creare un progetto con questo nome da parte di un responsabile o dipendente viene rifiutato.

Questa scelta è preferita all'alternativa di file speciali nella radice della *repository*$_G$ perché non richiede modifiche architetturali al motore e mantiene la coerenza del modello — la verifica della radice di fiducia usa esattamente la stessa logica della verifica di qualsiasi altro progetto.

Il progetto `_rvc_root` opera al livello di sicurezza 2 — ogni *commit*$_G$ deve essere firmato dall'amministratore e la firma viene verificata contro il campo `allowed_signers` del `.sig`, che contiene esclusivamente la chiave-pubblica operativa dell'amministratore. Il livello 2 è il minimo che garantisce la verifica dell'identità del firmatario senza richiedere la cifratura del contenuto — `_rvc_root` deve rimanere leggibile da qualsiasi soggetto che voglia verificare la catena di fiducia.

4.3.10 Il file .rvc_policy

Il file `.rvc_policy` definisce le proprietà di sicurezza di un progetto ed è collocato nella radice dello ZIP di ogni *commit*$_G$. È un file speciale — la sua creazione e modifica sono operazioni amministrative riservate al responsabile o all'amministratore. I campi che il file deve contenere sono i seguenti:

- `security_level`: valore intero da 0 a 4 che definisce il livello di sicurezza del progetto. Questo valore viene estratto dal motore e riportato nel campo `security_level` del `.sig` ad ogni *commit*$_G$, in modo che il livello sia verificabile senza accedere allo ZIP. Il livello non può essere abbassato nei *commit*$_G$ successivi.
- `recipients`: lista delle chiavi pubbliche dei destinatari autorizzati alla decifratura. Presente solo nei progetti a livello 4. Definisce la lista iniziale dei destinatari al momento della creazione del progetto e include sia i soggetti autorizzati alla scrittura sia eventuali «Guest» in sola lettura (clienti, auditor). Le modifiche successive avvengono tramite

commit$_G$ amministrativi che aggiornano sia questo campo che l'header *AGE*$_G$ dello ZIP cifrato.

Il file `.rvc_policy` non contiene informazioni sulle identità dei dipendenti — quelle risiedono in `allowed_Dipendenti`. La separazione tra policy di sicurezza e lista delle identità permette di aggiornare i due aspetti indipendentemente, mantenendo in entrambi i casi la tracciabilità completa nella catena dei *commit*$_G$.

4.3.11 Struttura del file .sig nel modello proposto

Il file `.sig` è il punto di contatto tra il contenuto crittografico e il modello di sicurezza. Nel modello proposto la sua struttura estende quella attuale di *RVC*$_G$ con i campi necessari per supportare la gerarchia di fiducia, i livelli di sicurezza configurabili e la gestione dei *branch*$_G$. Il `.sig` è firmato crittograficamente nella sua interezza — qualsiasi modifica a uno qualsiasi dei suoi campi invalida la firma e quindi il *commit*$_G$.

I campi del `.sig` nel modello proposto sono i seguenti:

- `author`: identificativo dell'autore del *commit*$_G$, nella forma definita dal file `allowed_Dipendenti` del progetto. Il formato — email, nome opaco o qualsiasi altra convenzione — è una scelta dell'organizzazione che gestisce la *repository*$_G$.
- `comment`: messaggio descrittivo del *commit*$_G$.
- `fn`: nome del file ZIP corrispondente.
- `id`: identificativo del *commit*$_G$, composto da timestamp in base36 e *hash*$_G$ parziale del contenuto — ad esempio `0Q6JTD7XVZ_A3F2B1C4`.
- `prevId`: identificativo del *commit*$_G$ precedente.
- `hash`: SHA256 del file ZIP di questo *commit*$_G$.
- `prevHash`: SHA256 del file ZIP del *commit*$_G$ precedente.
- `cumulativeHash`: SHA256 della concatenazione dell'hash attuale con il `cumulativeHash` del *commit*$_G$ precedente.
- `security_level`: livello di sicurezza del progetto — da 0 a 4. Estratto dal file `.rvc_policy` dello ZIP e riportato in chiaro nel `.sig` per permettere al motore di applicare le regole corrette senza dover decifrare il contenuto. Questo è necessario in particolare per i progetti a livello 4 — il motore deve sapere che il contenuto è cifrato prima ancora di tentare di leggerlo. La conseguenza è che il livello di sicurezza di un progetto è visibile a chiunque possa leggere il `.sig`, incluso il fatto che un progetto sia riservato. Questo è considerato accettabile perché l'esistenza di un progetto è già visibile dalla struttura dei file nella *repository*$_G$.
- `allowed_signers`: elenco delle chiavi pubbliche *SSH*$_G$ degli identificativi autorizzati a committare al momento di questo *commit*$_G$. Presente solo nei progetti a livello 2, 3

e 4 — estratto dal file `allowed_Dipendenti` dello ZIP prima di qualsiasi cifratura e riportato in chiaro nel `.sig`. Questo garantisce che il campo sia sempre leggibile indipendentemente dal livello di sicurezza del progetto: anche al livello 4, dove lo ZIP viene cifrato dopo l'estrazione, `allowed_signers` rimane in chiaro nel `.sig` e permette la verifica delle firme senza dover decifrare il contenuto. Per il progetto `_rvc_root` contiene esclusivamente la chiave-pubblica operativa dell'amministratore. Nei commit ordinari ai livelli 0 e 1 questo campo è assente. Nei commit amministrativi ai livelli 0 e 1, dove solo l'amministratore può operare, il campo è presente e contiene esclusivamente la chiave-pubblica operativa dell'amministratore.

- `branch_status`: stato corrente del *branch*$_G$ — `active`, `archived` o `compromised`. È presente in ogni *commit*$_G$ e riflette il contenuto del file `.rvc_branch_status` dentro lo ZIP. Il motore legge sempre questo campo direttamente dal `.sig` — senza dover accedere allo ZIP — indipendentemente dal livello di sicurezza del progetto. Questo garantisce che la gestione dei *branch*$_G$ funzioni correttamente anche per i progetti a livello 4 dove lo ZIP è cifrato. Il file `.rvc_branch_status` dentro lo ZIP rimane la fonte di verità completa e può contenere informazioni aggiuntive — motivazione, riferimenti, note — accessibili a chi ha i permessi di lettura.
- `recipients`: elenco delle identità complete dei destinatari autorizzati alla decifratura del contenuto ZIP. Presente solo nei progetti a livello 4. Contiene le chiavi pubbliche *AGE*$_G$ dei destinatari in chiaro — chiunque possa leggere il `.sig` può determinare chi ha accesso al contenuto cifrato. Questa scelta è deliberata: la complessità di meccanismi di oscuramento parziale introduce buchi nella verificabilità senza offrire garanzie di riservatezza robuste.

Dopo questi campi il file `.sig` contiene la firma *SSH*$_G$ dell'autore nel formato standard, assente al livello 0:

```
1  -----BEGIN OPENSSH SIGNATURE-----
2  <contenuto della firma>
3  -----END OPENSSH SIGNATURE-----
```

Esempio di firma *SSH*$_G$

La presenza di `allowed_signers` nel `.sig` in chiaro risolve il problema della verificabilità per qualsiasi livello di sicurezza: il motore di *RVC*$_G$ e qualsiasi terzo possono verificare la firma del *commit*$_G$ e la legittimità del firmatario leggendo esclusivamente il `.sig`, senza dover decifrare il contenuto dello ZIP. Il file `allowed_Dipendenti` dentro lo ZIP rimane la fonte di verità completa — contiene le chiavi pubbliche degli autorizzati con le relative

informazioni ed eventuali dati aggiuntivi ad uso interno — ma non è necessario per la verifica crittografica.

4.3.12 Catena di fiducia tra progetti

Il progetto `_rvc_root` non collega crittograficamente i progetti della *repository*$_G$ tra loro — ogni progetto mantiene una propria catena di *commit*$_G$ indipendente. La funzione di `_rvc_root` è certificare le identità autorizzate, non la struttura dei dati. La fiducia tra i progetti è gerarchica attraverso le identità, non crittografica attraverso la struttura.

La verifica completa di un *commit*$_G$ di un qualsiasi progetto segue questa catena:

1. La firma del *commit*$_G$ viene verificata crittograficamente contro le chiavi pubbliche presenti nel campo `allowed_signers` del `.sig` di quel *commit*$_G$. Il campo `allowed_signers` è fidato perché il motore garantisce che solo il responsabile o l'amministratore possano produrre i *commit*$_G$ amministrativi che lo modificano — qualsiasi altra modifica viene rifiutata prima della creazione del *commit*$_G$. Di conseguenza, se un *commit*$_G$ esiste ed è crittograficamente valido, il suo campo `allowed_signers` riflette una lista di autorizzati approvata da chi ne aveva il potere.
2. La firma del *commit*$_G$ di `_rvc_root` che ha prodotto la versione corrente di `allowed_Responsabili` viene verificata contro la chiave-pubblica operativa dell'amministratore, presente nel campo `allowed_signers` del `.sig` di `_rvc_root`. A differenza degli altri progetti, il campo `allowed_signers` di `_rvc_root` contiene esclusivamente la chiave-pubblica operativa dell'amministratore — i responsabili sono il contenuto di `_rvc_root`, non i suoi firmatari.
3. La legittimità della chiave-pubblica operativa viene verificata tramite il certificato firmato con la chiave master, presente nel primo *commit*$_G$ di `_rvc_root`.
4. La chiave-pubblica master viene verificata tramite il canale indipendente dalla *repository*$_G$.

Questa catena implica un requisito operativo: la verifica completa di qualsiasi *commit*$_G$ richiede la presenza di `_rvc_root` nella *repository*$_G$. Chi riceve la *repository*$_G$ riceve automaticamente tutti i progetti incluso `_rvc_root` — ma un sistema che distribuisce solo i file di un singolo progetto non permette la verifica completa della catena di fiducia.

Questa catena di verifica si applica ai progetti a livello 2 o superiore, dove il campo `allowed_signers` è presente nel `.sig`. Per i progetti a livello 0 e 1 la verifica delle identità attraverso la catena non è applicabile — è una conseguenza diretta del livello di sicurezza scelto, che non prevede né l'autorizzazione esplicita né la lista degli autorizzati. In questi progetti l'unica garanzia verificabile è l'integrità della catena degli *hash*$_G$. Questa

limitazione è documentata come scelta consapevole: i livelli 0 e 1 sono destinati a contesti dove la tracciabilità formale delle identità non è un requisito.

4.3.13 Implicazioni di sicurezza della radice pubblica

Il progetto `_rvc_root` non può essere cifrato — deve essere leggibile da qualsiasi soggetto che voglia verificare la catena di fiducia, incluso il cliente. Questa necessità introduce una tensione strutturale tra verificabilità pubblica e riservatezza organizzativa.

Il contenuto di `_rvc_root` espone le chiavi pubbliche dei responsabili e, implicitamente, la struttura organizzativa dell'azienda. Le chiavi pubbliche non sono segrete per definizione, ma la lista dei responsabili è informazione sensibile — rivela chi ha potere decisionale sulla *repository*$_G$ e rende questi soggetti bersagli privilegiati per attacchi di ingegneria sociale e spear phishing. Analogamente, i nomi dei file ZIP nella *repository*$_G$ rivelano i nomi dei progetti anche quando il contenuto è cifrato a livello 4.

Il modello propone due mitigazioni parziali. La prima riguarda le identità: invece di utilizzare indirizzi email o nomi reali nel file `allowed_Responsabili`, si possono adottare identificativi opachi — ad esempio `resp-001` — riducendo la leggibilità immediata senza eliminare la tracciabilità per chi ha accesso alle informazioni di mappatura. La seconda riguarda i nomi dei progetti: l'uso di identificativi non descrittivi — ad esempio `PRJ-4A2F` invece di `ModuloPagamenti` — impedisce la mappatura immediata del contenuto della *repository*$_G$ a partire dalla struttura dei file.

Queste mitigazioni riducono la superficie di esposizione ma non la eliminano. Il trade-off tra verificabilità pubblica e riservatezza organizzativa è una limitazione strutturale del modello — qualsiasi sistema che permette la verifica autonoma della catena di fiducia deve necessariamente rendere pubblica almeno la radice di quella catena.

4.4 Livelli di sicurezza configurabili

Un sistema di versionamento distribuito utilizzato in contesti aziendali deve servire esigenze di sicurezza eterogenee. Un prototipo interno in fase esplorativa, un modulo di produzione distribuito a clienti e un componente che gestisce dati finanziari hanno requisiti di protezione radicalmente diversi. Imporre un livello di sicurezza uniforme a tutti i progetti è eccessivamente restrittivo per alcuni e insufficiente per altri.

Il modello proposto introduce livelli di sicurezza configurabili per progetto, definiti alla creazione del progetto nel file `.rvc_policy` e non abbassabili nel tempo. Il vincolo di non abbassabilità è una scelta deliberata: un attaccante che compromette l'account di un responsabile non può degradare le garanzie di sicurezza di un progetto già avviato — può

solo alzarle. Ogni livello è un sovrainsieme del precedente: un progetto a livello 3 soddisfa tutti i requisiti dei livelli 0, 1 e 2.

4.4.1 Livello 0 — Aperto

Nessuna firma è richiesta per i *commit*$_G$ ordinari. I *commit*$_G$ amministrativi (come il primo *commit*$_G$ di inizializzazione prodotto dall'amministratore) costituiscono un'eccezione architetturale globale: devono sempre essere firmati e includere il campo `allowed_signers` nel `.sig`. Per i *commit*$_G$ ordinari, invece, chiunque abbia accesso fisico alla *repository*$_G$ può aggiungere *commit*$_G$. Il sistema non impone la verifica automatica dell'identità né dell'integrità, sebbene quest'ultima resti calcolabile matematicamente tramite la catena degli *hash*$_G$. Il file `.sig` per i *commit*$_G$ ordinari al livello 0 contiene solo i campi strutturali — `author`, `comment`, `fn`, `id`, `prevId`, `hash`, `prevHash`, `cumulativeHash`, `security_level` e `branch_status` — ma non il campo `allowed_signers` e non la firma *SSH*$_G$. L'assenza della firma nei *commit*$_G$ ordinari è la caratteristica distintiva del livello 0 e viene rilevata dal motore come indicazione che il progetto opera senza controllo delle identità.

Questo livello è appropriato per prototipi interni in fase esplorativa dove la velocità di sviluppo è prioritaria e la tracciabilità formale non è richiesta. Non fornisce nessuna delle quattro proprietà di sicurezza definite nella sezione precedente — né integrità crittografica delle identità, né autenticità, né non ripudio, né ordine verificabile tramite firme.

4.4.2 Livello 1 — Autenticato

Ogni *commit*$_G$ deve contenere una firma *SSH*$_G$ valida nel formato standard. Il motore verifica che la firma sia presente e crittograficamente corretta — non verifica l'identità del firmatario né se la chiave appartenga a un soggetto autorizzato. Questo livello garantisce autenticità e non ripudio: ogni *commit*$_G$ è attribuibile a chi possiede la chiave-privata corrispondente alla firma, e la presenza della firma è prova crittografica della paternità. Non garantisce autorizzazione — chiunque possieda una chiave *SSH*$_G$ può committare. È appropriato per *repository*$_G$ interne dove tutti i partecipanti sono implicitamente fidati ma si vuole mantenere la tracciabilità delle modifiche.

4.4.3 Livello 2 — Autorizzato

Ogni *commit*$_G$ deve essere firmato da una chiave presente nel file `allowed_Dipendenti` del progetto. Il sistema verifica sia la validità crittografica della firma sia l'appartenenza dell'identità alla lista degli autorizzati. Questo livello garantisce autenticità, non ripudio e autorizzazione — solo i soggetti esplicitamente nominati dal responsabile possono produrre *commit*$_G$ validi. È il livello base raccomandato per qualsiasi progetto in produzione.

4.4.4 Livello 3 — Verificato

Come il livello 2, con l'aggiunta della verifica obbligatoria della catena degli *hash*$_G$ a ogni operazione. Nessuna operazione — lettura, aggiornamento, push — può procedere se la catena risulta corrotta o incompleta. Mentre nei livelli precedenti la verifica della catena è un'operazione esplicita eseguita su richiesta, al livello 3 è una precondizione implicita di qualsiasi interazione con il progetto. Questo livello è appropriato per codice distribuito a clienti esterni, dove la garanzia di integrità deve essere continua e non delegabile a verifiche periodiche manuali.

4.4.5 Livello 4 — Riservato

Come il livello 3, con l'aggiunta della cifratura del contenuto degli archivi ZIP tramite *AGE*$_G$. Solo i soggetti la cui chiave-pubblica è registrata tra i destinatari autorizzati possono decifrare e leggere il contenuto. La verifica della catena e delle firme rimane possibile senza decifrare — l'hash nel file `.sig` è calcolato sul contenuto cifrato, non su quello in chiaro. Questo livello è appropriato per progetti che contengono codice o dati la cui riservatezza è un requisito contrattuale o legale.

Una proprietà fondamentale del Livello 4 è il disaccoppiamento esplicito tra permessi di lettura e permessi di scrittura. Mentre la capacità di produrre *commit*$_G$ è governato dal file `allowed_Dipendenti`, la capacità di leggere i sorgenti è governata dalla lista dei destinatari in `.rvc_policy`. Questa asimmetria permette di definire la figura del «Guest» (ad esempio auditor, tester o clienti): utenti la cui chiave è inclusa tra i destinatari per consentire l'ispezione del codice, ma a cui è inibita la scrittura poiché assenti dall'elenco degli `allowed_Dipendenti`. In un progetto a Livello 4, l'insieme degli utenti autorizzati in scrittura deve essere un sottoinsieme degli utenti autorizzati in lettura.

AGE$_G$ supporta nativamente la cifratura per destinatari multipli: il contenuto è cifrato una volta sola con una chiave di sessione, e la chiave di sessione è cifrata separatamente per ogni destinatario autorizzato. La gestione dei destinatari è descritta in dettaglio nella sezione seguente.

Nei progetti a livello 4 il file `allowed_Dipendenti` risiede all'interno dello ZIP cifrato — è parte del contenuto riservato e non è accessibile a chi non ha i permessi di lettura. La verifica crittografica rimane comunque possibile per qualsiasi osservatore grazie al campo `allowed_signers` presente in chiaro nel `.sig`: questo campo contiene le chiavi pubbliche degli autorizzati al momento del *commit*$_G$ ed è parte del contenuto firmato, quindi la sua integrità è garantita dalla firma stessa. Un osservatore senza permessi di lettura può verificare che il *commit*$_G$ sia firmato da una chiave presente negli `allowed_signers`

del `.sig`, risalire alla gerarchia tramite `_rvc_root` e verificare l'intera catena di fiducia — senza mai dover decifrare il contenuto del progetto.

Il file `allowed_Dipendenti` dentro lo ZIP rimane la fonte di verità completa per chi ha i permessi di lettura — contiene le chiavi pubbliche degli autorizzati con le relative informazioni ed eventuali dati aggiuntivi ad uso interno.

4.4.6 Gestione dei destinatari nel livello 4

Nei progetti a livello 4 il campo `recipients` del `.sig` contiene le identità complete dei destinatari autorizzati alla decifratura — le loro chiavi pubbliche *AGE*$_G$ in chiaro. Chiunque possa leggere il `.sig` può determinare chi ha accesso al contenuto cifrato.

Questa scelta è deliberata. Meccanismi di oscuramento parziale — come fingerprint delle chiavi o lista nascosta — introducono complessità implementativa e buchi nella verificabilità senza offrire garanzie di riservatezza robuste: un osservatore che conosce le chiavi pubbliche dei candidati può sempre risalire alle identità. La riservatezza reale dei destinatari è garantita dalla scelta organizzativa di non distribuire le chiavi pubbliche dei dipendenti, non da meccanismi tecnici nel `.sig`.

La gestione dei destinatari segue le stesse regole degli `allowed_signers`: l'amministratore è sempre incluso tra i destinatari di qualsiasi progetto a livello 4. Aggiungere o rimuovere un destinatario richiede un nuovo *commit*$_G$ amministrativo firmato dal responsabile o dall'amministratore. Questo *commit*$_G$ aggiorna il campo `recipients` nel file `.rvc_policy` e rigenera l'header *AGE*$_G$ del file ZIP cifrato per riflettere la nuova lista dei destinatari — il contenuto cifrato non deve essere nuovamente prodotto, poiché *AGE*$_G$ separa la cifratura del contenuto dalla cifratura delle chiavi di sessione. Il nuovo *commit*$_G$ produce un nuovo file `.sig` con il campo `recipients` aggiornato — i file `.sig` delle commit precedenti rimangono immutati e continuano a riflettere la lista dei destinatari valida al momento della loro produzione.

Livello	Firma richiesta	Signers verificati	Verifica catena	Contenuto cifrato
0 — Aperto	No	No	No	No
1 — Autenticato	Sì	No	No	No
2 — Autorizzato	Sì	Sì	No	No
3 — Verificato	Sì	Sì	Sì	No
4 — Riservato	Sì	Sì	Sì	Sì

Confronto tra i livelli di sicurezza configurabili.

Il livello di sicurezza è definito nel primo *commit*$_G$ del progetto tramite il file `.rvc_policy` e non può essere abbassato nei *commit*$_G$ successivi. Al primo *commit*$_G$ il motore accetta il livello dichiarato nel `.rvc_policy` senza confronto con *commit*$_G$ precedenti — non esistendone. Per ogni *commit*$_G$ successivo il motore verifica che il campo `security_level` del `.sig` sia maggiore o uguale a quello del *commit*$_G$ precedente. Qualsiasi tentativo di abbassare il livello viene rifiutato indipendentemente dall'identità del firmatario.

L'innalzamento del livello di sicurezza può essere effettuato in qualsiasi momento tramite un *commit*$_G$ firmato dal responsabile del progetto o dall'amministratore. Una volta alzato, il nuovo livello diventa il minimo accettabile per tutti i *commit*$_G$ successivi — il sistema non permette di tornare al livello precedente.

4.5 Gestione delle identità e ciclo di vita delle chiavi

La sicurezza di un sistema basato su crittografia-asimmetrica dipende interamente dalla riservatezza delle chiavi private. Una chiave-privata compromessa annulla tutte le garanzie crittografiche — autenticità, non ripudio e autorizzazione diventano prive di significato se un attaccante può produrre firme valide a nome di un utente legittimo. Il modello deve quindi definire procedure esplicite per la gestione ordinaria delle chiavi e per la risposta agli eventi straordinari.

4.5.1 Cambio chiave ordinario

Un dipendente può cambiare la propria coppia di chiavi *SSH*$_G$ in qualsiasi momento — per cambio di dispositivo, per policy aziendale di rotazione periodica o per precauzione in seguito a eventi sospetti. La procedura è la seguente:

1. Il dipendente genera una nuova coppia di chiavi con `ssh-keygen -t ed25519`.

2. Il dipendente comunica la nuova chiave-pubblica al responsabile.
3. Il responsabile aggiorna il file `allowed_Dipendenti` rimuovendo la vecchia chiave-pubblica e aggiungendo la nuova.
4. Il responsabile produce un *commit*$_G$ amministrativo firmato con la modifica al file `allowed_Dipendenti` — il motore verifica che la firma appartenga al responsabile o all'amministratore prima di accettarla.
5. Dal *commit*$_G$ successivo il dipendente firma con la nuova chiave-privata.

I *commit*$_G$ prodotti con la vecchia chiave rimangono validi — erano firmati da un'identità autorizzata al momento della firma e il file `allowed_Dipendenti` di quei *commit*$_G$ conteneva la vecchia chiave-pubblica. La storia del progetto è immutabile e ogni cambio di chiave è tracciato nella catena.

4.5.2 Revoca per compromissione

Se una chiave-privata viene compromessa — rubata, esposta accidentalmente o sospettata tale — la revoca deve avvenire nel minor tempo possibile. La procedura è identica al cambio ordinario ma con priorità immediata: il responsabile aggiorna `allowed_Dipendenti` rimuovendo la chiave compromessa e produce un *commit*$_G$ amministrativo che documenta l'evento.

Esiste una finestra di rischio tra il momento della compromissione e il *commit*$_G$ di revoca: durante questo intervallo un attaccante in possesso della chiave-privata rubata può produrre *commit*$_G$ fraudolenti che risultano validi. La dimensione di questa finestra dipende dalla rapidità con cui la compromissione viene rilevata e comunicata al responsabile. Il sistema non può eliminare questa finestra — è una limitazione strutturale di qualsiasi sistema basato su revoca — ma la minimizza richiedendo che la revoca sia operativa dal *commit*$_G$ successivo senza procedure straordinarie.

I *commit*$_G$ fraudolenti prodotti durante la finestra di rischio rimangono nella storia e risultano validi rispetto al file `allowed_Dipendenti` di quel momento. La loro identificazione richiede un'analisi manuale della storia del progetto nel periodo sospetto.

4.5.3 Revoca offline

In un sistema distribuito non esiste un meccanismo di revoca immediata globale — la revoca è un *commit*$_G$ che deve raggiungere tutti i nodi della rete. Se un dipendente revocato tenta di fare push su una *repository*$_G$ che non ha ancora ricevuto il *commit*$_G$ di revoca, il push viene accettato localmente ma rifiutato al momento della sincronizzazione con una *repository*$_G$ aggiornata.

Questo comportamento è accettabile nel contesto d'uso di *RVC*$_G$: la sincronizzazione avviene tipicamente tramite push esplicito, e il rifiuto è immediato al primo tentativo di push successivo alla revoca. Se il dipendente revocato ha accesso fisico diretto alla *repository*$_G$ — ad esempio può copiare file nella cartella della *repository*$_G$ senza passare per il motore di *RVC*$_G$ — il problema non è più di sicurezza del sistema di versionamento ma di controllo degli accessi fisici all'infrastruttura, che è fuori dallo scope di questo modello.

4.5.4 Successione del responsabile

Se un responsabile lascia l'azienda o viene rimosso dal ruolo, l'amministratore è l'unico soggetto autorizzato a nominare un sostituto. La procedura è la seguente:

1. L'amministratore aggiorna il file `allowed_Responsabili` in `_rvc_root` rimuovendo la chiave del responsabile uscente e aggiungendo quella del nuovo responsabile.
2. L'amministratore firma la modifica con la propria chiave operativa e produce un *commit*$_G$ su `_rvc_root`.
3. Il nuovo responsabile acquisisce immediatamente i permessi sul progetto e può modificare `allowed_Dipendenti`.

Fino alla nomina del sostituto il progetto rimane in stato di attesa: i dipendenti esistenti continuano a committare normalmente, ma nessuna modifica ai permessi è possibile. Questo stato non interrompe lo sviluppo — interrompe solo la gestione amministrativa del progetto. Se il responsabile uscente era l'unico soggetto con la conoscenza operativa del progetto, il problema è organizzativo e non tecnico — il sistema garantisce la continuità dei *commit*$_G$ esistenti ma non può sostituire la conoscenza umana.

4.5.5 Compromissione della chiave dell'amministratore

La compromissione della chiave operativa dell'amministratore è lo scenario più critico del modello. L'amministratore usa la chiave master — conservata offline — per revocare la chiave operativa compromessa e nominarne una nuova. La procedura è la seguente:

1. L'amministratore recupera il dispositivo offline contenente la chiave-privata master.
2. Genera una nuova coppia di chiavi operativa e ne firma la parte pubblica con la chiave master, creando il nuovo certificato di delega.
3. Produce uno speciale *commit*$_G$ amministrativo su `_rvc_root` che aggiorna il file `allowed_Dipendenti` (inserendo la nuova chiave operativa e rimuovendo la vecchia compromessa) e aggiorna il certificato di delega.
4. Questo *commit*$_G$ di revoca viene firmato eccezionalmente con la **chiave-privata master**.

5. Il motore di *RVC*$_G$ riceve il *commit*$_G$. Poiché il firmatario non è presente in `allowed_Dipendenti`, il motore — prima di emettere il rifiuto definitivo — verifica la firma contro il file `master.pub` registrato in modo immutabile nel *commit*$_G$ iniziale di `_rvc_root`. Se la firma combacia, il motore riconosce l'autorità suprema della chiave master, accetta il *commit*$_G$ e rende operativa la nuova delega; altrimenti lo rifiuta.

Chiunque possieda la chiave-pubblica master può verificare la legittimità della nuova chiave operativa e, da lì, ricominciare a verificare la catena di fiducia. I *commit*$_G$ prodotti con la vecchia chiave operativa rimangono validi — erano legittimi al momento della firma. I *commit*$_G$ prodotti da un attaccante con la chiave compromessa durante la finestra di rischio sono identificabili come fraudolenti tramite analisi della storia nel periodo sospetto.

La chiave master non viene mai usata nelle operazioni ordinarie — il suo utilizzo è limitato a questo scenario e alla firma iniziale della chiave operativa. Questa separazione garantisce che la compromissione della chiave operativa, per quanto critica, non comporti la perdita irreversibile del controllo della *repository*$_G$.

4.6 Gestione dei branch

I *branch*$_G$ sono uno strumento fondamentale nello sviluppo software parallelo — permettono di isolare funzionalità, correzioni e sperimentazioni senza interferire con il lavoro principale. In un sistema di versionamento sicuro la gestione dei *branch*$_G$ introduce scenari che vanno affrontati esplicitamente: un *branch*$_G$ può diventare obsoleto, può essere abbandonato o può risultare compromesso a seguito di *commit*$_G$ non autorizzati.

Il principio fondamentale che governa la gestione dei *branch*$_G$ nel modello proposto è l'**immutabilità della storia**: nessun *commit*$_G$ viene mai cancellato o modificato retroattivamente. Qualsiasi intervento su un *branch*$_G$ — archiviazione, chiusura o dichiarazione di compromissione — avviene aggiungendo nuovi *commit*$_G$ che ne attestano lo stato, non rimuovendo quelli esistenti. Questo principio garantisce che la storia del progetto rimanga sempre verificabile nella sua interezza, inclusa la traccia degli eventi straordinari.

4.6.1 Archiviazione di un branch

Un *branch*$_G$ di sviluppo che ha concluso il proprio ciclo di vita — perché la funzionalità è stata integrata nel *branch*$_G$ principale o perché è stata abbandonata — può essere marcato come archiviato tramite un *commit*$_G$ amministrativo. Il responsabile produce un *commit*$_G$ firmato sul *branch*$_G$ contenente il file speciale `.rvc_branch_status` dentro lo ZIP, che dichiara lo stato `archived` e può includere motivazione, riferimenti e note aggiuntive. Lo stato viene estratto da questo file e riportato nel campo `branch_status` del `.sig` — in chiaro e parte del contenuto firmato. Il motore legge esclusivamente il campo

`branch_status` del `.sig` per determinare lo stato del *branch*$_G$, senza dover accedere allo ZIP — questo garantisce il corretto funzionamento a qualsiasi livello di sicurezza, incluso il livello 4 con contenuto cifrato.

I *branch*$_G$ archiviati sono trattati dal motore in una speciale modalità protetta: rifiutano tassativamente qualsiasi nuovo *commit*$_G$ ordinario, cristallizzando di fatto lo stato del codice. È possibile leggerne la storia in modo completo, ma i dipendenti non possono aggiungervi modifiche. L'archiviazione è un'operazione reversibile esclusivamente tramite un *commit*$_G$ amministrativo con la modifica del file `.rvc_branch_status` per impostarlo ad `active`, con il conseguente aggiornamento del campo `branch_status` nel `.sig`. Questo *commit*$_G$ riporta il *branch*$_G$ allo stato operativo normale e ristabilisce i permessi di scrittura standard.

4.6.2 Chiusura di branch compromessi

Un *branch*$_G$ compromesso è un *branch*$_G$ su cui sono state prodotte uno o più *commit*$_G$ fraudolenti o non autorizzati — ad esempio durante la finestra di rischio successiva alla compromissione di una chiave-privata. La gestione di questo scenario segue una procedura in due fasi.

Nella prima fase il *branch*$_G$ compromesso viene dichiarato tale tramite un *commit*$_G$ amministrativo firmato dal responsabile o dall'amministratore. Il file `.rvc_branch_status` dentro lo ZIP dichiara lo stato `compromised`, l'identificativo del primo *commit*$_G$ sospetto, la motivazione e qualsiasi informazione aggiuntiva utile alla gestione dell'incidente. Lo stato `compromised` viene estratto e riportato nel campo `branch_status` del `.sig` — il motore legge questo campo direttamente e blocca immediatamente il *branch*$_G$ senza dover decifrare il contenuto, indipendentemente dal livello di sicurezza del progetto. I commit fraudolenti rimangono nella storia e sono visibili, ma il branch è marcato come non affidabile. Nei casi in cui la presenza stessa del contenuto fraudolento costituisca un problema legale o di sicurezza, è possibile applicare il meccanismo di Redazione Trasparente descritto nella Sezione 4.7 per rendere inaccessibile il contenuto pur mantenendo la catena intatta.

Nella seconda fase viene creato un nuovo *branch*$_G$ pulito a partire dall'ultimo *commit*$_G$ verificato come integro prima della compromissione. Lo sviluppo riprende sul nuovo *branch*$_G$. Il branch compromesso rimane nella repository come evidenza dell'incidente — la sua storia è verificabile e costituisce la prova crittografica di cosa è accaduto e quando. Se necessario, il contenuto dei commit fraudolenti può essere rimosso tramite Redazione Trasparente (Sezione 4.7) mantenendo comunque intatta la traccia forense delle firme e dei timestamp.

Questa procedura garantisce che la risposta a una compromissione non introduca ambiguità nella storia del progetto. Un approccio alternativo — cancellare i commit fraudolenti rompendo la catena crittografica — renderebbe impossibile distinguere una storia ripulita da una storia alterata da un attaccante. Il meccanismo di Redazione Trasparente (Sezione 4.7) offre una terza via: rendere inaccessibile il contenuto fraudolento senza rompere la catena, con una traccia formale firmata dalla chiave master.

4.6.3 Branch e permessi

Come definito nella sezione sulla gerarchia di fiducia, i permessi sono per progetto e non per *branch*$_G$. Un dipendente autorizzato su un progetto può committare su qualsiasi *branch*$_G$ di quel progetto. La disciplina sui *branch*$_G$ — lavorare su *branch*$_G$ di sviluppo e fare merge sul *branch*$_G$ principale solo quando il codice è verificato — è una convenzione operativa del team, non un vincolo tecnico imposto dal sistema.

Questa scelta è deliberata: introdurre permessi per *branch*$_G$ aggiungerebbe complessità gestionale significativa senza un corrispondente aumento delle garanzie di sicurezza. La sicurezza del sistema si basa sull'autenticità dei *commit*$_G$, non sulla restrizione dei *branch*$_G$ su cui è possibile scrivere. Un dipendente che produce un *commit*$_G$ non autorizzato — ad esempio direttamente sul *branch*$_G$ principale saltando il processo di verifica — è comunque identificabile tramite la firma crittografica e la sua azione è permanentemente tracciata nella storia del progetto.

4.7 Redazione Trasparente

In un sistema di versionamento distribuito basato sul principio di immutabilità della storia, emerge una tensione strutturale con i requisiti legali e organizzativi che possono richiedere la rimozione di contenuto specifico dalla *repository*$_G$. Un dipendente infedele o un attaccante che compromette le credenziali di un dipendente può inserire nella *repository*$_G$ contenuto illegale, segreti industriali altrui o dati personali non autorizzati — il cosiddetto *poisoning* della *repository*$_G$. In un sistema centralizzato l'amministratore può riscrivere la storia sul server, ma in un sistema distribuito questa operazione rompe la catena crittografica e lascia tutti i client con una storia divergente senza nessuna traccia formale di cosa è successo e perché.

Il modello proposto introduce un meccanismo denominato **Redazione Trasparente** che permette di rendere inaccessibile il contenuto di uno o più *commit*$_G$ senza rompere la catena crittografica, mantenendo la piena verificabilità dei *commit*$_G$ precedenti e successivi, e lasciando una traccia formale firmata dall'autorità più alta del sistema.

4.7.1 Principio matematico

La catena degli hash in *RVC*$_G$ segue questa struttura:

$$\text{cumulativeHash}(N) = \text{SHA256}(\text{hash}(\text{ZIP}_N) + \text{cumulativeHash}(N-1))$$

La verifica normale controlla che l'hash del file ZIP corrisponda al campo `hash` nel `.sig` e che il `cumulativeHash` sia calcolato correttamente. La Redazione Trasparente introduce una regola di eccezione nel motore: se il `.sig` di un *commit*$_G$ contiene il campo `redacted: true` firmato dalla chiave master, il motore salta la verifica di `hash` e `cumulativeHash` per quel nodo e riprende normalmente dal *commit*$_G$ successivo.

Il punto matematicamente cruciale è che il `cumulativeHash` dei *commit*$_G$ successivi è calcolato sul `cumulativeHash` dichiarato nel `.sig` del *commit*$_G$ redatto — che non cambia. Il `.sig` redatto aggiunge campi nuovi ma non modifica i campi crittografici originali. Di conseguenza i *commit*$_G$ successivi al *commit*$_G$ redatto rimangono validi senza nessuna modifica — la loro catena è intatta.

4.7.2 Struttura del commit redatto

Un *commit*$_G$ redatto mantiene tutti i campi originali del `.sig` invariati e aggiunge i seguenti campi firmati dalla chiave master:

- `redacted`: valore booleano `true` che segnala al motore di applicare la regola di eccezione.
- `redaction_zip_hash`: SHA256 del nuovo file ZIP che sostituisce quello originale. Permette a chiunque di verificare l'integrità del nuovo ZIP senza dover fidarsi del suo contenuto.
- `redaction_authority`: impronta della chiave master che ha autorizzato la redazione.
- `redaction_timestamp`: timestamp della redazione.
- `redaction_legal_ref`: riferimento al procedimento legale o alla motivazione organizzativa che ha giustificato la redazione. Campo libero.
- `redaction_content`: dichiarazione del tipo di contenuto nel nuovo ZIP — `none`, `sanitized`, `encrypted_master` o `encrypted_authority`.
- `redaction_signature`: firma della chiave master su tutti i campi del `.sig` inclusi quelli di redazione. Questa è la firma aggiuntiva che si affianca alla firma originale del dipendente, che rimane presente e verificabile.

La firma originale del dipendente sul *commit*$_G$ non viene rimossa — è prova forense di chi ha prodotto il contenuto originale e quando. La `redaction_signature` della chiave master certifica che l'autorità più alta del sistema ha autorizzato la modifica.

4.7.3 Opzioni per il contenuto del nuovo ZIP

L'amministratore sceglie cosa inserire nel nuovo ZIP in base alla gravità del caso e ai requisiti legali. In tutti i casi il nuovo ZIP contiene sempre il file `REDACTION_NOTICE.json` con i campi: identificativo del *commit*$_G$ originale, hash originale, data della redazione, riferimento legale, tipo di contenuto sostitutivo e contatto per informazioni.

Le opzioni disponibili sono le seguenti.

Nessun contenuto (`redaction_content: none`) — il nuovo ZIP contiene solo il file `REDACTION_NOTICE.json`. Tutto il contenuto originale viene rimosso dalla *repository*$_G$. Questa opzione soddisfa i requisiti legali che richiedono la distruzione del dato — il file originale non esiste più nella *repository*$_G$, e il limite residuo è quello strutturale di qualsiasi sistema distribuito: le copie già scaricate sui dispositivi locali prima della redazione non possono essere raggiunte dal motore.

Contenuto bonificato (`redaction_content: sanitized`) — il nuovo ZIP contiene il contenuto originale con i file problematici rimossi o sostituiti e tutti gli altri file mantenuti. Utile quando il problema è localizzato a un singolo file in un *commit*$_G$ che contiene anche lavoro legittimo che si vuole preservare.

Cifrato per l'amministratore (`redaction_content: encrypted_master`) — il contenuto originale viene cifrato con *AGE*$_G$ usando esclusivamente la chiave master. Solo l'amministratore può recuperarlo accedendo al dispositivo offline. Utile per preservare il contenuto per uso interno o per future indagini mantenendolo inaccessibile a tutti gli altri.

Cifrato per l'autorità (`redaction_content: encrypted_authority`) — il contenuto originale viene cifrato con *AGE*$_G$ usando la chiave pubblica dell'autorità giudiziaria o regolatoria competente, oltre alla chiave master. L'autorità può accedere al contenuto originale per le sue indagini tramite la propria chiave privata. Utile nei casi in cui l'autorità ha bisogno di accedere al contenuto come prova.

4.7.4 Redazione massiva e automazione

Quando il dato problematico è distribuito su più *commit*$_G$ — ad esempio un file rimasto nella *repository*$_G$ per diversi commit consecutivi — la redazione deve essere applicata a tutti i *commit*$_G$ che lo contengono. Il motore supporta tre modalità operative.

Redazione singola — applicata a un singolo *commit*$_G$ identificato dal suo identificativo.

Redazione su range — applicata a tutti i *commit*$_G$ di un branch compresi tra due identificativi specificati.

Redazione di branch intero — applicata a tutti i *commit*$_G$ di un branch dall'inizio alla fine, con marcatura automatica del branch come `compromised`. Questa modalità è

la risposta al caso più grave: un branch il cui contenuto è problematico fin dalla prima commit. Il branch rimane nella *repository*$_G$ come evidenza formale — la sua catena è verificabile, le firme originali dei dipendenti sono leggibili, i timestamp sono certificati — ma nessun contenuto è accessibile. Anche in questo caso la traccia forense è preservata: si sa chi ha lavorato su cosa e quando, anche se il cosa non è più leggibile.

In tutti e tre i casi il motore produce automaticamente il file `REDACTION_NOTICE.json` per ogni *commit*$_G$ redatto, verifica che la firma della chiave master sia presente e valida prima di procedere, e aggiorna il `branch_status` del branch a `compromised` se non lo è già.

4.7.5 Sincronizzazione con i client esistenti

Quando un client che aveva già sincronizzato la *repository*$_G$ si aggiorna, riceve il nuovo `.sig` con `redacted: true` per il *commit*$_G$ interessato. Il motore riconosce questo campo, verifica la `redaction_signature` contro la chiave master, e accetta l'aggiornamento come autoritativo — sovrascrivendo il `.sig` locale.

Il vecchio file ZIP originale rimane sul dispositivo locale ma il motore non lo distribuisce mai ad altri client, poiché il `.sig` ora dichiara il *commit*$_G$ come redatto. Il sistema non può cancellare fisicamente i file già presenti sui dispositivi — questa è una limitazione strutturale di qualsiasi sistema di versionamento distribuito e non è specifica di *RVC*$_G$. Nessun sistema distribuito esistente — inclusi Git e i suoi derivati — può garantire la cancellazione fisica sui client che hanno già sincronizzato.

4.7.6 Garanzie e limitazioni

La Redazione Trasparente offre le seguenti garanzie.

La catena crittografica non si rompe mai — i *commit*$_G$ precedenti e successivi al *commit*$_G$ redatto rimangono verificabili senza modifiche. Nessun nuovo client che riceve la *repository*$_G$ dopo la redazione può accedere al contenuto rimosso. La redazione è visibile a tutti — non esiste nessuna storia nascosta, solo una storia dichiarata come modificata con la firma della massima autorità. Esiste una traccia forense completa: firma originale del dipendente, timestamp originale, firma della chiave master, riferimento legale. L'abuso della funzione è rilevabile — ogni redazione è visibile nella *repository*$_G$ e non può essere nascosta, e ogni uso della chiave master lascia una traccia nel progetto `_rvc_root`.

Le limitazioni residue sono le seguenti.

I file già scaricati sui client locali prima della redazione non possono essere rimossi dal motore — questa è la limitazione strutturale del modello distribuito già discussa. La funzione richiede la chiave master — un attaccante che compromette la chiave master può

produrre redazioni fraudolente. La mitigazione è che ogni redazione è pubblica e rilevabile, e che la chiave master è conservata offline.

4.8 Analisi del divario

Questa sezione confronta il modello di sicurezza definito nelle sezioni precedenti con lo stato iniziale di *RVC*$_G$, identificando per ciascun requisito il grado di soddisfacimento nella versione del sistema disponibile durante lo stage. I requisiti vengono classificati in tre stati: **soddisfatto** (il comportamento nella versione iniziale corrisponde al requisito), **parziale** (esiste una base implementativa ma il requisito non è completamente soddisfatto) e **assente** (il comportamento non è implementato).

Nella versione iniziale di *RVC*$_G$ nessun requisito risulta completamente soddisfatto — i migliori risultati sono parziali, il che riflette la natura deliberatamente ridotta della versione fornita per lo stage.

4.8.1 Integrità e ordine verificabile

L'integrità strutturale è la proprietà meglio supportata dalla versione iniziale di *RVC*$_G$. Il calcolo dell'hash SHA256 del file ZIP e del `cumulativeHash` sono già presenti nel flusso di commit — ogni commit produce un `.sig` con i valori corretti. Tuttavia la verifica di questi valori non è ancora accessibile tramite interfaccia a riga di comando: esistono funzioni helper nel codice sorgente predisposte per la verifica, ma non sono ancora esposte come comandi utilizzabili. Il sistema calcola ma non verifica — la garanzia di integrità è quindi presente nella struttura dati ma non è ancora azionabile dall'utente.

L'identificativo del commit è attualmente composto dal solo timestamp codificato in base36, senza la componente hash del contenuto prevista dal modello. Questo lo rende vulnerabile a collisioni intenzionali basate sulla manipolazione del timestamp. Infine, la limitazione dell'ordine temporale assoluto non è documentata esplicitamente in nessun documento tecnico al di fuori di questa relazione.

Codice	Descrizione	Stato
RS01	Verificabilità tramite hash crittografico del contenuto	Parziale
RS02	Verifica della catena degli hash a ogni operazione	Parziale
RS03	Identificativi univoci e non manipolabili tramite timestamp	Assente
RS04	Documentazione esplicita delle limitazioni sull'ordine temporale	Assente

Analisi del divario — integrità e ordine verificabile.

RS01 è parziale perché l'hash viene calcolato e memorizzato ma non verificato automaticamente. RS02 è parziale perché il `cumulativeHash` esiste nella struttura dati ma la sua verifica non è esposta all'utente. RS03 è assente perché l'identificativo è il solo timestamp. RS04 è assente perché la limitazione non era documentata prima di questa relazione.

4.8.2 Autenticità e non ripudio

La firma *SSH*$_G$ è il punto di maggiore maturità della versione iniziale. Il meccanismo è implementato e funzionante — ogni commit può essere firmato con una chiave *SSH*$_G$ e la firma viene apposta al file `.sig`. Tuttavia la firma è opzionale: deve essere abilitata esplicitamente al momento del commit tramite un parametro della riga di comando. Il modello proposto la rende obbligatoria dal livello di sicurezza 1 in poi.

Non esiste invece nessun concetto di radice di fiducia o prima commit privilegiata. Tutte le commit sono trattate allo stesso modo dal motore — non c'è distinzione tra la commit iniziale che dovrebbe stabilire l'ancora di fiducia e le commit successive. Il progetto `_rvc_root` e l'intera gerarchia di fiducia sono assenti.

Codice	Descrizione	Stato
RS05	Prima commit come radice di fiducia verificabile autonomamente	Assente
RS06	Firma digitale SSH supportata e imposta per i livelli di sicurezza maggiori o uguali a 1	Parziale

Analisi del divario — autenticità e non ripudio.

RS05 è assente perché non esiste il concetto di radice di fiducia né di commit privilegiata. RS06 è parziale perché la firma è implementata e funzionante ma opzionale — il modello richiede che sia imposta automaticamente in base al livello di sicurezza del progetto.

4.8.3 Gestione delle identità

La gestione delle identità è l'area con il divario più ampio tra il modello proposto e la versione iniziale. Non esiste nessuna distinzione tra utenti — amministratore, responsabile e dipendente sono concetti assenti dal motore. Non esiste un file `allowed_Dipendenti` né nessun altro meccanismo per limitare chi può produrre commit validi su un progetto. Di conseguenza non esiste nemmeno il concetto di revoca — non c'è nulla da revocare se non c'è nessuna lista di autorizzati.

La versione fornita per lo stage era deliberatamente sprovvista di questi meccanismi, con l'obiettivo di permettere uno studio autonomo delle vulnerabilità e la progettazione di soluzioni originali. L'assenza di questi controlli è il punto di partenza dell'intero modello proposto in questo capitolo.

Codice	Descrizione	Stato
RS07	Gerarchia di fiducia a tre livelli: amministratore, responsabile e dipendente	Assente
RS08	Permessi di scrittura configurabili per progetto tramite file di autorizzazione versionato	Assente
RS09	Revoca efficace dal commit successivo alla modifica del file di autorizzazione	Assente
RS10	Successione del responsabile gestita esclusivamente dall'amministratore	Assente

Analisi del divario — gestione delle identità.

Tutti i requisiti di questa categoria sono assenti per la ragione strutturale già descritta: senza una lista di autorizzati non è possibile implementare nessuno dei meccanismi che ne dipendono.

4.8.4 Sicurezza configurabile

Non esiste nella versione iniziale nessun concetto di livello di sicurezza per progetto. Tutti i progetti sono trattati in modo identico dal motore — non c'è nessun file `.rvc_policy`

né nessun altro meccanismo per configurare il comportamento del sistema per singolo progetto. La cifratura del contenuto tramite *AGE*$_G$ è completamente assente: la versione fornita per lo stage non includeva nessuna implementazione di cifratura degli archivi ZIP.

Codice	Descrizione	Stato
RS11	Livelli di sicurezza configurabili per progetto, non abbassabili nel tempo	Assente
RS12	Cifratura dei commit con AGE per progetti riservati	Assente

Analisi del divario — sicurezza configurabile.

4.8.5 Gestione dei branch e incidenti

Non esiste nella versione iniziale nessun meccanismo formale per dichiarare lo stato di un branch. I branch sono sequenze di commit senza nessuna metainformazione sullo stato — non esiste il concetto di branch archiviato, compromesso o bloccato. Il file `.rvc_branch_status` e il campo `branch_status` nel `.sig` sono proposte del modello ideale, assenti nell'implementazione corrente. Il meccanismo di Redazione Trasparente, che permette di rendere inaccessibile il contenuto di commit problematici senza rompere la catena crittografica, è anch'esso una proposta originale di questa relazione e non ha nessuna corrispondenza nella versione iniziale.

Codice	Descrizione	Stato
RS13	Branch compromessi chiudibili con commit firmato che ne attesti la compromissione	Assente

Analisi del divario — gestione dei branch e incidenti.

4.8.6 Sintesi

Il confronto tra il modello proposto e lo stato iniziale di *RVC*$_G$ evidenzia un sistema con solide basi crittografiche — la struttura degli hash cumulativi e il meccanismo di firma *SSH*$_G$ sono già presenti e funzionanti — ma privo dei meccanismi organizzativi e di controllo degli accessi necessari per un uso in contesti aziendali con requisiti di sicurezza non banali.

Codice	Requisito	Stato
RS01	Verificabilità tramite hash crittografico	Parziale
RS02	Verifica della catena degli hash	Parziale
RS03	Identificativi univoci e non manipolabili	Assente
RS04	Documentazione limitazioni ordine temporale	Assente
RS05	Radice di fiducia verificabile autonomamente	Assente
RS06	Firma digitale SSH imposta per livello ≥ 1	Parziale
RS07	Gerarchia di fiducia a tre livelli	Assente
RS08	Permessi configurabili per progetto	Assente
RS09	Revoca efficace dal commit successivo	Assente
RS10	Successione del responsabile	Assente
RS11	Livelli di sicurezza configurabili	Assente
RS12	Cifratura AGE per progetti riservati	Assente
RS13	Gestione branch compromessi	Assente

Sintesi dell'analisi del divario.

I requisiti RS01, RS02 e RS06 sono parzialmente soddisfatti — la base implementativa esiste ma non è ancora completa o accessibile all'utente. Tutti gli altri requisiti sono assenti. Questo divario non è una critica al sistema esistente — $\underline{RVC}_G$ è stato progettato con obiettivi diversi e la versione fornita per lo stage era deliberatamente ridotta — ma definisce con precisione l'area di intervento che i capitoli successivi affrontano con la simulazione degli scenari di attacco e l'implementazione dei miglioramenti prioritari.

Capitolo 5

Simulazione scenari di attacco

In questo capitolo effettuo l'analisi degli utenti, sviluppo le user stories e compongo la lista dei requisiti dividendoli per tipologia e necessità.

Capitolo 6

Miglioramenti implementati

In questo capitolo effettuo l'analisi degli utenti, sviluppo le user stories e compongo la lista dei requisiti dividendoli per tipologia e necessità.

Capitolo 7

Conclusioni

In questo capitolo traggo le conclusioni sul progetto.

7.1 Consuntivo finale

Una volta terminato il progetto ho redatto il consuntivo orario finale nella 7.1 che suddivide in maniera approssimata le ore dedicate alle varie fasi.

Fase	**Ore**
Onboarding del progetto	5
Analisi dei requisiti	30
...	...
Totale	320

Consuntivo orario finale.

7.2 Raggiungimento degli obiettivi

7.3 Requisiti soddisfatti

Arrivato alla fine del progetto ho implementato...

Tipo	Mandatory	Desirable	Optional	Somma
Functional	0/2	0/0	0/0	0/2
Qualitative	0/1	0/1	0/1	0/3
Constraint	0/1	0/0	0/0	0/1
Totale	**0/4**	**0/1**	**0/1**	**0/6**

Riepilogo dei requisiti soddisfatti.

7.4 Rischi occorsi e mitigati

I rischi emersi durante lo stage sono riportati in 7.3.

Descrizione	Mitigazione
R1 – Descrizione del rischio	Soluzione

Rischi occorsi con la loro mitigazione.

7.5 Valutazione personale

Glossario

AGE – Actually Good Encryption: Strumento moderno per la cifratura di file. Utilizza algoritmi crittografici contemporanei come X25519 e ChaCha20-Poly1305, con un'interfaccia volutamente semplice. 3., 4., 8., 9., 11., 14., 15., 26., 30., 34., 35., 39., 40., 48., 53.

Black-box: Approccio di analisi della sicurezza condotto senza accesso al codice sorgente, osservando esclusivamente il comportamento esterno del sistema in risposta agli input forniti. 5.

Branch: Linea di sviluppo parallela all'interno di un sistema di versionamento. Permette di lavorare su funzionalità o correzioni in modo isolato rispetto al ramo principale. 26., 27., 30., 34., 35., 44., 45., 46.

Commit: Unità atomica di modifica in un sistema di versionamento. Rappresenta uno snapshot dello stato dei file in un determinato momento, accompagnato da metadati quali autore, data e messaggio descrittivo. v, vi, 3., 4., 16., 17., 18., 19., 20., 21., 22., 23., 24., 25., 26., 27., 28., 29., 30., 31., 32., 33., 34., 35., 36., 38., 39., 40., 41., 42., 43., 44., 45., 46., 47., 48., 49.

CPL – CodePainter Language: Linguaggio di programmazione proprietario sviluppato da Zucchetti S.p.A. Interpretato, tipizzato staticamente, con sintassi simile al Pascal. Utilizzato per lo sviluppo di RVC e dei prodotti software Zucchetti. 3., 4., 5., 8., 9., 17., 19., 20.

Ed25519: Algoritmo di firma digitale basato sulla curva ellittica Curve25519. Produce chiavi di 256 bit con un livello di sicurezza equivalente a RSA con chiavi da 3000 bit, risultando più efficiente e compatto. 8., 13., 14.

Git: Sistema di versionamento distribuito open source, creato da Linus Torvalds nel 2005. Garantisce l'integrità della storia tramite una catena di hash crittografici. v, 2., 16., 17., 22., 23.

Hash: Funzione matematica che trasforma un dato di dimensione arbitraria in una stringa di lunghezza fissa. Una minima modifica dell'input produce un output completamente diverso, garantendo l'integrità dei dati. v, vi, 2., 9., 12., 17., 18., 21., 22., 24., 34., 36., 38., 39.

JIT – Just-In-Time: Tecnica di compilazione che traduce il codice intermedio in codice macchina durante l'esecuzione, migliorando le prestazioni rispetto alla sola interpretazione. 19.

Manifest: Nel contesto di RVC, file che descrive lo stato di tutti i file tracciati in un progetto in un dato momento: nomi, dimensioni, hash e versione di ogni file. 20.

Namespace: Nel contesto della firma SSH, stringa che identifica il contesto d'uso di una firma e previene attacchi di riutilizzo: una firma prodotta con un certo namespace non può essere usata in un contesto diverso. 14.

Passphrase: Sequenza di parole o caratteri utilizzata per proteggere una chiave privata. Più lunga e complessa di una password tradizionale, viene richiesta ogni volta che si utilizza la chiave. 14.

Repository: Archivio che contiene la storia completa delle modifiche di un progetto sotto controllo versione, inclusi tutti i commit, i branch e i tag. v, 3., 8., 9., 16., 17., 20., 22., 24., 25., 28., 29., 30., 31., 32., 33., 34., 36., 37., 38., 42., 43., 44., 46., 48., 49.

RVC – Repositoryless Version Control: Sistema di versionamento distribuito sviluppato da Zucchetti S.p.A. come alternativa a Git. Garantisce l'integrità dei contenuti tramite verifica crittografica degli hash di ogni commit, permettendo a qualsiasi utente di accertare autonomamente l'autenticità della repository ricevuta. v, vi, vii, xii, 2., 3., 4., 5., 8., 9., 11., 14., 17., 18., 19., 20., 21., 23., 31., 32., 34., 35., 43., 44., 47., 49., 50., 53., 54.

SSH – Secure Shell: Protocollo crittografico per la comunicazione sicura su reti non affidabili. Utilizza crittografia asimmetrica per l'autenticazione e la cifratura del canale. v, vi, xii, 2., 3., 4., 8., 9., 11., 13., 14., 17., 18., 19., 22., 23., 25., 34., 35., 38., 41., 51., 53.

VCS – Version Control System: Sistema software che registra le modifiche ai file nel tempo, permettendo di recuperare versioni precedenti, confrontare cambiamenti e coordinare il lavoro di più sviluppatori. 15., 16.

White-box: Approccio di analisi della sicurezza con piena visibilità del codice sorgente e dell'architettura interna del sistema. Si contrappone all'analisi black-box, condotta senza accesso ai sorgenti. 5.