ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA

Corso di Laurea in Ingegneria e Scienze Informatiche

Nascondere in piena vista: un'introduzione operativa ai canali occulti nelle reti

Tesi di laurea in:
Reti di Telecomunicazione

Relatore
Prof. Franco Callegati

Candidato
Davide Fiocchi

I Sessione di Laurea
Anno Accademico 2024-2025

Sommario

I canali nascosti di rete (*Network Covert Channel*) sono un mezzo per trasmettere informazioni attraverso reti di calcolatori in modo da occultare il fatto stesso che una comunicazione stia avendo luogo. Trovano impiego in scenari in cui le comunicazioni convenzionali risultano troppo rivelanti e l'uso della crittografia è inammissibile, inadeguato o insufficiente. La tesi affronta il tema coniugando la presentazione di alcuni fondamentali aspetti teorici a una prospettiva sperimentale. Dopo aver esaminato la definizione, la natura e la classificazione dei *NCC*, viene presentato un *testbed* progettato per l'analisi *in vitro* di tecniche di comunicazione nascosta. Tre meccanismi per la realizzazione di canali nascosti sono descritti e implementati, andando a costituire, senza perdere di generalità, la base per una discussione sulle sfide e le possibilità legate all'uso dei *Network Covert Channel* (*NCC*). Infine, il traffico generato da ciascuna tecnica è analizzato, confrontato e discusso alla luce del quadro teorico introdotto in precedenza, offrendo alcune riflessioni conclusive.

A mamma e papà, per il loro supporto completo e incondizionato.

Indice

Elenco delle figure

Elenco dei listati

"Aveva deciso che lasciarla proprio sotto il naso del mondo intero fosse il miglior modo per impedire al mondo di notarla."

Edgar Allan Poe, *La lettera rubata*

Capitolo 1

Introduzione

1.1 Comunicare in sicurezza

La trasmissione sicura[1] di messaggi segreti in una comunicazione implica due aspetti fondamentali: la sicurezza del contenuto e la sicurezza della connessione [TXLG20].

Nelle moderne reti di calcolatori, la sicurezza del contenuto di una comunicazione è normalmente raggiunta attraverso la crittografia. In alcuni casi, tuttavia, l'esistenza stessa di una comunicazione, o il cambiamento delle sue caratteristiche (ad esempio un aumento della frequenza dei messaggi), può essere sufficiente a sollevare sospetti e/o rivelare informazioni [ZAB07]. Inoltre, non è sempre possibile fare affidamento sulla disponibilità o l'autorizzazione all'uso di meccanismi crittografici sufficientemente sicuri. Ad esempio, un governo potrebbe custodire segretamente i migliori protocolli, vietarne l'utilizzo o, peggio, renderne disponibili versioni modificate contenenti *backdoors*[2] [GKS18].

In termini di sicurezza della connessione, i meta-dati (indirizzo sorgente e destinazione, etc.) e alcune caratteristiche della comunicazione (andamento del traffico, etc.) non possono in molti casi essere nascosti attraverso cifratura, con il rischio

[1] La sicurezza delle informazioni è la tutela di confidenzialità, integrità e disponibilità; possono essere incluse altre proprietà come autenticità, non ripudiabilità e affidabilità [fS18].

[2] Una *backdoor* è una funzionalità intenzionalmente nascosta in un sistema allo scopo di ottenervi accesso, anche non autorizzato, in un secondo momento. Nel contesto crittografico, può consistere in una vulnerabilità la cui conoscenza permette di violare il cifrario.

che vengano utilizzati da un eventuale ascoltatore per ricavare informazioni sull'identità e attività dei partecipanti [TXLG20]. Un *Network Covert Channel* (*NCC*) è un mezzo di comunicazione inconsueto che può impedire al traffico (cifrato o meno) di essere rilevato grazie alle sue proprietà di occultamento.

1.1.1 Canali di comunicazione

Un *Overt Channel* (*OC*, canale evidente, aperto) è un canale di comunicazione all'interno di un sistema o una rete di elaboratori progettato per il trasferimento autorizzato di informazioni. Un *Covert Channel* (*CC*, canale nascosto, occulto), invece, è qualsiasi canale di comunicazione che può essere sfruttato per trasferire informazioni in un modo che viola le politiche di sicurezza del sistema [MP14].

I *CC*s vengono inizialmente studiati nel contesto di singoli sistemi, come meccanismi attraverso cui processi a un alto livello di confidenzialità possano far trapelare informazioni, consegnandole a processi che non vi avrebbero altrimenti accesso [Lam73].

A partire dai primi anni '90, con lo sviluppo delle reti di elaboratori, l'interesse si sposta sulla possibilità di nascondere informazioni all'interno dei protocolli di rete. La vasta quantità di traffico e il variegato panorama di protocolli operanti in Internet lo rendono infatti un vettore ideale per le informazioni nascoste (vedi sezione 1.1.4) [ZAB07]. In questo nuovo contesto, l'obiettivo non è più tanto quello di stabilire una comunicazione che violi le regole, quanto piuttosto quello di trasferire informazioni attraverso la rete in modo tale da nascondere l'esistenza stessa della comunicazione. Nascono così i *Network Covert Channel* (*NCC*) (canali occulti di rete): tecniche che consentono di nascondere uno scambio di informazioni sfruttando i protocolli di rete come mezzi di trasporto apparentemente legittimi (canali evidenti) [WZFH14].

1.1.2 Scenari applicativi

Tipicamente, l'utilizzo di un *CC* è motivato dall'esistenza di una relazione antagonistica tra due o più parti, di cui una cerca di comunicare senza essere scoperta, mentre l'altra tenta di intercettare o rilevare la comunicazione [ZAB07].

Molte applicazioni dei *CC*s sono di natura malevola, come l'esfiltrazione di dati sensibili da parte di malware, o il coordinamento di botnet [CZN+20]. Ciononostante, le stesse tecniche possono essere impiegate anche per fini legittimi, come l'elusione della censura [CZJ+24], o il rintracciamento dell'origine di attacchi informatici (ad esempio di tipo *Denial of Service* (*DoS*)) [ZAB07].

In definitiva, un *CC* è uno strumento come altri: la sua eticità dipende dai fini per cui è impiegato; è quindi fondamentale non solo comprenderne le potenzialità, ma anche sviluppare contromisure efficaci per difendersi da un uso improprio.

1.1.3 Nascondere in piena vista

L'elusività dei *CC*s deriva dall'utilizzo di mezzi di comunicazione non progettati per il trasferimento di informazioni [ZAB07]. Questo rende difficile per un potenziale ascoltatore determinare se un utente stia trasmettendo dati nascosti o meno, contribuendo a proteggere l'identità di entrambe le parti comunicanti.

Si tratta, in sostanza, di un'applicazione del principio di *"security through obscurity"* ("sicurezza mediante segretezza"), secondo cui la sicurezza di un sistema è affidata, in tutto o in parte, al fatto che il suo funzionamento (o la sua stessa esistenza) rimanga sconosciuta a un osservatore esterno. Applichiamo questo principio quotidianamente, magari senza accorgercene, ad esempio quando nascondiamo le chiavi di casa sotto allo zerbino o in un vaso di fiori [ZCC00]. Allo stesso modo, nel contesto dei *CC*s l'informazione è spesso[3] "nascosta in piena vista"; ovvero, viene incorporata all'interno di flussi di dati legittimi in un modo tale che, pur essendo tecnicamente accessibile, rimanga inosservata agli occhi di chi non conosce il meccanismo di codifica utilizzato.

D'altra parte, poiché il traffico dei *CC*s è tipicamente mescolato a grandi volumi di traffico aperto, anche se un ascoltatore riuscisse ad accorgersi dello scambio in corso, sarebbe comunque difficile stabilire se e chi stia inviando o ricevendo messaggi in un dato momento. Inoltre, nel caso di utilizzo abbinato a tecniche crittografiche, le caratteristiche di una comunicazione attraverso *CC* risultano dif-

[3] In base alla tecnica impiegata (vedi sezione 2.2). Ad esempio, un canale che sfrutta la temporizzazione delle trame in una rete *Ethernet broadcast* è accessibile a tutti i nodi, mentre un canale che nasconde dati all'interno di *PDU* incapsulate in *TLS* è molto più difficile da intercettare.

ficilmente interpretabili, rendendo inefficaci diversi attacchi basasti sull'analisi del traffico cifrato [TXLG20].

Per questi motivi, i *CC*s possono fornire un buon livello di sicurezza sulla connessione, andando a complementare la protezione fornita dalla crittografia, se presente [TXLG20].

1.1.4 Un terreno fertile

I protocolli operanti in internet rappresentano un vettore ideale per la realizzazione di un canale di comunicazione nascosto, per diverse ragioni.

In primo luogo, le definizioni dei protocolli nello *stack Transmission Control Protocol/Internet Protocol* (*TCP/IP*) sono state sviluppate con lo scopo di essere il più universali possibili, fornendo ai programmatori di reti un numeroso insieme di funzionalità opzionali [Kwe06]. Inoltre, alle diverse implementazioni è richiesto un buon grado di tolleranza anche a comportamenti non standard, permettendo in molti casi a implementazioni diverse delle stesse specifiche di utilizzare soluzioni diverse per raggiungere lo stesso (o quasi) risultato.

Questo comportamento, noto come "principio di robustezza" è stato formalizzato da Jon Postel[4] in una delle prime definizioni di *Transmission Control Protocol* (*TCP*):

> *"Le implementazioni di TCP dovrebbero seguire un principio generale di robustezza: essere conservativi in ciò che si invia e liberali in ciò che si accetta dagli altri [Pos80]."*

Ciononostante, uno studio del 2020 ha rilevato che un numero non trascurabile di host operanti in internet non aderisce nemmeno ai requisiti obbligatori delle specifiche *TCP/IP*, con anomalie che spaziano dal trattamento scorretto dei *checksum*[5] all'eliminazione di opzioni *TCP* da parte di nodi intermedi [KBR+20].

L'esistenza di standard particolarmente transigenti, in contrasto a una realtà implementativa invece spesso troppo rigida ha portato a un panorama protocollare

[4] Il principio di robustezza è anche conosciuto come Legge di Postel.

[5] Un meccanismo di rilevazione degli errori che si basa sul calcolo della somma modulo due dei blocchi che compongono un messaggio.

in cui sezioni, campi e valori opzionali sono generati e trasmessi dal mittente solo per essere ignorati o direttamente eliminati dal ricevitore [Kwe06].

Numerose tecniche di realizzazione dei *CC* si fondano sulle ambiguità presenti nei protocolli, come la coesistenza di più modalità per ottenere lo stesso comportamento o la possibilità di inserire dati in punti che le implementazioni comuni ignorano. Ad esempio, le specifiche di un protocollo possono prevedere funzionalità il cui uso non è obbligatorio, includere campi inizializzati casualmente o demandare all'implementazione la scelta del contenuto di determinate parti delle *Protocol Data Unit* (*PDU*) [DM17].

In conclusione, la perdurante esistenza di un ambiente favorevole allo sviluppo di tecniche di comunicazione nascosta attraverso i protocolli di rete è dovuta al delicato equilibrio da ricercare al momento della standardizzazione. Da un lato, è sicuramente desiderabile porre rimedio ad ambiguità ed eccessi già durante la fase di progettazione, ma dall'altro, specifiche troppo rigide potrebbero rendere un protocollo difficile da implementare o utilizzare. L'attività di prevenzione dei *CC* è quindi un compito difficile, che richiede di soppesare accuratamente i pro costituiti dalle funzionalità messe a disposizione da un protocollo con i contro costituiti dalla superficie d'attacco che esse introducono [CM24].

1.2 Modello di riferimento

1.2.1 Il problema del prigioniero e il canale subliminale

Nel 1983, Simmons introduce il modello concettuale del *Prisoner's Problem* (Problema del Prigioniero)[6], che nel tempo si è costituito come modello *de facto* di una comunicazione mediante *CC* [ZAB07, Sim83]. Lo scenario, visibile in fig. 1.1, è il seguente:

> Due persone macchiatesi di un reato vengono arrestate e messe in isolamento. Chiameremo i nostri complici Alice e Bob[7].

[6] Da non confondere con il "dilemma del prigioniero" della teoria dei giochi.

[7] Seguendo la tradizione del mondo crittografico iniziata da Rivest, Shamir e Adleman e diventata ormai convenzione universalmente accettata [RSA78, Shi07].

> Il guardiano della prigione, Wendy, è disposto a permettere ai prigionieri di scambiarsi messaggi, nella speranza di indurre uno dei due a rivelare informazioni compromettenti, tramite un messaggio fraudolento modificato in parte o interamente scritto da egli stesso. Tuttavia, avendo ogni ragione di credere che i complici vogliano organizzare un tentativo di fuga, Wendy permetterà agli scambi di continuare solo se le informazioni contenute nei messaggi gli sono evidenti e le presume innocue.
>
> Alice e Bob sono disposti ad accettare il rischio di essere ingannati pur di riuscire a comunicare, in quanto hanno bisogno di coordinarsi per pianificare la fuga. Per riuscire nel loro intento, devono ingannare il guardiano e trovare un modo di comunicare segretamente attraverso gli scambi, ovvero di stabilire un "canale subliminale"[8] in piena vista del guardiano, scambiandosi messaggi apparentemente innocui.

Handel et al. sfrutta questo modello per descrivere le comunicazioni nascoste nel contesto delle reti di elaboratori [HI96]. Nel nuovo scenario, Alice e Bob possono fare uso di due computer connessi da una rete, i quali realizzano un canale di comunicazione apparentemente innocuo, ma utilizzato come vettore di un canale nascosto. Un guardiano (Wendy) gestisce la rete e può monitorare ed eventualmente alterare il traffico per verificare l'esistenza di canali nascosti.

[8] Vedi sezione 1.3.

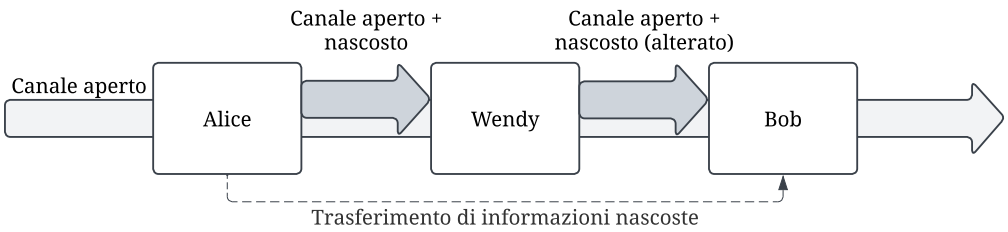


Figura 1.1: Il modello concettuale proposto da Simmons [Sim83, ZAB07].

Il modello descritto rappresenta comunque una semplificazione della realtà, in quanto in generale i canali nascosti non sono limitati alla comunicazione punto-punto, e Alice e Bob non devono essere per forza gli estremi del canale aperto (vedi sezione 2.3).

1.2.2 Tipologie di guardiani

Nel contesto di un *Network Covert Channel* si può distinguere tra tre categorie principali di avversari (guardiani, nel problema del prigioniero), in base alla modalità di attacco [Cra98]:

- Un guardiano **passivo** tenta di determinare la presenza di un canale nascosto e di estrapolare le informazioni che vi transitano, o anche solo di dimostrare il coinvolgimento di una delle parti della comunicazione nascosta, limitandosi ad ascoltare il traffico.
- Un guardiano **attivo** può modificare leggermente il contenuto dei messaggi scambiati tra i prigionieri, ma senza alternarne considerevolmente il significato. Ovvero, non deve modificare i messaggi tanto da impedire un'eventuale comunicazione innocua.
- Un guardiano **malevolo**[9] può alterare a piacimento i messaggi scambiati, o addirittura fabbricarne taluni da zero, impersonando uno dei prigionieri. Questo scenario non è per noi di grande interesse in quanto presenta poche opportunità di comunicazione per i prigionieri, inoltre le situazioni reali spesso impediscono un tale comportamento da parte di un guardiano.

Esistono ulteriori criteri sulla base dei quali è possibile classificare i guardiani [TXLG20], che tralasceremo per non appesantire la trattazione.

1.3 Terminologia

Il termine "canale subliminale" (in origine *"subliminal channel"*) introdotto da Simmons (sezione 1.2.1) non è altro che una variante del concetto di *Covert*

[9] In letteratura è comune trovare gli avversari malevoli e quelli attivi raggruppati in un'unica categoria, definiti semplicemente come "attivi".

Channel che si distingue per l'utilizzo di algoritmi e/o protocolli crittografici [MP14].

La letteratura, infatti, utilizza diversi termini per descrivere l'atto di nascondere informazioni sia in generale, sia nel contesto specifico dei protocolli di rete [ZAB07]. Questa frammentazione è in parte dovuta alla natura multidisciplinare dell'argomento, che coinvolge autori con prospettive molto diverse, ma anche all'evoluzione del lessico, che ha visto nel tempo ridefinizioni talvolta contrastanti nonché l'introduzione di concetti e terminologie precedentemente inesistenti.

Come accennato, il primo utilizzo del termine *Covert Channel* (*CC*) avviene in un contesto locale a una singola macchina. Viene coniato da Lampson nel 1973 per indicare una delle tre classi da egli individuate di canali attraverso cui è possibile trasferire informazioni in modo non autorizzato tra processi in esecuzione a diversi livelli di confidenzialità, violando le politiche di sicurezza di un sistema multiprogrammato. I canali *covert*, in particolare, sono quelli

> *"non intesi affatto per il trasferimento di informazioni"*

come, ad esempio, gli effetti dell'esecuzione di un programma sul carico del sistema [Lam73].

Il termine è successivamente recuperato dal Dipartimento della Difesa degli Stati Uniti (*Department of Defense*) che lo definisce come

> *"qualsiasi canale di comunicazione che può essere sfruttato da un processo per trasferire informazioni in un modo che viola le politiche di sicurezza di un sistema",*

perdendo l'enfasi sull'idea di utilizzo imprevisto [oD85].

La successiva rielaborazione di questa idea nell'atto di nascondere informazioni all'interno degli header di diversi protocolli di rete viene introdotta sotto il nome di *Network Covert Channel*, principalmente perché termini più adeguati come *Information Hiding* non erano ancora stati inventati [Mil99]. Infatti, Petitcolas nel 1999 riunisce sotto questo unico termine ombrello diverse classi (fig. 1.2) di tecniche utilizzate per l'occultamento di informazioni di vario genere [PAK99], tra cui i *CC* e la steganografia.

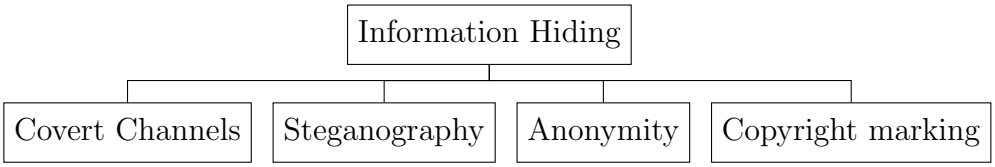


Figura 1.2: La tassonomia identificata da Petitcolas, qui riportata solo fino al primo livello [PAK99].

Steganografia (*steganography*) deriva dal greco e significa letteralmente "scrittura nascosta". Il termine è ricorrentemente utilizzato per riferirsi all'atto di nascondere informazione all'interno di altri dati. I *NCC* presentano diverse similarità con la steganografia, al punto da esserne a volte considerati una sottoclasse [MP14]. Tuttavia, il principio alla base è differente e questo comporta una sottile ma significativa discrepanza: la steganografia richiede qualche forma di contenuto come copertura, mentre un *NCC* richiede solo l'esistenza di un protocollo di rete [ZAB07].

Adottiamo, coerentemente con i lavori di Zander et al. e Petitcolas, il termine *"Covert Channel"* (e i corrispondenti termini italiani "canale nascosto" e "canale occulto"), che utilizzeremo in modo intercambiabile con *"Network Covert Channel"* quando ciò non introduca ambiguità, per indicare le sole tecniche di occultamento delle informazioni all'interno di protocolli di rete **che non comportano la modifica del contenuto informativo**. Chiameremo inoltre "informazioni nascoste" le informazioni trasferite attraverso un *Covert Channel*.

1.4 Struttura e obiettivi del lavoro

Il resto del lavoro è strutturato come segue. Il capitolo 2 fornisce una panoramica di alcuni dei principali lavori esistenti in tema di classificazione e tassonomia dei *CC*, corredata da alcuni esempi rappresentativi. L'obiettivo è quello di offrire un primo orientamento nel campo dei *CC*, che si dimostrerà utile a contestualizzare le tecniche approfondite nel seguito. Il capitolo 3 descrive i requisiti e l'architettura di un *testbed* sviluppato appositamente per l'esplorazione sperimentale di meccanismi di comunicazione nascosta in uno scenario di rete simulato. I successivi capitoli da 4 a 6 presentano quindi tre esempi di canali nascosti, sviluppando la

discussione a partire dai presupposti teorici fino ad arrivare all'implementazione. Il traffico generato dalla comunicazione nascosta attraverso ciascun *CC* è registrato, analizzato e confrontato, andando a costituire la base per la discussione dei risultati ottenuti, presentati nel capitolo 7.

Capitolo 2

Classificazione e tassonomia

2.1 Letteratura di riferimento

La classificazione dei *CC*s offre un quadro di riferimento utile per analizzare a un livello più astratto le tecniche esistenti e future, evidenziando le somiglianze tra le tecniche e semplificando lo sviluppo di nuove contromisure [ZAB07]. Nel corso degli anni, sono stati proposti diversi sistemi di classificazione delle tecniche di *CC*, in molti casi costruendo incrementalmente sui lavori precedenti.

Diversi lavori catalogano le tecniche sulla base di quale strato OSI [HI96] o TCP/IP [MP14, Row97] interessino. Tuttavia, molte tecniche presentano meccanismi di base che si prestano a essere utilizzati in modo molto simile anche in contesti diversi. Per questo, Zander et al. suggeriscono l'importanza di considerare la possibile applicabilità di una tecnica anche al di fuori del contesto in cui è stata sviluppata, individuando una classificazione basata su 19 meccanismi di funzionamento generali [ZAB07].

Ciò nonostante, questa classificazione risulta piuttosto dettagliata, e non propone né una struttura gerarchica tra le tecniche, né una base teorica facilmente estendibile o adatta a supportare sviluppi futuri [WZFH14].

Con l'obiettivo di superare i limiti delle classificazioni esistenti, Wendzel et al. analizzano 109 tecniche sviluppate tra il 1987 e il 2013, integrando i lavori precedenti, tra cui quello di Zander et al. Ne risulta una classificazione in soli 11 pattern (riportata in fig. 2.1), di cui appena 4 raccolgono circa il 70% di tutti i *CC*

analizzati, a dimostrazione dell'importanza di una struttura concettuale al fine di valutare il grado di novità delle tecniche emergenti [WZFH14].

Altri autori hanno analizzato i canali nascosti da punti di vista differenti, focalizzandosi su proprietà della comunicazione osservabili a prescindere dalle tecniche di implementazione. Un esempio significativo in tal senso è rappresentato dalla tassonomia basata sull'entropia delle sorgenti esplicite proposta da Zhiyong e Yong [ZY09].

2.2 Tecnica impiegata

Qualsiasi risorsa condivisa può potenzialmente essere utilizzata per realizzare un canale di comunicazione nascosto [MP14]. Fino ad ora abbiamo parlato di *CC*s in modo generico, ma il concetto è stato esplorato in diversi contesti, per cui si può parlare di *CC*s *host-based* (locali), *network-based* (di rete) e *physical* (fisici) [MDS23]. In questa trattazione ci occuperemo solo di *Network Covert Channel*s, indicandoli semplicemente come *Covert Channel*s.

Tradizionalmente, i *CC*s vengono divisi in due categorie principali [oD85]:

- **canali di immagazzinamento (*storage channels*)**: permettono la trasmissione di informazioni da parte del mittente attraverso la scrittura (diretta o indiretta) di dati all'interno di una risorsa condivisa, e la successiva lettura (diretta o indiretta) di tali dati da parte del destinatario;
- **canali di temporizzazione (*timing channels*)**: consentono al mittente di codificare informazioni modulando nel tempo il proprio utilizzo di una qualche risorsa condivisa in modo tale che il cambiamento possa essere visto e decodificato da un destinatario.

I canali di immagazzinamento sono i più comuni, in quanto solitamente soggetti a una quantità limitata o nulla di rumore, tuttavia la maggior parte di essi è anche facile da rilevare ed eliminare. I canali di temporizzazione sono di più difficile implementazione e sempre soggetti a rumore, ma sono generalmente molto più furtivi e difficili da contrastare [WZFH14].

Nonostante questa distinzione possa risultare alle volte sfumata [ZAB07], rappresenta comunque la base di molte tassonomie.

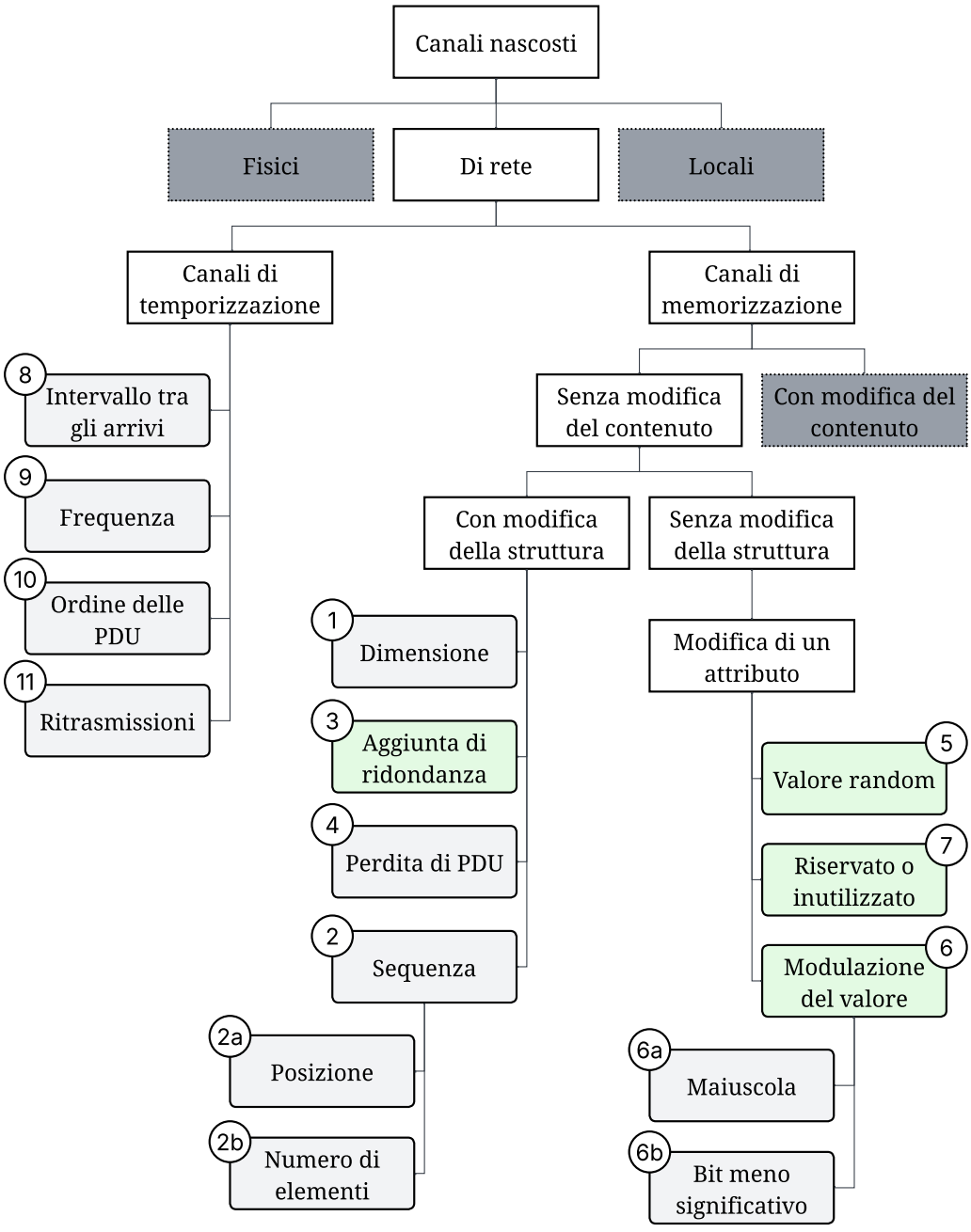


Figura 2.1: La tassonomia individuata da Wendzel et al. [WZFH14]. In verde i quattro pattern alla base della maggior parte delle tecniche.

2.2.1 Canali di temporizzazione

Tra i canali di temporizzazione, uno dei pattern più comuni è quello dell'*intervallo tra gli arrivi* (fig. 2.1). Le tecniche che ricadono in questo schema alterano l'intervallo temporale tra l'invio (e, di conseguenza, la ricezione) di *PDU*s successive, al fine di codificare informazioni nascoste [WZFH14]. Un esempio calzante è rappresentato dal *Keyboard JitterBug*, un *CC* che sfrutta le caratteristiche temporali del flusso di pacchetti di una sessione *Secure SHell* (*SSH*) per nascondere informazioni. In molte applicazioni interattive via rete, infatti, ogni pressione di un tasto causa l'invio di un pacchetto (generalmente cifrato). Modulando artificialmente i ritardi tra la pressione di un tasto e l'invio del relativo pacchetto, è possibile codificare informazioni nascoste che un ricevitore attento può decodificare misurando il tempo di attesa tra pacchetti successivi [SM06].

La temporizzazione nelle reti è uno strumento molto versatile. Al fine di codificare informazioni è anche possibile: modulare la frequenza di invio dei pacchetti in un determinato intervallo di tempo (pattern *frequenza*), scegliere un ordinamento arbitrario tra le possibili permutazioni di un insieme di *PDU* da inviare (pattern *ordine delle PDU*), o fingere la perdita di pacchetti e *acknowledgements* ritrasmettendo *PDU* già riscontrate e/o saltando numeri di sequenza al momento dell'invio (pattern *ritrasmissioni*) [WZFH14].

Il punto di forza, e contemporaneamente di debolezza, di questa classe di canali è la natura stocastica del traffico di rete. Infatti, la temporizzazione dei pacchetti può, in generale, essere condizionata da una grande varietà di fattori. Questo rende difficile (ma non impossibile, vedi sezione 2.4.3) distinguere le anomalie introdotte da una manipolazione intenzionale da quelle dovute alla natura del canale aperto utilizzato. Se da un lato questo rende i canali di temporizzazione più furtivi, dall'altro richiede una gestione più attenta della codifica e decodifica.

2.2.2 Canali di immagazzinamento

I canali di immagazzinamento si prestano per la loro diversità a essere classificati in due ulteriori categorie: i canali che utilizzano tecniche per nascondere informazione all'interno del contenuto delle *PDU* e i canali che nascondono informazione senza alterare il contenuto. La prima di queste categorie fa riferimento a tecniche

tipicamente riconducibili alla steganografia, ed esula dall'ambito di questa tesi. La seconda, invece, presenta tecniche che possono essere utilizzate per realizzare quelli che definiamo come *Covert Channel*s veri e propri (vedi sezione 1.3), e può essere nuovamente suddivisa in base alla caratteristica di modificare o meno la struttura delle *PDU*. Un canale che non modifica la struttura risulta in generale più furtivo [WZFH14].

Con modifica della struttura

Una modifica della struttura consiste in un'alterazione non distruttiva della sintassi di una *PDU*, come la variazione dell'ordine degli elementi che la costituiscono, o la modifica della sua dimensione [WZFH14]. Un perfetto esempio è rappresentato da *HIDE_DHCP*, un *CC* che integra ben tre meccanismi di modifica della struttura di messaggi *Dynamic Host Configuration Protocol* (*DHCP*) per potersi adattare a contesti con diversi requisiti in termini di larghezza di banda e furtività [ROL12]:

- **Numero di opzioni**: la lettera "A", ad esempio, potrebbe essere codificata dalla trasmissione di un messaggio con 2 opzioni, la lettera "B" da un messaggio con 3 opzioni e così via (pattern *numero di elementi*).

- **Ordine delle opzioni**: ci sono diversi modi per sfruttare l'ordinamento; ad esempio, potremmo considerare le opzioni A, B, C e D in questo particolare ordine, e trattarle come un numero binario di cui ogni cifra equivale a 1 se l'opzione corrispondente è in posizione corretta e 0 se non lo è (pattern *sequenza*). In questo caso, un messaggio contenente le opzioni C, B, A, D corrisponderebbe al messaggio 0101_2.

- **Tipo delle opzioni**: potendo impostare arbitrariamente il tipo di un'opzione *DHCP* all'interno del range 0-255, possiamo codificare un *byte* (ad esempio, un carattere) all'interno di ogni messaggio scegliendo quale opzione inviare in una data posizione (pattern *position*). Ad esempio, se la seconda posizione è quella scelta per la trasmissione e il messaggio contiene l'opzione numero 65 (*"NIS-Server-Addr"*) in seconda posizione, l'informazione nascosta è la lettera "A".

Illustreremo nel dettaglio altre due tecniche che appartengono a questa categoria nei capitoli 4 e 6.

Senza modifica della struttura

Se un *CC* non modifica la struttura, è necessario che modifichi dei dati al loro interno (e.g. un campo dell'header) [WZFH14]. Rowland propone diverse tecniche per la comunicazione nascosta che ricadono all'interno del pattern *valore random*, caratterizzato dallo sfruttamento di campi i cui valori sono inizializzati casualmente [Row97, WZFH14]. Ad esempio, è possibile codificare un *byte* di informazione nascosta utilizzandolo come ottetto più significativo del *TCP Initial Sequence Number* (*ISN*) (Numero di Sequenza Iniziale), oppure del campo *ID* di un pacchetto *IP* frammentato.

Un ulteriore esempio è costituito dalla tecnica presentata nel capitolo 5.

2.3 Ruolo di mittente e destinatario

Sono possibili diversi scenari di comunicazione nascosta, a seconda del ruolo che Alice e Bob ricoprono nel canale palese (*overt*): potrebbero essere i mittenti/destinatari diretti o agire da intermediari (*middleman*), manipolando un canale tra utenti inconsapevoli.

Se il mittente del canale nascosto è anche il mittente del canale aperto si definisce un **mittente attivo**, ed è in grado controllare pienamente la comunicazione segreta, ad esempio ottimizzandone la capacità o la furtività. In certi casi, tuttavia, il mittente nascosto non può o non vuole creare canali aperti per non attirare l'attenzione, e decide di inserire il canale nascosto in una comunicazione aperta già esistente, assumendo il ruolo di **mittente passivo** [WSZK25, ZAB07].

Anche il destinatario nascosto può sia coincidere con il destinatario del canale aperto (**destinatario attivo**) che agire da intermediario (**destinatario passivo**) estraendo le informazioni da una comunicazione diretta verso un terzo utente inconsapevole. In questa situazione, il ricevitore nascosto potrebbe voler rimuovere il canale nascosto per ridurre la possibilità di rilevazione [WSZK25, ZAB07].

Essere intermediari non implica necessariamente una separazione fisica dagli attori principali: i partecipanti alla comunicazione nascosta possono trovarsi su router o gateway lungo il percorso della comunicazione, o anche sullo stesso dispositivo, ma a livelli più bassi dello stack di protocolli di rete [ZAB07].

Le possibili combinazioni di mittenti e destinatari sono rappresentate in fig. 2.2, distinguiamo tra le seguenti tipologie di canali:

Mittente	**Destinatario**	**Canale**
Attivo	Attivo	Attivo
Attivo	Passivo	Semiattivo
Passivo	Attivo	Semipassivo
Passivo	Passivo	Passivo

Tabella 2.1: Classificazione dei canali in base alle caratteristiche di mittente e destinatario [WSZK25].

Citiamo per completezza il recente lavoro di Wendzel et al. [WSZK25], in cui sono state individuate due ulteriori categorie di canali che trasmettono informazioni facendo uso di traffico generato da terzi, ma senza modificarlo, e non ricadono in nessuna delle quattro classi descritte.

2.4 Altre caratteristiche

2.4.1 Semantica

Un *CC* modifica la semantica di una *PDU* se, attraverso la modifica di elementi dell'*header*, causa un cambiamento nel modo in cui essa viene interpretata. Ad esempio, la semantica di un header *IPv4* è modificata se l'opzione *"Record Route"* è impostata, mentre è preservata se il *flag "Reserved"* è impostato, in quanto quest'ultimo non ha effetto sull'interpretazione del pacchetto da parte di un ricevitore. In generale un canale attira meno attenzione se non implica una modifica della semantica [WZFH14].

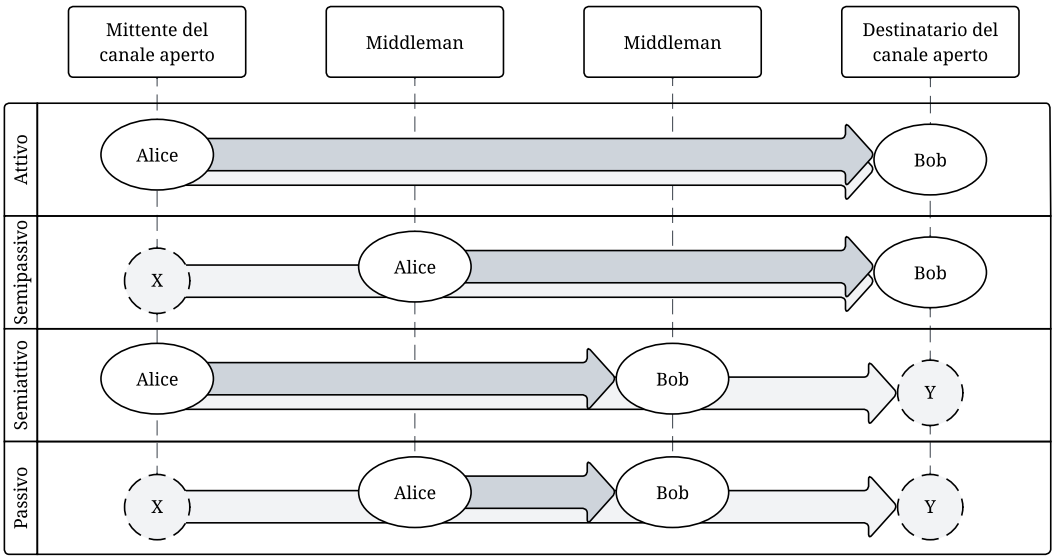


Figura 2.2: Gli scenari più comuni di una comunicazione mediante *CC*. [ZAB07, WSZK25]

2.4.2 Rumore

Come qualsiasi altro canale di comunicazione un covert channel può in generale essere affetto da rumore. Questo dipende in primo luogo dalla natura aleatoria delle reti di elaboratori, in cui trame e pacchetti possono arrivare a destinazione fuori ordine, danneggiati, o non arrivarci affatto; tuttavia, non è da escludere nemmeno la possibilità che sia un guardiano attivo a introdurre rumore volontariamente (ad esempio attraverso l'invio di pacchetti fittizi), con l'obiettivo di confondere una possibile comunicazione nascosta [ZAB07].

In base alla tecnica utilizzata per implementare i canali nascosti, questi disservizi negli *Overt Channel*s possono tradursi o meno in perdite o errori nell'informazione nascosta. Ad esempio, alcuni canali di tipo storage sono realizzati tenendo conto dei meccanismi di affidabilità del/i protocollo/i su cui si appoggiano (e.g. TCP), e, assumendo che i campi dell'header che utilizzano non vengano cambiati durante il tragitto, si possono considerare a tutti gli effetti esenti da rumore (*noiseless*). I canali nascosti di tipo timing, al contrario, sono di natura soggetti al rumore (*noisy*), in quanto la temporizzazione di trame e pacchetti è strettamente

legata alle condizioni della rete (e.g. traffico intenso, congestione) [WZFH14].

2.4.3 Entropia

Nella teoria dell'informazione, una sorgente (di informazione) può essere vista come una variabile aleatoria X che assume un insieme di valori a_i secondo una distribuzione di probabilità $p(a_i)$. La quantità di informazione associata a un dato valore è $I(a_i) = -\log(p(a_i))$; ovvero, valori più rari sono più informativamente "preziosi". L'entropia di una sorgente è definita come la speranza matematica della sua quantità di informazione, ed è data da:

$$H(X) = -\sum_{i=1}^{n} p(a_i) \log(p(a_i))$$

L'entropia rappresenta un indicatore della prevedibilità di una sorgente: quanto più è bassa l'entropia, tanto più è prevedibile il suo comportamento. È possibile classificare i *CC* in base al canale aperto che utilizzano [ZY09]:

- Una sorgente ad **entropia fissa** (*fixed entropy*) invia dati in modo determinato. Ad esempio invia contenuti fissi, oppure associa sempre ad un particolare innesco la stessa risposta.
- Una sorgente a **entropia costante** (*constant entropy*) è definita da una variabile casuale standard. In altri termini, la sorgente genera dati secondo una distribuzione di probabilità nota.
- Una sorgente a **entropia variabile** (*variety entropy*) può generare dati secondo molteplici distribuzioni di probabilità.

Approfondire maggiormente l'argomento richiederebbe l'introduzione di strumenti matematici e statistici che esulano dall'ambito di questa tesi. Ci limiteremo a notare che la furtività di un *CC* è direttamente proporzionale all'entropia del canale aperto che utilizza. Infatti, un canale nascosto che desideri rimanere anonimo deve mimetizzare il proprio traffico, rendendolo il più possibile indistinguibile da quello del canale aperto che utilizza. Tuttavia, una sorgente altamente prevedibile offre poco margine di manovra per l'inserimento di informazione nascosta senza alterare in modo rilevabile le caratteristiche statistiche del traffico [ZY09].

"Ti sei accorto che i più imparano meglio facendo le cose piuttosto che sentendone parlare?"

Platone, *Repubblica, Libro VII*

Capitolo 3

Sviluppo di un *testbed*

3.1 Scopo e requisiti

Ci proponiamo ora di realizzare un *testbed* (banco di prova) per approfondire l'analisi delle tecniche di comunicazione nascosta attraverso *Covert Channel*s. L'obiettivo è quello di affiancare alla discussione teorica alcuni esempi concreti, nella speranza che questi permettano di mettere in luce aspetti di complessità che potrebbero sfuggire a un'analisi esclusivamente concettuale.

Le condizioni minime per la realizzazione di un *CC* sono l'esistenza di un protocollo di rete, di un canale di comunicazione e di una coppia di nodi che ne faccia uso: un mittente e un destinatario. Volendo studiarne le caratteristiche, è necessario aggiungere un terzo nodo in grado di osservare il traffico scambiato sul canale. Questo ascoltatore, analogamente a un guardiano passivo nel contesto del problema del prigioniero (vedi sezione 1.2.1), non partecipa alla comunicazione ma è in grado di monitorarla e, potenzialmente, rilevare la presenza di attività anomale o sospette.

Parametri importanti da considerare sono la controllabilità e la riproducibilità del contesto in cui avviene la comunicazione. In un ambiente reale, fattori esterni possono introdurre variabilità indesiderata (ad esempio, ritardi o perdite di pacchetti dovuti alla congestione della rete), e il comportamento dei nodi intermedi (come *router* e *gateway*) è imperscrutabile agli utenti. Questi dispositivi, infatti, possono agire in maniera simile a guardiani attivi, alterando e normalizzando i

pacchetti in transito e rendendo difficile isolare gli effetti della tecnica che si sta studiando [Cav21].

Realizzare un'analisi critica del funzionamento di un *CC* richiede, invece, la possibilità di mantenere costanti le componenti del sistema, al fine di confrontare diverse tecniche o configurazioni al netto delle medesime condizioni iniziali. Risulta quindi essenziale avere il controllo su tutti e tre i nodi: i due estremi devono essere in grado di comunicare tra loro secondo le regole del canale nascosto, mentre il nodo centrale deve essere in grado di intercettare e catturare il traffico senza alterarlo, rendendone disponibile una registrazione.

A partire da questi dati, è importante chiedersi in che modo e mediante quali strumenti un ascoltatore possa accorgersi della presenza di un *CC*, e quali analisi sia opportuno e/o possibile effettuare per evidenziare schemi, anomalie o segnali che indichino la trasmissione di informazioni non esplicite.

Da ciò nasce un'esigenza di interpretabilità del traffico catturato, che rende necessaria una selezione attenta dei protocolli da utilizzare. In particolare, per i nostri scopi ha poco senso impiegare protocolli che comportino un uso estensivo della crittografia, per quanto questa sia ormai la norma negli scenari reali (vedi sezione 6.1). La corretta applicazione di tecniche crittografiche, infatti, maschera il contenuto e la struttura del traffico, aggiungendo un livello di complicazione non utile all'analisi del comportamento di un *CC*.

D'altra parte, come già accennato nel capitolo 1, l'impiego della crittografia e l'utilizzo di tecniche di comunicazione nascosta rispondono a logiche differenti e agiscono su livelli separati: i *CC* possono essere implementati indipendentemente dall'utilizzo di tecniche crittografiche, e il loro funzionamento rimane sostanzialmente invariato in entrambi i casi. Più precisamente, la presenza o meno di cifratura non incide generalmente sulle modalità con cui un *CC* trasmette l'informazione nascosta, ma può incidere sulla sua rilevabilità. Ad esempio, nella maggior parte dei *CC* attivi, in cui il destinatario della comunicazione nascosta rappresenta anche il destinatario del canale aperto (ovvero, partecipa attivamente al processo di codifica e decodifica) la crittografia rappresenta al massimo un livello di protezione aggiuntivo. Nel caso di un destinatario passivo (ovvero, non in grado di decodificare il contenuto della comunicazione), diverse tecniche restano comunque efficaci, poiché fanno leva su aspetti non direttamente influenzati dalla crittografia,

come la temporizzazione dei pacchetti o le caratteristiche complessive del traffico [WZFH14].

In conclusione, operare in un contesto controllato rende gli esperimenti riproducibili ed elimina eventuali fattori esterni che potrebbero influenzare i risultati. Allo stesso modo, evitare l'uso della crittografia consente di osservare senza offuscamenti il fenomeno oggetto di studio, permettendo un'analisi più chiara ed efficace delle tecniche di comunicazione nascosta. Le sezioni seguenti descrivono l'architettura di una possibile soluzione che soddisfa questi requisiti.

3.2 Descrizione del testbed

Per facilitare l'analisi dei tre *proof of concept* (*PoC*) implementati, garantendo uno scenario riproducibile, è stato realizzato un semplice ambiente virtualizzato, ispirato al problema del prigioniero presentato nella sezione 1.2.1: due nodi virtuali, Alice e Bob, possono comunicare tra loro passando attraverso un terzo nodo, Wendy, che agisce da *router*. Alice e Bob eseguono due istanze di un'applicazione di messaggistica realizzata in modo tale da occultare l'invio dei messaggi scambiati mediante l'impiego di diversi tipi di *CC*s, selezionabili attraverso i parametri forniti in input. Come nel modello originale, Wendy è in grado di osservare i messaggi scambiati tra Alice e Bob; ovvero, è in grado di catturare tutto il traffico che instrada, rendendolo disponibile in tempo reale per l'analisi immediata o la registrazione su disco.

3.2.1 Ambiente virtualizzato

I nodi virtuali sono stati realizzati utilizzando la suite di virtualizzazione a livello di sistema operativo *Docker*[1]. In particolare, Alice, Bob e Wendy sono tre *container* che eseguono un'immagine leggera[2] di *Linux*, sui quali è installato il software necessario per la simulazione di una comunicazione nascosta.

[1] `https://www.docker.com`
[2] In particolare, *Alpine Linux* (`https://www.alpinelinux.org/`).

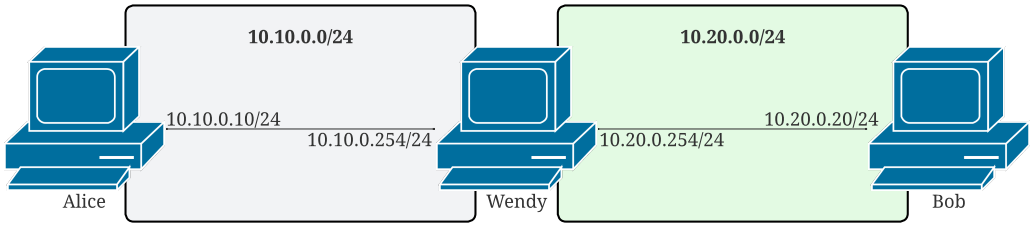


Figura 3.1: Topologia della rete del testbed.

Alice e Bob possono comunicare tra loro soltanto attraverso Wendy, che è configurato per effettuare l'inoltro dei pacchetti *IP* ricevuti. La semplice topologia della rete è mostrata in fig. 3.1.

La creazione e messa in opera del testbed è automatizzata mediante *Docker Compose*[3], uno strumento della *suite Docker* che permette di definire ed eseguire applicazioni *multi-container*. Un *Compose file*[4] specifica i servizi corrispondenti a ciascun nodo, la configurazione delle rispettive interfacce di rete e i parametri necessari per la comunicazione tra i *container*. In questa fase viene anche configurato il montaggio della cartella contenente lo script per la comunicazione nascosta, rendendola accessibile dai *filesystem* dei *container* Alice e Bob.

L'installazione dei pacchetti necessari e le operazioni di avvio di ogni *container* sono gestite dai rispettivi *Dockerfile*. Tra le altre cose, ogni nodo necessita di uno strumento (nel nostro caso `iproute2`) per la configurazione delle rotte statiche, la quale è effettuata da uno script *bash* che viene eseguito all'avvio.

Le due sottoreti `net_alice` (tra Alice e Wendy) e `net_bob` (tra Bob e Wendy) utilizzano il driver di default (`bridge`), che fornisce connettività a livello di rete (livello 3) tra i *container*. Questo significa che il traffico viene instradato tra i *container* tramite indirizzi *Internet Protocol* (*IP*), ma non è possibile gestire direttamente le trame di livello 2. Avremmo potuto utilizzare il driver `macvlan`, che invece lo permette, assegnando a ciascun *container* un indirizzo *Media Access Control* (*MAC*) univoco; tuttavia, come vedremo in sezione 3.2.3, per le tecniche che impiegheremo il driver `bridge` è sufficiente.

[3] `https://docs.docker.com/compose/`
[4] `https://docs.docker.com/reference/compose-file/`

La definizione degli indirizzi degli *host* e delle sottoreti è centralizzata in un file `.env`, a cui accedono sia *Compose* che gli script di configurazione.

3.2.2 Analisi del traffico

Il traffico generato da Alice e Bob viene catturato da Wendy mediante *tcpdump*[5], uno strumento da linea di comando che consente di catturare e salvare i pacchetti che transitano su una o più determinate interfacce di rete. Il *software* offre numerose opzioni per filtrare il traffico in base a protocolli, indirizzi, porte e altri criteri specifici; questo ci torna particolarmente utile, in quanto l'analisi è svolta all'interno del sistema operativo *host* (ovvero il sistema che ospita i *container*), ed è quindi necessario esporre i dati catturati da Wendy in qualche modo. Nel nostro caso, un'istanza di *netcat*[6] rimane in ascolto su una porta predefinita, attendendo una connessione *TCP* su cui inviare in tempo reale i dati raccolti. Tuttavia, se *tcpdump* fosse configurato per catturare tutto il traffico indiscriminatamente, catturerebbe anche i pacchetti generati dalla sua stessa connessione con il sistema operativo *host*, causando un inutile circolo vizioso. Per questo, al momento dell'avvio di *tcpdump* viene specificato un filtro che esclude la porta (nel nostro caso, 3000) su cui *netcat* è in ascolto, in modo da catturare solo il traffico generato da Alice e Bob. Il comando eseguito da Wendy per catturare il traffico è quindi il seguente:

```
tcpdump -i any -U -w - "not port 3000" | nc -l -p 3000
```

Nonostante *tcdump* offra diverse funzionalità di analisi del traffico, preferiamo affidarci al più versatile *Wireshark*[7], un noto strumento di analisi del traffico di rete, che permette di visualizzare, filtrare e analizzare i pacchetti catturati in modo interattivo; a tal fine, un'istanza del programma viene avviata sul sistema operativo *host* e configurata per leggere i dati dalla connessione con Wendy, così da monitorare in tempo reale la comunicazione tra Alice e Bob.

[5] `https://www.tcpdump.org/`
[6] `https://netcat.sourceforge.net/`
[7] `https://www.wireshark.org/`

3.2.3 Scelta delle tecniche

Innanzitutto, è bene chiedersi a quale livello della pila protocollare sia opportuno realizzare un *CC*.

Tipicamente, l'implementazione degli strati inferiori del modello *TCP/IP* è fornita direttamente dal sistema operativo, ed è virtualmente impossibile (salvo la presenza di vulnerabilità) modificarne le *PDU* senza disporre di privilegi elevati [SM07]. A livello applicativo, invece, l'utente è generalmente lasciato libero di instaurare connessioni in uscita e di gestire arbitrariamente il contenuto dei messaggi. Infatti, qualsiasi scambio di dati tra l'utente il livello di trasporto è, nel modello *TCP/IP*, considerato parte delle operazioni svolte dal livello applicativo, e non viene generalmente modificato da meccanismi di sicurezza. Di conseguenza, i canali nascosti implementati a livello applicativo dispongono del margine operativo più ampio possibile [Kwe06]. Inoltre, poiché la consegna dei messaggi generati dalle applicazioni può essere delegata a protocolli di trasporto affidabili, è possibile implementare canali nascosti di memorizzazione sostanzialmente esenti da rumore.

Tra i protocolli applicativi, i migliori candidati come vettori di *CC* sono i più comuni, in quanto la loro ampia adozione tende a ridurre il livello di sospetto che possono generare [DM17]. *HTTP* è uno dei protocolli più utilizzati per lo scambio di informazioni su Internet [GKS18], inoltre, il suo funzionamento si basa su messaggi testuali in chiaro, strutturati secondo un formato leggibile da un essere umano. Questo lo rende particolarmente adatto alla progettazione e analisi di canali nascosti al suo interno, grazie alla facilità con cui è possibile osservare e manipolare il contenuto dei messaggi senza necessità di trasformazioni complesse.

HTTP

HyperText Transfer Protocol (*HTTP*) è un protocollo di livello applicativo, che segue i paradigmi *client-server* e *request-response*, consentendo a un *client*, che avvia la comunicazione, di richiedere risorse quali ad esempio pagine *HTML*, file multimediali o dati in formato *JSON*, a un *server* in ascolto.

Più precisamente, *HTTP* fornisce un'interfaccia uniforme per interagire con una generica risorsa, a prescindere dal suo tipo, natura o implementazione, at-

traverso l'invio di messaggi che manipolano o trasferiscono sue rappresentazioni[8]. Ogni messaggio *HTTP* è o una richiesta o una risposta. Un *client* costruisce dei messaggi di richiesta che codificano (per mezzo di un insieme di metodi predefiniti) le sue intenzioni verso determinate risorse, e invia tali messaggi a un *server* in ascolto. Il *server* elabora le richieste ricevute e interpreta il significato del messaggio in relazione alla risorsa indicata, inviando uno o più messaggi di risposta al richiedente [FNR22a].

In principio, *HTTP* è un protocollo *stateless* (privo di stato), in cui ogni richiesta da parte di un *client* viene trattata dal *server* come indipendente dalle precedenti [FGM+99]; tuttavia, esistono meccanismi definiti da standard complementari, come gli *header Cookie* e *Set-Cookie*, che rendono possibili interazioni dotate di continuità tra richieste successive [Bar11].

L'estrema popolarità del protocollo ha comportato un continuo lavoro da parte dei progettisti per mantenere lo standard al passo con le crescenti richieste delle applicazioni *web*. *HTTP/1.0*[9] permetteva di avere una sola richiesta in corso per ogni connessione *TCP*; *HTTP/1.1* ha introdotto il meccanismo del *pipelining*, permettendo di inviare una serie di richieste prima di ricevere la prima risposta, ma questo non ha risolto del tutto i problemi prestazionali come l'*head-of-line blocking*[10], né ha affrontato l'inefficienza dovuta alla ripetizione degli *header* in ogni richiesta, che può causare congestione nelle fasi iniziali di una connessione *TCP* [BPT15]. *HTTP/2* ha mantenuto la semantica delle versioni precedenti, mettendo a disposizione un sistema ottimizzato per il trasporto di più *stream* di richieste e risposte in parallelo sulla stessa connessione *TCP*, oltre a un meccanismo di compressione degli *header* (*HPACK*) per ridurre l'*overhead* del protocollo [DM17]. La versione più recente, *HTTP/3.0*, supporta le stesse funzionalità appoggiandosi però a un nuovo protocollo di trasporto: *QUIC* [Bis22].

Concludiamo qui questa breve presentazione del protocollo; approfondiremo

[8]Per *HTTP* una rappresentazione è un insieme di informazioni che codificano lo stato passato, corrente o desiderato di una data risorsa in un formato trasferibile mediante i messaggi del protocollo [FR14].

[9]Il primo standard di *HTTP* non specificava un numero di versione e fu solo in seguito denominato HTTP/0.9 per distinguerlo dai successori. Si trattava di una versione primitiva del protocollo, che supportava richieste di una sola riga.

[10]Il fenomeno per cui una richiesta in attesa di essere elaborata blocca l'elaborazione di tutte le richieste successive nella coda, causando ritardi nel sistema.

alcuni aspetti del funzionamento di *HTTP/1.1* e *HTTP/2* nel corso della trattazione, contestualmente e quando utile all'analisi dei *PoC* sviluppati.

3.2.4 Messaggistica occulta

Il componente responsabile della generazione del traffico che incorpora i *CC*s è il *package Python* `covert_messenger`, il cui funzionamento è descritto in sezione 3.3.

Python[11] è un linguaggio di programmazione ad alto livello e interpretato. Un *package* è una *directory* che contiene un file `__init__.py` e uno o più moduli; un modulo è un singolo file `.py` che può definire funzioni, classi o variabili ed essere importato da altri script.

3.3 *Covert Messenger*

Il *package* `covert_messenger` (da ora in avanti, *Covert Messenger*) realizza un semplice applicativo di messaggistica da linea di comando, ed è strutturato secondo il paradigma architetturale *Model View Controller* (*MVC*). Lo *script* consente di instaurare una comunicazione nascosta tra due sistemi che siano in grado di scambiarsi messaggi *HTTP*, previa specifica dell'indirizzo *IP* dell'altro estremo come parametro di *input*. La struttura del *package* è mostrata in fig. 3.2.

Ogni *PoC* è realizzato da un *model* (modello) specifico, che si occupa di gestire la codifica e decodifica del messaggio secondo la logica specifica del *CC* che implementa. Una *View* (visualizzazione) gestisce l'interazione con l'utente e comunica con il *Model* attraverso il *Controller* (controllore), che agisce da *"message box"* ("casella di posta") mettendo a disposizione strutture dati sincrone per l'inserimento e il recupero dei messaggi in trasmissione e in ricezione.

In questo e nei prossimi capitoli, saranno mostrati alcuni frammenti di codice quando utile a chiarire l'implementazione descritta. In questi casi, il codice potrebbe essere presentato in forma alleggerita per evidenziare i concetti chiave, e non corrispondere riga per riga a quello effettivamente utilizzato. Ci riferiremo con il termine "estratto" a frammenti di codice che ricalcano l'originale, con l'omissione di alcune parti (commenti, gestione degli errori, *logging*, ecc.). Utilizzeremo il

[11] `https://www.python.org`

termine "parafrasi" per riferirci a pseudocodice che esemplifica il funzionamento del codice originale, semplificandone la struttura. Per una visione completa del codice, si rimanda al repository *GitHub*[12] del progetto.

3.3.1 L'infrastruttura

Il modulo `__main__.py` è il punto di ingresso del *package*, e si occupa di gestire l'avvio dell'applicazione. Il suo compito principale è quello di leggere i parametri di avvio (elencati nel listato 3.1), che possono essere passati tramite riga di comando, e di instanziare il *Controller* con gli indirizzi e le classi corrette a seconda dei comandi dell'utente.

```
usage: Covert Messenger [-h] -e {A,B,C} [-b] [-d] [--local-port LOCAL_PORT] [--
    remote-port REMOTE_PORT] remote_address

A didactic tool for showcasing covert channels

positional arguments:
  remote_address        Remote host address.

options:
  -h, --help            Show this help message and exit
  -e, --example {A,B,C}
                        Which of the examples to execute.
  -b, --basic-view      Display no visual cues. Useful for input/output
      redirection.
  -d, --debug           Enable debug log.
  --local-port LOCAL_PORT
                        Port on which the local server should listen.
  --remote-port REMOTE_PORT
                        Remote server port.
```

Listato 3.1: Le opzioni di avvio del *package* `covert_messenger`

Il *Controller* mette a disposizione due code sincrone (realizzate dalla classe `queue.Queue`), una per i messaggi in arrivo e una per quelli in uscita, nonché un evento di terminazione (`threading.Event`) che può essere utilizzato per chiudere l'applicazione in modo controllato.

La *View* riceve l'*input* dell'utente e lo consegna al *Controller* chiamando il metodo `put_message_to_send()`. Il *Model* potrà prelevare il messaggio dalla coda

[12] `https://github.com/fiocchidavide/Bachelors-Testbed`

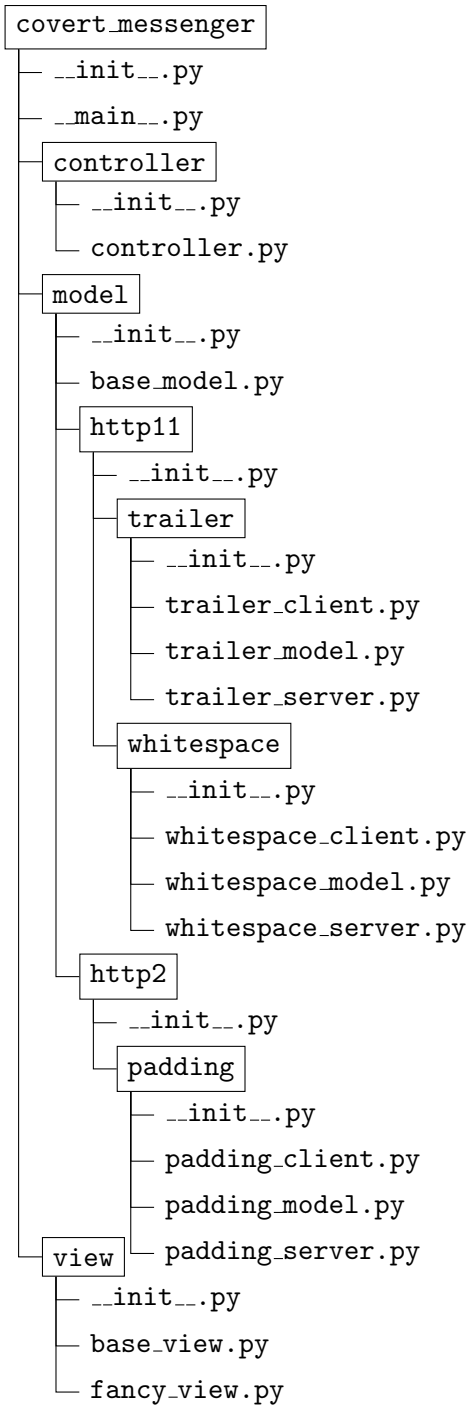


Figura 3.2: Struttura di *Covert Messenger*: i rettangoli rappresentano i *package*, mentre i file `.py` sono i moduli.

quando sarà pronto a inviarlo, tramite il metodo `get_message_to_send()`. Analogamente, in seguito alla ricezione di un messaggio, il *Model* lo consegna al *Controller* tramite il metodo `put_message_to_display()`, rendendolo disponibile alla *View* attraverso il metodo `get_message_to_display()`.

Tutti i *thread* in esecuzione controllano a ogni iterazione che l'utente non abbia richiesto la chiusura dell'applicazione, interrogando l'evento di terminazione attraverso il metodo `should_run()` del *Controller*.

Un sistema di *logging* permette di registrare le operazioni svolte dall'applicazione, e di visualizzare eventuali errori o avvisi, includendo informazioni più dettagliate se richiesto al momento dell'avvio (attraverso l'opzione `-d`).

3.3.2 I *Model*

Analizzeremo tre *proof of concept* di canali nascosti in *HTTP*. Le tecniche sono state selezionate in modo da rappresentare una progressione concettuale in termini di sofisticazione nell'ambito dei *CC* di tipo *storage*: i primi due esempi sfruttano le *PDU* di *HTTP/1.1*, e sono di carattere maggiormente didattico ed esemplificativo, mentre il terzo esempio cresce in complessità e realismo, facendo uso di *HTTP/2* e di un meccanismo più furtivo.

Tutti e tre i *PoC* implementati sono basati su canali di comunicazione unidirezionali tra un *server* e un *client HTTP*[13]. Il *server* resta in attesa di richieste *HTTP* da parte di un *client* e, quando interpellato, verifica la presenza di dati nascosti da inviare, eventualmente codificandoli nella risposta in modo appropriato a seconda della particolare tecnica impiegata. La comunicazione avviene quindi attraverso un meccanismo di *polling*, in cui la responsività della comunicazione è determinata dalla frequenza di interrogazione del *server* da parte del *client*.

Per realizzare una comunicazione bidirezionale, è necessario che gli attori facciano uso in contemporanea di due canali, come mostrato in fig. 3.3; ovvero, ciascuno dei due nodi deve eseguire sia un *server* che invia dati, che un *client* che li riceve.

I *Covert Channel*s realizzati non prevedono la presenza di intermediari nella comunicazione, e sono quindi di tipo **attivo**.

[13] L'architettura è quella proposta in [GKS18].

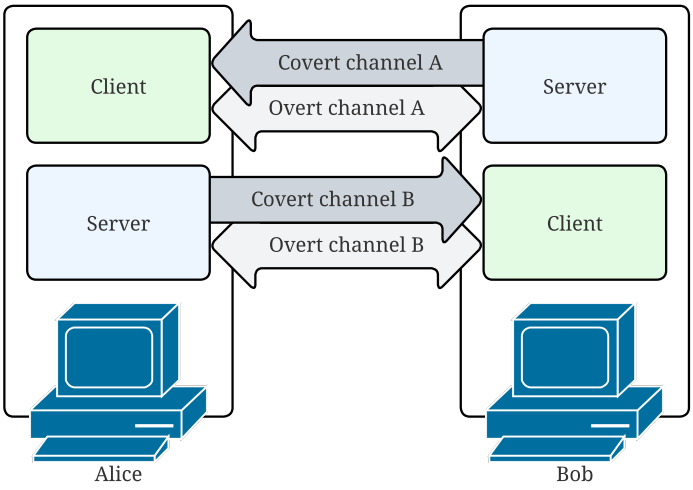


Figura 3.3: Realizzazione di una comunicazione bidirezionale attraverso il modello proposto [GKS18].

Implementazione comune

Ognuno dei tre *model* estende la classe `BaseModel`, che descrive il comportamento comune a tutti i modelli. Nel listato 3.2 possiamo vedere come questa classe si occupi di instanziare un *client HTTP* per la ricezione di messaggi nascosti e un *server* per la loro trasmissione, nonché di gestire il ciclo di vita dei relativi *thread*.

```
class BaseModel(Thread):
  def __init__(self, controller, server_class, client_class, ...):
    self.controller = controller
    self.server = server_class(...)
    self.client = client_class(...)

  def run(self):
    self.server.start()
    self.client.start()

    self.controller.wait_for_stop()

    self.server.join()
    self.client.join()
```

Listato 3.2: Parafrasi della classe `BaseModel`

Capitolo 4

Esempio A: *Trailer*

Questa tecnica, descritta in [GKS18], si basa sull'utilizzo dell'*header* `Trailer` di *HTTP*. Per comprenderne meglio la natura e il funzionamento, è necessario introdurre alcuni concetti sul funzionamento del protocollo.

4.1 Cenni su *HTTP*

4.1.1 La struttura dei messaggi

Sia i messaggi *HTTP* di richiesta (`Request`) che di risposta (`Response`) aderiscono al formato generico riportato nel listato 4.1, che prevede una riga iniziale, zero o più campi di intestazione (*header fields* o *headers*), una riga vuota (ovvero contenente solo i caratteri `CR` e `LF`) a indicare il termine dell'intestazione e, opzionalmente, il corpo del messaggio (*message body*) [FGM+99].

```
1 generic-message = start-line
2                   *(message-header CRLF)
3                   CRLF
4                   [ message-body ]
5
6 start-line      = Request-Line | Status-Line
7 message-header  = field-name ":" [ field-value ]
```

Listato 4.1: La struttura di un generico messaggio *HTTP* [FGM+99].

Ogni *header* è composto da un nome (non *case-sensitive*) e un valore, separati dai due punti (`:`). Esistono diversi tipi di *header*, con funzioni e significati diversi, tra cui [FGM+99]:

- gli *header* generali, validi sia per richieste che per risposte;
- gli *header* di richiesta, che forniscono informazioni aggiuntive al *server* (come l'identità del *client* o le sue preferenze sul formato);
- gli *header* di risposta, utilizzati dal *server* per descrivere la risposta inviata;
- gli *header* di entità, che specificano metadati relativi al corpo del messaggio, come la lunghezza o il tipo di contenuto.

4.1.2 Le codifiche di trasporto

I *transfer codings* (codifiche di trasporto) sono trasformazioni applicate al contenuto di un messaggio *HTTP* per ottenere dei vantaggi in termini di efficienza o sicurezza del trasporto attraverso la rete. L'*header* generale `Transfer-Encoding` viene utilizzato per elencare i nomi dei *transfer codings* che sono stati o saranno applicati al contenuto per formare il corpo del messaggio.

Il *transfer coding* "`chunked`" (letteralmente "a blocchi") incapsula il contenuto per trasmetterlo come una serie di blocchi, con la possibilità di specificare per ciascuno una dimensione variabile. Questo consente di trasferire flussi di contenuto la cui dimensione risulta sconosciuta al momento dell'inizio della trasmissione, suddividendoli in sequenze di buffer chiaramente delimitati [FNR22b].

Nel listato 4.2 vediamo come il corpo di un messaggio `chunked` sia seguito da una `trailer-section`. Questa sezione permette al mittente di apporre ulteriori campi in seguito all'invio del corpo, ad esempio per fornire informazioni sul contenuto che non erano disponibili all'inizio della trasmissione[1].

[1] Un esempio (proposto in [FNR22a]) è quello di una firma calcolata su dati che stanno ancora venendo generati e inviati.

```
chunked-body   = *chunk
                 last-chunk
                 trailer-section
                 CRLF

chunk          = chunk-size [ chunk-ext ] CRLF
                 chunk-data CRLF
chunk-size     = 1*HEXDIG
last-chunk     = 1*("0") [ chunk-ext ] CRLF

chunk-data     = 1*OCTET ; a sequence of chunk-size octets
```

Listato 4.2: La struttura del corpo di un messaggio `chunked` [FNR22b].

4.1.3 Il campo *Trailer*

L'*header* "`Trailer`" può essere aggiunto all'intestazione di un messaggio per indicare la lista dei nomi dei campi che il mittente prevede di inviare all'interno della `trailer-section` del messaggio [FNR22a].

La possibilità di utilizzare questo campo allo scopo di nascondere informazioni è dovuta al largo grado di libertà che le specifiche lasciano. Infatti, lo standard non richiede che il campo sia necessariamente presente, ma semplicemente **suggerisce** di includerlo per aiutare il destinatario nell'interpretazione del messaggio; inoltre non è richiesto né che il mittente invii effettivamente i campi specificati, né che il destinatario li accetti [FNR22a].

È interessante osservare come nessuna *RFC* stabilisca esplicitamente che l'invio di un *header* `Trailer` senza che il messaggio utilizzi il *transfer coding* "`chunked`" sia da considerare un errore. Tuttavia diverse implementazioni di server *HTTP* (ad esempio *Node.js*[2]) trattano tali messaggi come malformati, rifiutandoli. Possiamo quindi concludere che, al fine di massimizzare la furtività del *CC*, sia opportuno impiegare il *transfer coding* "`chunked`" quando si utilizza l' *header* `Trailer`, in quanto questo costituisce lo standard *de facto*.

[2] `https://nodejs.org/api/errors.html#err_http_trailer_invalid`

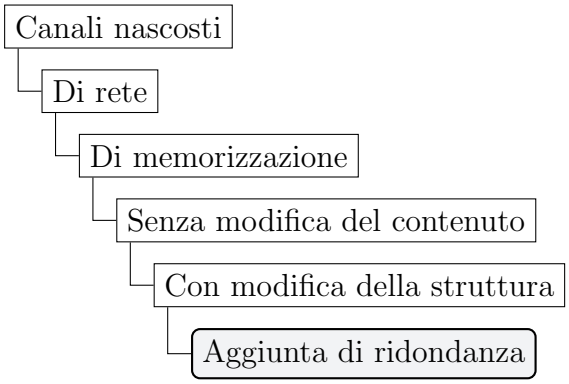


Figura 4.1: Classificazione del *Covert Channel* realizzato attraverso l'*header* `Trailer` nel contesto della tassonomia individuata da Wendzel et al. (fig. 2.1).

4.2 Classificazione del *Covert Channel*

Il canale realizzato attraverso questa tecnica nasconde informazioni all'interno del valore del *header* `Trailer`, quindi è un **canale di memorizzazione**. Ciononostante, non modifica il contenuto del messaggio, ma solo la sua intestazione, quindi è **senza modifica del contenuto**. Infine, durante la codifica delle informazioni si **modifica la struttura** del messaggio, in quanto è necessario l'utilizzo del *transfer coding* "`chunked`"[3] e l'aggiunta di un campo all'intestazione. Non essendo alcuna di queste operazioni richiesta normalmente, il canale appartiene al pattern **"aggiunta di ridondanza"** (vedi fig. 4.1) [GKS18, WZFH14].

4.3 Implementazione

Questo *PoC* fa affidamento sul *package Python* `http`, una libreria per la creazione di semplici *server* e *client HTTP*. La copertura per la comunicazione nascosta è rappresentata da una semplice pagina *HyperText Markup Language* (*HTML*) che mostra l'ora corrente, ed è accessibile attraverso una richiesta `GET` alla risorsa `/time`.

[3]Fatta eccezione per situazioni in cui si faccia già uso della codifica `chunked` per altre ragioni.

4.3.1 *Server*

L'implementazione di `TrailerServer` si basa su `http.server.HTTPServer`, una classe che si occupa di gestire la comunicazione di rete tramite un *socket TCP*, ascoltando su un indirizzo e una porta specificati, e delegando la gestione delle richieste *HTTP* a un apposito *handler*. Nel contesto della gestione dei *thread* in *Python*, il metodo *run* rappresenta il punto in cui viene definito il codice che verrà effettivamente eseguito all'avvio del *thread*. Il metodo `TrailerServer.run` è riportato nel listato 4.3: vediamo come, una volta avviato, il *server* rimanga in attesa di richieste da parte dei *client*, delegandone l'elaborazione al `RequestHandler` fornito nel costruttore.

Per modificare il comportamento del *server*, è quindi sufficiente definire una classe che estenda `BaseHTTPRequestHandler`, contenente i metodi `do_METODO` (dove `METODO` è il metodo *HTTP* da implementare, ad esempio `GET` o `POST`) che siamo interessati a supportare. Nel nostro caso, ci interessa solamente che il *server* risponda alle richieste `GET`, in particolare quelle indirizzate alla risorsa `/time`; ci basterà quindi definire il solo metodo `do_GET`, all'interno del quale abbiamo la possibilità di specificare come il nostro messaggio segreto debba essere incorporato all'interno del traffico lecito. Questa operazione è svolta dal metodo `build_handler_class`, visibile nel listato 4.4.

Come anticipato, in caso sia necessario codificare un messaggio nascosto, è opportuno impiegare il *transfer coding* "`chunked`". Per fare ciò, è sufficiente impostare l'*header* `Transfer-Encoding` del messaggio di risposta a "`chunked`", e inviare il corpo del messaggio come una serie di blocchi, ciascuno preceduto dalla sua dimensione in esadecimale. La funzione `make_chunk`, riportata nel listato 4.5, si occupa di generare un blocco[4] a partire da un oggetto `bytes`.

4.3.2 *Client*

L'implementazione di `TrailerClient` non presenta particolari complessità, e si limita ad utilizzare le funzionalità offerte dal modulo `http.client` per effettuare una richiesta `GET` alla risorsa `/time` del server a intervalli di tempo scelti casualmente. Vediamo nel listato 4.6 come la risposta del server venga quindi analizzata

[4] Vedi listato 4.2 per il formato.

```
def run(self):

  server = HTTPServer(("0.0.0.0", self.port), self.build_handler_class())

  while self.should_run():
    server.handle_request()
```

Listato 4.3: Un estratto del metodo `TrailerServer.run`

```
def build_handler_class(self):
  class TrailerAddingHTTPRequestHandler(BaseHTTPRequestHandler):
    def do_GET(_self):
      if _self.path == "/time":
        _self.send_response(200)
        _self.send_header("Content-Type", "text/html; charset=utf-8")

        # Check if any message is available to send
        try:
          message = self.produce_message()
        except queue.Empty:
          message = None

        # If we are sending a Trailer, we need to send a chunked body
        if message:
          # Specify the chunked transfer coding
          _self.send_header("Transfer-Encoding", "chunked")

          # Encode the massage as value of Trailer header
          _self.send_header("Trailer", message)

          # Encode the response into a chunked body
          body = self.make_chunk(build_response()) + self.make_chunk(None)
        # If we are not, we can send a regular body
        else:
          body = build_response()

        _self.end_headers()
        _self.wfile.write(body)

  return TrailerAddingHTTPRequestHandler
```

Listato 4.4: Un estratto del metodo `TrailerServer.build_handler_class`

```
@staticmethod
def make_chunk(message):
    if message is None:
        return b"0\r\n\r\n"
    else:
        length = len(message)
        return f"{length:x}\r\n".encode("utf-8") + message + b"\r\n"
```

Listato 4.5: Un estratto del metodo `TrailerServer.make_chunk`

```
def run(self):
  while self.should_run():
    # Generate a random delay based on a normal distribution
    seconds = max(0, random.normalvariate(REQUEST_TIME_MEAN,
        REQUEST_TIME_STD_DEV))
    time.sleep(seconds)

    conn = http.client.HTTPConnection(self.remote_address, self.remote_port)
    conn.request("GET", "/time")
    response = conn.getresponse()
    trailer_value = response.msg.get("Trailer")

    if trailer_value:
      # Calls controller.put_message_to_display
      self.accept_message(trailer_value)
```

Listato 4.6: Parafrasi del metodo `TrailerClient.run`

per determinare la presenza di un messaggio nascosto il quale, se presente, viene passato al `controller` per la visualizzazione.

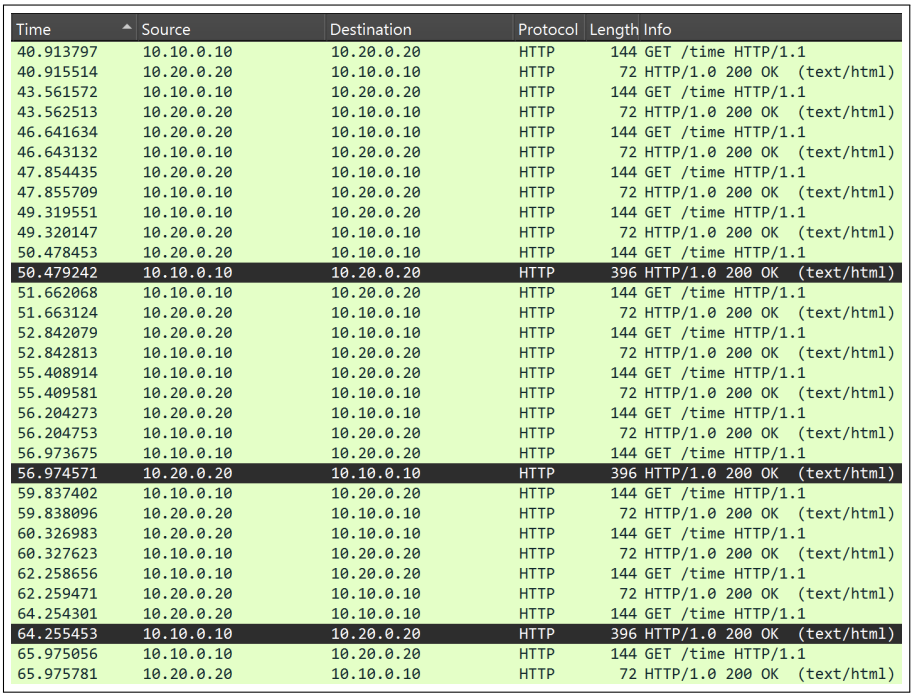
4.4 Analisi del traffico

Per prima cosa, introduciamo il semplice esempio di conversazione che utilizzeremo per analizzare il traffico generato da ciascuna tecnica. Lo scambio di battute è quello riportato in fig. 4.2, in cui Bob e Alice si mettono d'accordo su un possibile orario per l'evasione. Le righe che cominciano con `$` rappresentano l'input dell'utente, mentre il carattere `>` indica un messaggio ricevuto.

(a) Il terminale di Alice

(b) Il terminale di Bob

Figura 4.2: La conversazione analizzata.



Time	Source	Destination	Protocol	Length	Info
40.913797	10.10.0.10	10.20.0.20	HTTP	144	GET /time HTTP/1.1
40.915514	10.20.0.20	10.10.0.10	HTTP	72	HTTP/1.0 200 OK (text/html)
43.561572	10.10.0.10	10.20.0.20	HTTP	144	GET /time HTTP/1.1
43.562513	10.20.0.20	10.10.0.10	HTTP	72	HTTP/1.0 200 OK (text/html)
46.641634	10.20.0.20	10.10.0.10	HTTP	144	GET /time HTTP/1.1
46.643132	10.10.0.10	10.20.0.20	HTTP	72	HTTP/1.0 200 OK (text/html)
47.854435	10.20.0.20	10.10.0.10	HTTP	144	GET /time HTTP/1.1
47.855709	10.10.0.10	10.20.0.20	HTTP	72	HTTP/1.0 200 OK (text/html)
49.319551	10.10.0.10	10.20.0.20	HTTP	144	GET /time HTTP/1.1
49.320147	10.20.0.20	10.10.0.10	HTTP	72	HTTP/1.0 200 OK (text/html)
50.478453	10.20.0.20	10.10.0.10	HTTP	144	GET /time HTTP/1.1
50.479242	10.10.0.10	10.20.0.20	HTTP	396	HTTP/1.0 200 OK (text/html)
51.662068	10.10.0.10	10.20.0.20	HTTP	144	GET /time HTTP/1.1
51.663124	10.20.0.20	10.10.0.10	HTTP	72	HTTP/1.0 200 OK (text/html)
52.842079	10.20.0.20	10.10.0.10	HTTP	144	GET /time HTTP/1.1
52.842813	10.10.0.10	10.20.0.20	HTTP	72	HTTP/1.0 200 OK (text/html)
55.408914	10.10.0.10	10.20.0.20	HTTP	144	GET /time HTTP/1.1
55.409581	10.20.0.20	10.10.0.10	HTTP	72	HTTP/1.0 200 OK (text/html)
56.204273	10.20.0.20	10.10.0.10	HTTP	144	GET /time HTTP/1.1
56.204753	10.10.0.10	10.20.0.20	HTTP	72	HTTP/1.0 200 OK (text/html)
56.973675	10.10.0.10	10.20.0.20	HTTP	144	GET /time HTTP/1.1
56.974571	10.20.0.20	10.10.0.10	HTTP	396	HTTP/1.0 200 OK (text/html)
59.837402	10.10.0.10	10.20.0.20	HTTP	144	GET /time HTTP/1.1
59.838096	10.20.0.20	10.10.0.10	HTTP	72	HTTP/1.0 200 OK (text/html)
60.326983	10.20.0.20	10.10.0.10	HTTP	144	GET /time HTTP/1.1
60.327623	10.10.0.10	10.20.0.20	HTTP	72	HTTP/1.0 200 OK (text/html)
62.258656	10.10.0.10	10.20.0.20	HTTP	144	GET /time HTTP/1.1
62.259471	10.20.0.20	10.10.0.10	HTTP	72	HTTP/1.0 200 OK (text/html)
64.254301	10.20.0.20	10.10.0.10	HTTP	144	GET /time HTTP/1.1
64.255453	10.10.0.10	10.20.0.20	HTTP	396	HTTP/1.0 200 OK (text/html)
65.975056	10.10.0.10	10.20.0.20	HTTP	144	GET /time HTTP/1.1
65.975781	10.20.0.20	10.10.0.10	HTTP	72	HTTP/1.0 200 OK (text/html)

Figura 4.3: I messaggi *HTTP* prodotti dalla conversazione tra Alice e Bob. In nero i messaggi che utilizzano la codifica di trasporto "`chunked`".

Possiamo intepretare il traffico catturato da Wendy come una sequenza di messaggi *HTTP* utilizzando i *dissectors* messi a disposizione da *Wireshark*. I *dissectors* sono componenti specializzati che analizzano le *PDU* catturate e ne interpretano il contenuto sulla base dei protocolli di rete che utilizzano, presentando le informazioni in modo leggibile e strutturato.

Isoliamo il traffico generato dalla conversazione tra Alice e Bob impostando il filtro di visualizzazione di *Wireshark* su "`http`". Il risultato è mostrato in fig. 4.3, dove possiamo notare diverse richieste `GET` per la risorsa `/time` e risposte `200 OK` tra i due nodi, generate a intervalli di tempo regolari leggermente variabili. Questi messaggi corrispondono alle richieste generate dai due client in esecuzione sulle macchine di Alice e Bob e alle risposte prodotte dai rispettivi server. Possiamo selezionare le risposte che fanno utilizzo della codifica di trasporto "`chunked`" attraverso il filtro: `http.transfer_encoding == "chunked"`. Colorando diversamente (in nero) i messaggi che soddisfano questa condizione, abbiamo una panoramica di quali siano i messaggi che trasportano informazione nascosta.

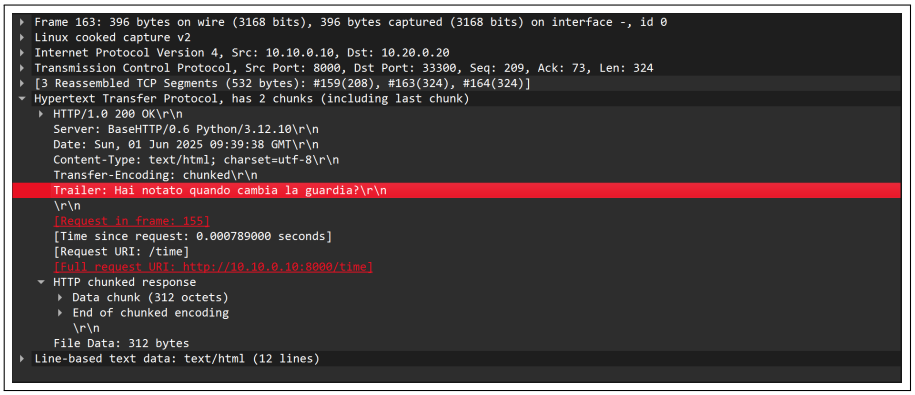


Figura 4.4: Il primo messaggio *HTTP* a trasportare informazione nascosta durante la conversazione.

Spostando l'attenzione sul primo messaggio colorato in questo modo, possiamo notare che è stato inviato dal server di Alice (`10.10.0.10`) al client di Bob (`10.20.0.20`). Come ci aspettiamo, ispezionando il messaggio possiamo notare in fig. 4.4 la presenza dell'*header* `Trailer`, con valore uguale al primo messaggio inviato da Alice.

Capitolo 5

Esempio B: *Whitespace*

Normalmente, le transazioni *HTTP* avvengono in maniera trasparente per l'utente, il quale potrebbe tranquillamente svolgere le proprie attività senza notarne l'avvenimento. Ciononostante, un'implementazione accorta di *CC* dovrebbe, sotto determinate ipotesi, passare inosservata anche a seguito di un'eventuale ispezione visiva dei messaggi che lo veicolano. Un *CC* simile è descritto in [Kwe06], e si basa sull'utilizzo degli spazi bianchi (*whitespace*) all'interno dei valori degli header di un messaggio *HTTP*.

5.1 Il formato degli *header HTTP*

La tecnica è resa possibile dal fatto che *HTTP/1.1* consideri qualsiasi numero di caratteri "spazio bianco" come equivalenti a un singolo spazio. Infatti, come riporta la *rfc* 2616 [FGM+99]:

> *"Ogni sequenza di spazi bianchi contigui [Linear White Space, LWS], compreso il folding*[1]*, ha lo stesso significato di un singolo SP."*

Quindi, completando la definizione data nel listato 4.1, abbiamo che il formato di un *header HTTP* è quello riportato nel listato 5.1.

[1] Il *folding* è un meccanismo che consente di spezzare il valore di un *header* su più righe, a patto di far cominciare ogni riga di continuazione con un carattere `SP` o `HT`.

```
message-header = field-name ":" [ field-value ]
field-name     = token
field-value    = *( field-content | LWS )
field-content  = <the OCTETs making up the field-value
                 and consisting of either *TEXT or combinations
                 of token, separators, and quoted-string>
```

Listato 5.1: Il formato ammissibile di un *header HTTP* [FGM+99].

In particolare, il `field-content` è implicitamente assunto privo di spazi bianchi iniziali o finali, i quali (se presenti, come concesso dallo standard) possono essere rimossi senza alterarne il significato. Inoltre, qualsiasi numero di spazi bianchi contigui inframezzati al `field-content` è considerato equivalente a un singolo spazio, e può essere collassato prima di interpretare il valore del campo o di inoltrare il messaggio a valle [FGM+99].

Abbiamo quindi la possibilità di codificare informazioni utilizzando combinazioni arbitrarie di caratteri "spazio bianco" all'interno del valore di un *header HTTP*. Ad esempio, il messaggio del listato 5.2 verrebbe trattato da una normale implementazione di *HTTP* come una richiesta valida e priva di errori. Tuttavia, un *server* configurato in modo tale da non ignorare gli spazi bianchi potrebbe, ad esempio, interpretare la sequenza inserita in coda al valore dell'*header* `Host` come un numero binario, in cui `SP` è usato per rappresentare lo zero e `HT` per rappresentare l'uno. In questo modo, l'informazione nascosta sarebbe interpretata come il numero binario $1100001_2 = 97_{10}$, equivalente al carattere "`a`" in codifica `ASCII`.

Risulta quindi evidente come due interlocutori, concordando preventivamente un protocollo, potrebbero sfruttare questo meccanismo per scambiarsi messaggi nascosti di lunghezza arbitraria.

```
GET[SP]/[SP]HTTP/1.1[CR][LF]
Host:[SP]www.unibo.it[HT][HT][SP][SP][SP][SP][HT][CR][LF]
[CR][LF]
```

Listato 5.2: Un esempio di richiesta *HTTP* in cui gli spazi bianchi sono stati evidenziati.

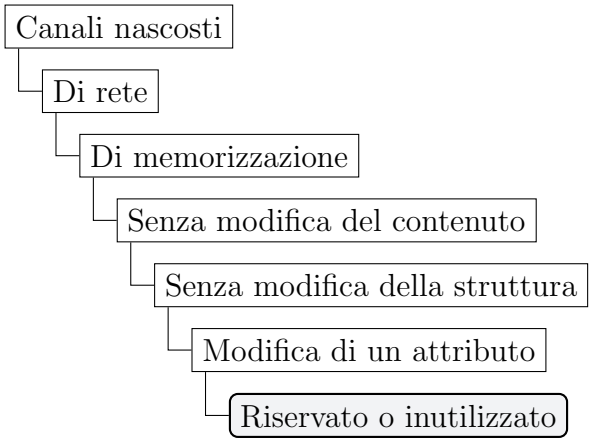


Figura 5.1: Classificazione del *Covert Channel* che sfrutta i caratteri bianchi negli *header* nel contesto della tassonomia individuata da Wendzel et al. (fig. 2.1).

5.2 Classificazione del *Covert Channel*

Il canale appena descritto, similmente a quello analizzato nell'Esempio A, è un **canale di memorizzazione senza modifica del contenuto**. Tuttavia, in questo caso **non abbiamo alcuna modifica alla struttura (sintassi) del messaggio**. Quello che avviene, invece è la **modifica di un attributo**, ovvero del valore di un *header* già presente. In particolare sfruttiamo un elemento (gli spazi finali del valore di un *header*) che non viene normalmente utilizzato per codificare informazione, quindi la tecnica ricade nel pattern **"riservato o inutilizzato"** (vedi fig. 5.1) [WZFH14].

5.3 Implementazione

Come dicevamo, lo standard prevede che ogni nodo della rete conforme alle specifiche ignori qualsiasi sequenza di spazi bianchi contigui all'interno del valore di un *header HTTP*, collassandoli in un singolo spazio. Di conseguenza, non potremo affidarci a implementazioni "*off-the-shelf*" di *server* e *client HTTP* per la realizzazione del *PoC*, in quanto le interfacce messe a disposizione da questi strumenti operano a un livello di astrazione troppo alto. Le manipolazioni consentite tramite le relative *Application Programming Interface* (*API*) avvengono infatti in seguito a un'elaborazione preliminare dei messaggi, durante la quale il contenu-

to “non standard” (sebbene tecnicamente conforme) si va a perdere a monte di troncature e normalizzazioni, risultando quindi inutilizzabile ai fini della tecnica proposta. Allo scopo di avere completo controllo sulle trasformazioni applicate alle *PDU*, utilizzeremo un *server* e un *client HTTP* rudimentali, utili unicamente come strumenti dimostrativi.

5.3.1 *Server*

La classe `WhitespaceServer` implementa un *server HTTP* che risponde in modo seriale alle richieste ricevute. Il metodo `run` (listato 5.3) si occupa di creare un socket *TCP*, e di metterlo in ascolto sulla porta specificata nel costruttore. In seguito, il *server* entra in un ciclo che termina solo su richiesta dell'utente, durante il quale rimane in attesa della ricezione di una richiesta da parte di un client.

```
def run(self):
  with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.bind(('0.0.0.0', self.port))
    s.listen(1)

    while self.should_run():
      conn, addr = s.accept()
      with conn:
        data = conn.recv(4096)
        request = data.decode(errors='ignore')
        response = self.handle_request(request)
        if response:
          conn.sendall(response)
```

Listato 5.3: Un estratto del metodo `WhitespaceServer.run`

Il metodo `handle_request` (listato 5.4) si occupa di elaborare le richieste ricevute e produrre una risposta. Il *server* gestisce solo le richieste `GET` per la risorsa `/time`, qualsiasi richiesta con metodo o risorsa diversi genererà una risposta `405 Method Not Allowed` o `404 Not Found`. In caso di richiesta valida, il *server* controlla la disponibilità di un messaggio da inviare, che se presente sarà inserito in forma codificata in coda al valore dell'header `Content-Lenght` (la scelta dell'header da utilizzare è arbitraria).

La codifica è realizzata dal metodo `message_to_whitespace` (listato 5.5), che riceve in input un messaggio e lo codifica in una sequenza di caratteri *“non printa-*

ble” (“non stampabili”, invisibili). Il processo di codifica consiste nell’ottenimento della rappresentazione binaria del messaggio, seguito dalla sostituzione di ogni bit con il carattere corrispondente. In particolare, scegliamo di far corrispondere lo zero al carattere `SP` (spazio) e l’uno al carattere `HT` (tabulazione).

```
def handle_request(self, request):
  # Attempt to extract the HTTP method and path from the request's first line.
  try:
    first_line = request.splitlines()[0]
    method, path, _ = first_line.split()
  except:
    return None

  # Only reply if the path matches the expected endpoint.
  if path == "/time":
    # Only GET requests are supported.
    if method.upper() == "GET":
      try:
        # Attempt to retrieve a message to encode; if none, message is None.
        message = self.produce_message()
      except queue.Empty:
        message = None

      body = build_response()
      # Covert message is appended to Content-Length header.
      response = (
        "HTTP/1.1 200 OK\r\n"
        "Content-Type: text/html; charset=utf-8\r\n"
        f"Content-Length: {len(body)}{self.message_to_whitespace(message) if
            message else ''}\r\n"
        "Connection: close\r\n"
        "\r\n"
      ).encode() + body
    else:
      # 405 Method Not Allowed
      response = ...
  else:
    # 404 Not Found
    response = ...

  return response
```

Listato 5.4: Un estratto del metodo `WhitespaceServer.handle_request`

```
def message_to_whitespace(self, message):
  # Convert string to its binary representation
  binary = ''.join(format(ord(c), '08b') for c in message)

  # Convert binary string to whitespace string
  return ''.join('\t' if bit == '1' else ' ' for bit in binary)
```

Listato 5.5: Un estratto del metodo `WhitespaceServer.message_to_whitespace`

```
def run(self):
  while self.should_run():
    # Sleep for a random amount of time
    seconds = max(0, random.normalvariate(3, 1))
    time.sleep(seconds)

    request = (
      f"GET /time HTTP/1.1\r\n"
      f"Host: {self.remote_address}:{self.remote_port}\r\n"
      f"Accept: identity\r\n"
      f"\r\n"
    )

    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
      s.connect((self.remote_address, self.remote_port))
      s.sendall(request.encode())
      response = s.recv(4096).decode()

      message = self.extract_covert_data(response)
      if message:
        self.accept_message(message)
```

Listato 5.6: Parafrasi del metodo `WhitespaceClient.run`

5.3.2 *Client*

`WhitespaceClient` si comporta in modo simile al client dell'Esempio A: il metodo `run` (listato 5.6) si occupa di produrre richieste `GET` per la risorsa `/time` a intervalli di tempo casuali. La risposta del server viene analizzata per determinare la presenza di un messaggio nascosto, che se presente viene consegnato al controller per la visualizzazione.

Il metodo `extract_covert_data` è responsabile della ricerca di eventuale informazione nascosta all'interno della risposta del *server*. Il processo consiste nell'identificare la riga contenente l'*header* `Content-Length`, e nell'estrarre l'eventuale sequenza di caratteri `SP` e `HT` che la termina. L'implementazione relativa è riportata nel listato 5.7.

```
def extract_covert_data(self, response):
  lines = response.split('\r\n')
  trailing_whitespace = ''

  # Look for the 'Content-Length' header and extract any trailing whitespace.
  for line in lines:
    if line.startswith("Content-Length:"):
      # Traverse backwards to collect the whitespace at the end of the header.
      for c in reversed(line.rstrip('\r\n')):
        if c in (' ', '\t'):
          trailing_whitespace = c + trailing_whitespace
        else:
          break

  if trailing_whitespace:
    return self.whitespace_to_message(trailing_whitespace)
  else:
    return None
```

Listato 5.7: Un estratto del metodo `WhitespaceClient.extract_covert_data`

```
def whitespace_to_message(self, whitespace):
  # Convert whitespace to binary string
  binary = ''.join('1' if c == '\t' else '0' for c in whitespace)

  # Split binary string into 8-bit chunks and convert to characters
  chars = [chr(int(binary[i:i+8], 2)) for i in range(0, len(binary), 8)]
  return ''.join(chars)
```

Listato 5.8: Un estratto del metodo `WhitespaceClient.whitespace_to_message`

Sapendo come sono codificati i messaggi, la decodifica ripercorre semplicemente i passaggi eseguiti, ma a ritroso. Il metodo `whitespace_to_message` (listato 5.8) traduce i caratteri non stampabili in una sequenza di *bit*, e successivamente abbina a ogni ottetto di *bit* il carattere corrispondente in *ASCII*[2].

5.4 Analisi del traffico

Vediamo ora come si presenta il traffico generato dalla conversazione descritta in sezione 4.4 quando trasmessa attraverso il *CC* appena affrontato.

[2]In realtà, le funzioni `chr` e `ord` di *Python* traducono i caratteri da e verso gli interi che rappresentano i *code point Unicode* corrispondenti, tuttavia per i nostri scopi (trasmettere caratteri semplici) non fa differenza, in quanto *ASCII* rappresenta un "sottoinsieme" di *Unicode*.

Time	Source	Destination	Protocol	Length Info
57.328651	10.10.0.10	10.20.0.20	HTTP	135 GET /time HTTP/1.1
57.329368	10.20.0.20	10.10.0.10	HTTP	483 HTTP/1.1 200 OK (text/html)
58.407637	10.10.0.10	10.20.0.20	HTTP	135 GET /time HTTP/1.1
58.408038	10.20.0.20	10.10.0.10	HTTP	483 HTTP/1.1 200 OK (text/html)
59.085535	10.20.0.20	10.10.0.10	HTTP	135 GET /time HTTP/1.1
59.085867	10.10.0.10	10.20.0.20	HTTP	483 HTTP/1.1 200 OK (text/html)
59.556060	10.10.0.10	10.20.0.20	HTTP	135 GET /time HTTP/1.1
59.556466	10.20.0.20	10.10.0.10	HTTP	483 HTTP/1.1 200 OK (text/html)
61.350969	10.20.0.20	10.10.0.10	HTTP	135 GET /time HTTP/1.1
61.352041	10.10.0.10	10.20.0.20	HTTP	771 HTTP/1.1 200 OK (text/html)
63.747452	10.10.0.10	10.20.0.20	HTTP	135 GET /time HTTP/1.1
63.747985	10.20.0.20	10.10.0.10	HTTP	483 HTTP/1.1 200 OK (text/html)
64.158435	10.20.0.20	10.10.0.10	HTTP	135 GET /time HTTP/1.1
64.158703	10.10.0.10	10.20.0.20	HTTP	483 HTTP/1.1 200 OK (text/html)
67.281546	10.20.0.20	10.10.0.10	HTTP	135 GET /time HTTP/1.1
67.282089	10.10.0.10	10.20.0.20	HTTP	483 HTTP/1.1 200 OK (text/html)
67.946185	10.10.0.10	10.20.0.20	HTTP	135 GET /time HTTP/1.1
67.946648	10.20.0.20	10.10.0.10	HTTP	723 HTTP/1.1 200 OK (text/html)
70.601615	10.10.0.10	10.20.0.20	HTTP	135 GET /time HTTP/1.1
70.602059	10.20.0.20	10.10.0.10	HTTP	483 HTTP/1.1 200 OK (text/html)
73.286167	10.20.0.20	10.10.0.10	HTTP	135 GET /time HTTP/1.1
73.286976	10.10.0.10	10.20.0.20	HTTP	675 HTTP/1.1 200 OK (text/html)
73.641113	10.10.0.10	10.20.0.20	HTTP	135 GET /time HTTP/1.1
73.641668	10.20.0.20	10.10.0.10	HTTP	483 HTTP/1.1 200 OK (text/html)
75.010818	10.10.0.10	10.20.0.20	HTTP	135 GET /time HTTP/1.1
75.011400	10.20.0.20	10.10.0.10	HTTP	483 HTTP/1.1 200 OK (text/html)

Figura 5.2: I messaggi *HTTP* prodotti dalla conversazione tra Alice e Bob. In nero i messaggi che contengono il carattere "\t" nell'*header* `Content-Lenght`.

Isoliamo i messaggi *HTTP* dal resto del traffico impostando il filtro di visualizzazione su "`http`". Possiamo inoltre colorare diversamente i messaggi che contengono un carattere di tabulazione ("\t") all'interno del valore dell'*header* `Content-Lenght` mediante il filtro: `http.content_length_header contains ''\t''`. Il risultato è mostrato in fig. 5.2, ed evidenzia i messaggi che trasportano informazione nascosta.

Esaminando il primo dei messaggi colorati, possiamo notare in fig. 5.3 una serie di caratteri di tabulazione inseriti in coda alle cifre che indicano la lunghezza del contenuto del messaggio. Vediamo che la sequenza di spazi bianchi inizia con `[SP] [HT] [SP] [SP] [HT] [SP] [SP] [SP]`, corrispondente al numero binario 01001000_2, ovvero il carattere "H" in *ASCII*. A seguire troviamo la codifica del resto del primo messaggio nascosto inviato da Alice a Bob.

Potrebbe sembrare che il canale sia facilmente individuabile a seguito di una semplice ispezione visiva. Tuttavia, è importante notare che una sequenza di caratteri `HT` non è tipicamente visibile, ed è risultata evidente nell'analisi appena condotta solo a causa della resa grafica impartitagli da *Wireshark*.

Se l'utente avesse ispezionato il traffico attraverso strumenti più comuni, come quelli messi a disposizione da un *browser web*, difficilmente avrebbe notato la

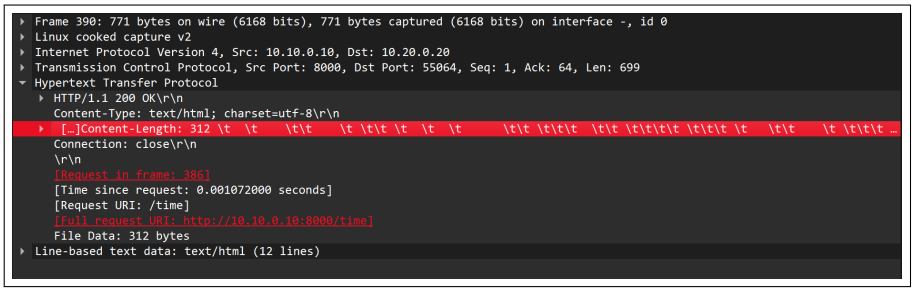
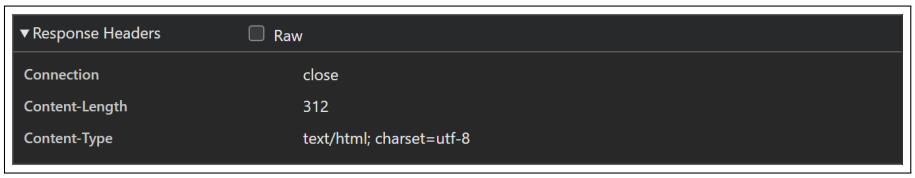


Figura 5.3: Il messaggio che trasporta il primo messaggio nascosto della conversazione.

trasmissione. Infatti, come visibile in fig. 5.4, tipicamente i valori degli *header* vengono visualizzati in modo compatto (fig. 5.4a), a seguito di un *parsing* del messaggio durante il quale gli spazi bianchi vengono rimossi. È soltanto forzando la visualizzazione di tutti i caratteri, e andando a selezionarli uno a uno (fig. 5.4b), che si riesce a notare la presenza di caratteri *non printable* come `HT` e `SP`. Addirittura, alcuni browser (nell'esempio *Firefox*[3]) potrebbero non mostrare affatto i caratteri `HT` e `SP` nonostante si sia richiesta la visualizzazione *raw* (non elaborata) degli *header* (fig. 5.4c). Questo è probabilmente dovuto a una normalizzazione del messaggio effettuata ancora prima di renderlo disponibile per l'analisi da parte dell'utente.

[3] `https://www.mozilla.org/firefox/`



(a) *Browser* basato su *Chromium*, visualizzazione compatta.

(b) *Browser* basato su *Chromium*, visualizzazione *raw*.

(c) *Firefox*, visualizzazione *raw*.

Figura 5.4: La visualizzazione attraverso gli analizzatori di diversi *browser web* del contenuto degli *header* di un messaggio che contiene informazione nascosta mediante la tecnica descritta.

Capitolo 6

Esempio C: *Padding*

Secondo il *Web Almanac*, nel 2020 *HTTP/1.1* veniva impiegato in una richiesta su 2; nel 2022 siamo scesi a 1 richiesta ogni 3, e al giorno d'oggi (dati risalenti al 2024) solo il 20% circa delle richieste fa ancora uso di *HTTP/1.1*[1]. La motivazione è semplice: nonostante *HTTP/1.1* funzioni ancora perfettamente, le versioni successive hanno introdotto numerosi miglioramenti in termini di sicurezza e prestazioni [MPB24].

Molte delle tecniche per la realizzazione di *CC* in *HTTP/1.X* sono state intenzionalmente rese inefficaci in *HTTP/2* [DM17]. Infatti, l'inflessibilità di diversi aspetti del protocollo rappresenta una scelta progettuale consapevole e mirata a ridurre la superficie di attacco. Risulta emblematico in questo senso il seguente passaggio tratto dalla *RFC* 7541:

> *"Il formato HPACK*[2] *è intenzionalmente semplice e inflessibile. Entrambe le caratteristiche riducono il rischio di problemi di sicurezza e interoperabilità dovuti a errori di implementazione. Non è previsto nessun meccanismo di estensibilità [PR15]."*

Ad esempio, la possibilità di utilizzare sequenze di spazi bianchi all'interno del valore di un *header* come nell'Esempio B, è stata soppressa dal nuovo standard, specificando (tra le altre cose) che:

[1] I dati si riferiscono al protocollo utilizzato per caricare la prima pagina di ogni sito web, i dati calcolati considerando anche le richieste successive sono ancora più a favore di *HTTP/2+*.

[2] Il formato di compressione degli *header* in *HTTP/2*.

> *"Il valore di un campo **non deve** iniziare o terminare con un carattere spazio bianco (`SP` o `HT`)[TB22]."*

Inoltre,

> *"Una richiesta contenente un campo che violi una qualsiasi di queste condizioni **deve** essere considerata malformata. In particolare, un intermediario **non deve** inoltrare alcun campo che contenga valori proibiti . . . e **dovrebbe** generare una risposta `400 (Bad Request)`[TB22]."*

Risulta chiaro, quindi, come i progettisti di *HTTP/2* abbiano voluto ridurre al minimo l'ambiguità e i comportamenti non standardizzati. Nonostante questo, permangono diverse possibilità per l'implementazione di *CC*.

In questo capitolo, discuteremo e implementeremo una tecnica presentata in [DM17], che sfrutta il meccanismo di *padding* (riempimento) di *HTTP/2*. Prima, però, è necessario fare una premessa riguardo all'utilizzo di *HTTPS*.

6.1 Cifrare o non cifrare?

Il *Transparency Report* di *Google* riporta che 97 dei 100 siti web più trafficati al mondo utilizzano *HTTPS* (*HTTP* su *TLS*) come impostazione standard, mentre solo 1 non lo supporta affatto [Goo24].

HTTP/2, infatti, è stato progettato con la crittografia in mente ma, dopo lunga discussione, il *Working Group* non ha raggiunto il consenso per renderne obbligatorio l'utilizzo[3]. Per questo, esistono due modalità di impiego per *HTTP/2*, identificate dalle stringhe "`h2`" e "`h2c`" [BPT15]:

- `h2` identifica il protocollo in cui *HTTP/2* fa uso di *Transport Layer Security* (*TLS*) per il trasporto sicuro, e dell'estensione *Application-Layer Protocol Negotiation* (*ALPN*) per negoziare i parametri della connessione.
- `h2c` identifica il protocollo che prevede l'utilizzo di *HTTP/2* trasportato in chiaro su *TCP*, e normalmente richiede l'*upgrade* di una connessione *HTTP/1.1* esistente (negoziato tramite l'*header* `Upgrade`).

[3] `https://http2.github.io/faq/#does-http2-require-encryption`

Ciononostante, lo standard *de facto* è rappresentato da `h2`, in quanto diverse implementazioni di *HTTP/2* hanno annunciato il supporto esclusivo per questa modalità, e, in particolare, nessuno dei principali *web browser* attualmente supporta `h2c`. Tuttavia, come già discusso in sezione 3.1, l'impiego della crittografia rappresenta un ostacolo per i nostri obiettivi; di conseguenza, impiegheremo `h2c` per la nostra dimostrazione.

6.2 Cenni su *HTTP/2*

Ad un alto livello di astrazione, *HTTP/2* consente il trasferimento simultaneo di più flussi bidirezionali indipendenti, ognuno dei quali trasporta gli stessi messaggi logici di *HTTP/1.1*, mediante una singola connessione *TCP*. I messaggi sono codificati in formato binario come *payload* di unità informative di base chiamate *frame*. La multiplazione (*multiplexing*) della connessione è ottenuta assegnando a ciascuno scambio di richieste e risposte un proprio *stream*. Ogni *stream* rappresenta quindi il raggruppamento logico dei *frame* che trasportano i messaggi di una data richiesta [TB22].

6.2.1 *Padding*

Esistono diversi tipi di *frame*, ognuno dei quali gioca un ruolo diverso nell'instaurazione e nella gestione della connessione nel suo complesso, o di *stream* specifici. Ogni tipo di *frame* è identificato da un codice univoco di 8 *bit* [TB22].

Tre tipi di frame, `DATA`, `HEADERS` e `PUSH PROMISE`, utilizzano il *padding* (riempimento) per offuscare la lunghezza del messaggio [DM17]. Questo meccanismo è stato introdotto come mitigazione contro alcuni specifici attacchi (ad esempio, *BREACH*) a cui *HTTP* è vulnerabile, in cui risposte compresse contenenti sia dati sotto il controllo dell'attaccante sia dati segreti possono essere sfruttate, tramite misurazioni ripetute della loro lunghezza, per inferire il contenuto riservato.

```
DATA Frame {
  Length (24),
  Type (8) = 0x00,

  Unused Flags (4),
  PADDED Flag (1),
  Unused Flags (2),
  END_STREAM Flag (1),

  Reserved (1),
  Stream Identifier (31),

  [Pad Length (8)],
  Data (..),
  Padding (..2040),
}
```

Listato 6.1: Il formato di un *frame* `DATA` [TB22].

La tecnica che impiegheremo sfrutta, in particolare, il *padding* all'interno di *frame* "`DATA`", in quanto le raccomandazioni dello standard lo rendono con buona probabilità privo di interferenze [TB22]:

> *"Gli intermediari **dovrebbero** mantenere il padding per i frame di tipo `DATA`, mentre **possono** rimuoverlo per i frame di tipo `HEADERS` e `PUSH PROMISE`."*

Questo tipo di *frame* (codice `0x00`) è utilizzato per trasmettere sequenze di *byte* di lunghezza variabile associate a un dato *stream*. Ad esempio, uno o più `DATA` *frame* possono essere utilizzati per trasmettere il corpo di una richiesta o risposta *HTTP*. Il formato di un *frame* di tipo `DATA` è mostrato nel listato 6.1.

Il campo indicato come `PADDED Flag` è un *bit* che, se impostato a 1, indica la presenza del campo opzionale `Pad Length`, che a sua volta specifica la lunghezza del campo `Padding` in *byte*. I *byte* di *padding* devono essere impostati a zero al momento dell'invio, mentre non è obbligatorio controllarne il valore in ricezione, anche se la presenza di *bit* diversi da zero è da considerare come un errore di connessione [TB22].

Lo standard prevede esplicitamente la possibilità di impostare il campo `Pad Length` a zero; di conseguenza, è possibile specificare l'assenza di *padding* attraverso due codifiche semanticamente equivalenti [TB22]:

- non impostare il *flag* `PADDED` e non includere il campo `Pad Length`, o
- impostare il *flag* `PADDED` e specificare un valore di `Pad Length` pari a zero.

Scegliendo tra i due modi alternativi di codificare l'assenza di *padding*, è possibile inviare un simbolo binario nascosto per ogni *frame* di tipo `DATA` inviato [DM17].

6.3 Classificazione del *Covert Channel*

Il *CC* descritto, come i precedenti, è un **canale di memorizzazione senza modifica del contenuto**. Si basa su una **modifica della struttura** del messaggio, in quanto il campo `Pad Length` viene aggiunto al messaggio per codificare un valore binario; siccome così facendo si va a "creare" uno spazio all'interno del messaggio che normalmente non è previsto, la tecnica rientra nel pattern **"aggiunta di ridondanza"** (vedi fig. 6.1).

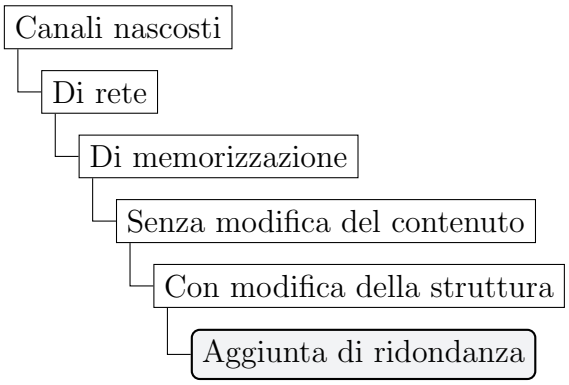


Figura 6.1: Classificazione del *Covert Channel* realizzato attraverso l'utilizzo del *padding* nel contesto della tassonomia individuata da Wendzel et al. (fig. 2.1)

6.4 Implementazione

L'implementazione del *CC* è basata sulla libreria *Python* `hyper-h2` [Bc], che fornisce una realizzazione in puro *Python* di *HTTP/2*. Il *server* e il *client* utilizzati in questo *PoC* sono ispirati agli esempi forniti dalla libreria stessa[4], adattati alla struttura del progetto e modificati per inserire dati nascosti secondo la tecnica descritta. Infatti, nonostante quest'ultima si basi su una manipolazione di basso livello, è comunque possibile applicarla attraverso l'interfaccia messa a disposizione dalle classi di `hyper-h2`, pur adottando un piccolo "trucco" in fase di ricezione.

Nel descrivere il funzionamento di questo *PoC* trascureremo, per non diluire troppo la trattazione, alcuni dettagli anche interessanti del funzionamento del *server* e del *client HTTP*. Per la visione del codice completo si rimanda al repository *GitHub*[5] del progetto.

6.4.1 La copertura

Il *throughput* di un *CC* si può analizzare in termini di *bit* nascosti trasmessi al secondo o di *bit* nascosti in ogni *PDU* (*Packet Raw Bit Rate*, *PRBR*). Nel caso ideale in cui nessun pacchetto venga perso, la capacità di trasmissione di un canale di memorizzazione può essere espresso come $C = PRBR \cdot N_{bps}$, dove N è il numero di *frame* di tipo `DATA` inviati al secondo [MP14].

Al contrario degli esempi precedenti, in cui il valore di *PRBR* era sostanzialmente arbitrario, questo *CC* ha un *throughput* limitato ad un singolo bit per ogni *frame* di tipo `DATA` inviato. Per garantire una capacità di trasmissione adatta ad una messaggistica in tempo reale, è quindi necessario che il tasso di invio dei frame (N) sia sufficientemente elevato.

Proponiamo quindi uno scenario differente rispetto a quelli precedenti, in cui le richieste lecite utilizzate come vettore per i messaggi nascosti si fingono generate in automatico dallo *user agent* di un fantomatico utente. In particolare, una richiesta `GET` alla risorsa `/dashboard` restituisce una semplice pagina *HTML* (visibile nel

[4] Il lettore interessato è invitato a consultare l'ottima spiegazione presente nella documentazione della libreria: `https://python-hyper.org/projects/hyper-h2/en/stable/basic-usage.html`.

[5] `https://github.com/fiocchidavide/Bachelors-Testbed`

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <title>Dashboard</title>
        <script>
            function fetchTemperature() {
                fetch('/status')
                    .then(response => response.json())
                    .then(data => {
                        const table = document.getElementById("tempTable");
                        const row = table.insertRow(1);
                        const timeCell = row.insertCell(0);
                        const tempCell = row.insertCell(1);
                        timeCell.textContent = new Date().toLocaleTimeString();
                        tempCell.textContent = data.temperature + " \u00b0C";
                    });
            }
            setInterval(fetchTemperature, 100);
        </script>
    </head>
    <body>
        <h1>Temperature monitoring</h1>
        <table id="tempTable" border="1">
            <thead>
                <tr><th>Time</th><th>Temperature</th></tr>
            </thead>
            <tbody></tbody>
        </table>
    </body>
</html>
```

Listato 6.2: La pagina web servita dal server *HTTP/2*.

listato 6.2), che mostra una tabella con un qualche tipo di dato proveniente dal server (nell'esempio, la temperatura). La tabella viene aggiornata in tempo reale da uno script *JavaScript* incluso nella pagina ed eseguito dal *browser* dell'utente. Lo script invia una richiesta `GET` alla risorsa `/status` ad intervalli di tempo regolari, a cui il server risponde con un messaggio `200 OK` contenente i dati aggiornati in formato *JavaScript Object Notation* (*JSON*), con i quali viene aggiornata la tabella.

6.4.2 Protocollo di codifica

Un'ulteriore differenza rispetto alle implementazioni realizzate finora è rappresentata dalla codifica dei messaggi. Infatti, attraverso questa tecnica i messaggi non sono più contenuti in una singola *PDU*, ma sono trasportati da una sequenza di

frame. Tralasciando di considerare aspetti che esulano dallo scopo di questo lavoro, come la resistenza al rumore[6], bisogna comunque affrontare il problema di definire i limiti dei caratteri (e, più in generale, dei messaggi) nascosti all'interno del flusso di dati.

A questo scopo, adottiamo il seguente protocollo:

- Ogni messaggio è preceduto da un preambolo di un *byte* uguale alla codifica binaria del numero 128 (`10000000`).

- I messaggi ammissibili sono composti da una sequenza di soli caratteri *ASCII* e terminano con il carattere `0x00` (`NULL`) o con l'inizio di un nuovo messaggio (segnalato dalla presenza di un preambolo).

- Ogni carattere è trasmesso come l'ottetto in cui il *bit* più significativo è uguale a zero, e i restanti 7 *bit* sono uguali alla codifica *ASCII* del carattere in formato *big-endian*[7].

- In assenza di alcun messaggio da inviare, sulla linea viene trasmesso costantemente il valore 0.

Secondo quanto appena descritto, per codificare il messaggio "`Ciao`", il server invierà il seguente flusso di *bit*:

```
10000000 01000011 01001001 01000001 01001111 00000000 00...
[ 128  ] [  C   ] [  i   ] [  a   ] [  o   ] [  0   ] [ ...
```

Per quanto riguarda la codifica dei *bit* all'interno dei singoli *frame*, la scelta di utilizzare l'uno o l'altro formato per rappresentare un determinato simbolo binario è puramente arbitraria. Nella nostra implementazione il *bit* trasmesso equivale al valore del *flag* `PADDED` del *frame*.

[6] Ovvero, non ci porremo il problema di correggere eventuali errori causati dalla perdita dei pacchetti (che nella nostra implementazione "ingenua" potrebbe ad esempio essere causata da richieste provenienti da altri client).

[7] Ovvero, la trasmissione procede dal *bit* più significativo a quello meno significativo.

6.4.3 *Server*

La classe `PaddingServer` si comporta in modo simile agli esempi precedenti; ovvero, il metodo `run` è costituito da un ciclo infinito in cui il server accetta connessioni da parte dei client e le gestisce in modo seriale.

Generare l'informazione nascosta

La logica utilizzata per determinare il successivo *bit* da inviare è incapsulata in un oggetto della classe `PaddingServer.Transmitter`, che si occupa di mantenere una rappresentazione interna in formato binario del messaggio che si sta trasmettendo. Nel listato 6.3 vediamo come alla prima chiamata del metodo `get_next_bit`, il `Transmitter` controlli la disponibilità di un messaggio da inviare, e in caso positivo lo codifichi secondo il protocollo descritto in sezione 6.4.2, memorizzando i *bit* da trasmettere in un *buffer*[8]. Ad ogni chiamata successiva, il metodo restituirà il prossimo *bit* da inviare, fino all'esaurimento del *buffer*; se questo risulta vuoto, e non è disponibile nessun messaggio per ripopolarlo, ogni chiamata a `get_next_bit` restituisce il valore 0.

Inserire l'informazione nascosta

Il metodo `PaddingServer.handle_request` si occupa di estrarre le richieste *HTTP* dai segmenti *TCP* ricevuti, delegando al metodo `PaddingServer.send_response` il compito di produrre una risposta adeguata.

Nel listato 6.4 è possibile vedere come il *server* codifichi un *bit* di informazione nascosta all'interno di ogni risposta a richieste `GET` per la risorsa `/status`. In particolare, includendo il parameto "`pad_length = 0`" nella chiamata al metodo `H2Connection.send_data`, il *server* invierà un *frame* di tipo `DATA` con il *flag* `PADDED` impostato a 1, e `Pad Length` pari a zero, codificando così un *bit* nascosto di valore 1.

[8]In realtà, l'implementazione proposta adotta un meccanismo di generazione *lazy* (puntuale) dei *bit* da mandare, senza mantenere in memoria la codifica dell'intero messaggio.

```
class Transmitter:
  def __init__(self, get_message_to_send):
    # Expects a function which behaves similarly to queue.Queue.get_nowait
    self.get_message_to_send = get_message_to_send
    self.current_iterator = None

  def fetch_message(self):
    # Raises queue.Empty if no message is available
    message = self.get_message_to_send()

    # A preamble is added
    bytes_to_send = (128).to_bytes(1, 'big') + message.encode(encoding='ascii')

    # Builds and returns an iterable of all the bits which compose the message
    bits_to_send = chain.from_iterable(map(self.get_bits, bytes_to_send))
    return bits_to_send

  def get_next_bit(self):
    while True:
      if self.current_iterator is None:
        try:
          self.current_iterator = self.fetch_message()
        except queue.Empty:
          return 0  # Default to zero when no messages are available
      try:
        return next(self.current_iterator)
      except StopIteration:
        self.current_iterator = None  # Message fully sent, fetch new one

  @staticmethod
  def get_bits(byte):
    return ((byte >> i) & 1 for i in range(7, -1, -1))
```

Listato 6.3: La classe `PaddingServer.Transmitter`

```
def send_response(self, http2_conn, event):
  if isinstance(event, RequestReceived):
    stream_id = event.stream_id
    headers = dict(event.headers)
    path = headers.get(b":path", b"")
    method = headers.get(b":method", b"")

    if path not in (b"/dashboard", b"/status"):
      # 404 Not Found
      ...
    elif method != b'GET':
      # 405 Method Not Allowed
      ...
    else:
      if path == b"/dashboard":
        # Send dashboard
        ...
      elif path == b"/status":
        response_headers = [
          (':status', '200'),
          ('content-type', 'application/json'),
          ('content-length', str(len(payload)))
        ]
        http2_conn.send_headers(stream_id, response_headers)

        # Generate a random temperature
        temperature = round(random.normalvariate(25, 2), 1)
        payload = json.dumps({"temperature": temperature}).encode('utf-8')

        # Embed a single bit using padding (Pad Lenght present if bit == 1)
        if self.transmitter.get_next_bit():
          http2_conn.send_data(stream_id, payload, pad_length=0, end_stream=
              True)
        else:
          http2_conn.send_data(stream_id, payload, end_stream=True)
```

Listato 6.4: Un estratto del metodo `PaddingServer.send_response`

6.4.4 *Client*

La classe `PaddingClient` si occupa di inviare una prima richiesta `GET` alla risorsa `/dashboard`, seguita da molteplici richieste `GET` per la risorsa `/status` a intervalli di tempo regolari, simulando il comportamento di un *browser* che esegua lo *script* incluso nella pagina.

La gestione dello *stream* che trasporta la risposta del *server* è affidata al metodo `PaddingClient.consume_stream`. Questo metodo si occupa di ricevere tutti i *frame* inviati dal *server*, fino alla chiusura dello *stream*, e di consegnare i *frame* di tipo `DATA` ad un *consumer* (consumatore) fornito come parametro. Nel nostro caso, si tratta di un oggetto della classe `PaddingClient.Receiver`, il cui compito è quello di estrarre i *bit* di informazione nascosta dai *frame* ricevuti, e di ricomporre il messaggio nascosto che codificano.

Estrarre l'informazione nascosta

Come anticipato, in fase di ricezione è necessario adottare un piccolo accorgimento per poter estrarre i *bit* nascosti. Infatti, la libreria `hyper_h2` non espone direttamente il flag `padded` o la presenza di *padding* nei *frame* ricevuti. Questo significa che non è possibile sapere esplicitamente, ad esempio tramite un attributo dell'evento, se il *frame* originale conteneva *padding* oppure no.

Per ovviare a questa limitazione, il metodo `Receiver.extract_bit` (riportato nel listato 6.5) confronta la lunghezza totale del *frame*, data dall'attributo `event.flow_controlled_length`, con la lunghezza effettiva dei dati ricevuti, ottenuta con `len(event.data)`. Se la prima è maggiore della seconda, significa che era presente il campo `Pad Length`, infatti, come specificato in [TB22]:

> *"L'intero payload del frame* ***`DATA`*** *è incluso nel computo del controllo di flusso, compresi i campi* ***`Pad Length`*** *e* ***`Padding`****, se presenti."*

Attraverso questo confronto, è facile dedurre il valore del *bit* nascosto.

```
def extract_bit(self, event):
  if event.flow_controlled_length > len(event.data):
    # Padding Length field present => hidden bit = 1
    return 1
  else:
    # No padding => hidden bit = 0
    return 0
```

Listato 6.5: Un estratto del metodo `PaddingClient.Receiver.extract_bit`

Ricomporre il messaggio nascosto

Una volta estratto un *bit* di informazione nascosta, è necessario comprendere quale sia il suo significato nel contesto della comunicazione in corso. La classe `PaddingClient.Receiver` implementa un semplice automa a stati finiti per decodificare i messaggi ricevuti. Il relativo diagramma è riportato in fig. 6.2, e comprende i seguenti stati:

- **In attesa**: stato iniziale e intermedio tra due messaggi. Il `Receiver` rimane in ascolto finché non riceve un *bit* pari a 1, che segnala il potenziale inizio di un messaggio. In tal caso, passa allo stato "Ricevuto un uno" e inizia a costruire il primo *byte*.

- **Ricevuto un uno**: in questo stato, il `Receiver` attende altri 7 *bit* per completare un *byte*. Se il *byte* risultante è $128_{10} = 10000000_2$, viene riconosciuto come preambolo valido e l'automa passa allo stato "In ricezione". In caso contrario, si entra nello stato "Errore".

- **In ricezione**: il `Receiver` continua a raccogliere gruppi di 8 *bit*, convertendoli in caratteri *ASCII* da aggiungere al messaggio corrente. Se il *byte* ricevuto è 0, il messaggio viene considerato terminato e inviato al consumatore (il *Controller*), poi il `Receiver` torna nello stato di attesa. Se invece il *byte* ricevuto vale 128, viene riconosciuto come preambolo valido, provocando la consegna del messaggio corrente al *Controller* e l'inizio della ricezione di un nuovo messaggio. In caso di *byte* non validi (non *ASCII*), il `Receiver` rileva la corruzione del messaggio corrente e passa allo stato di errore, in cui attende il termine della trasmissione.

- **Errore**: il `Receiver` attende una sequenza di 14 zeri prima di tornare allo stato di attesa. Questo, nell'ipotesi che solo un *bit* sia errato, garantisce che almeno gli ultimi 8 zeri ricevuti non facciano parte di un messaggio[9], ma segnalino effettivamente l'assenza di dati nascosti.

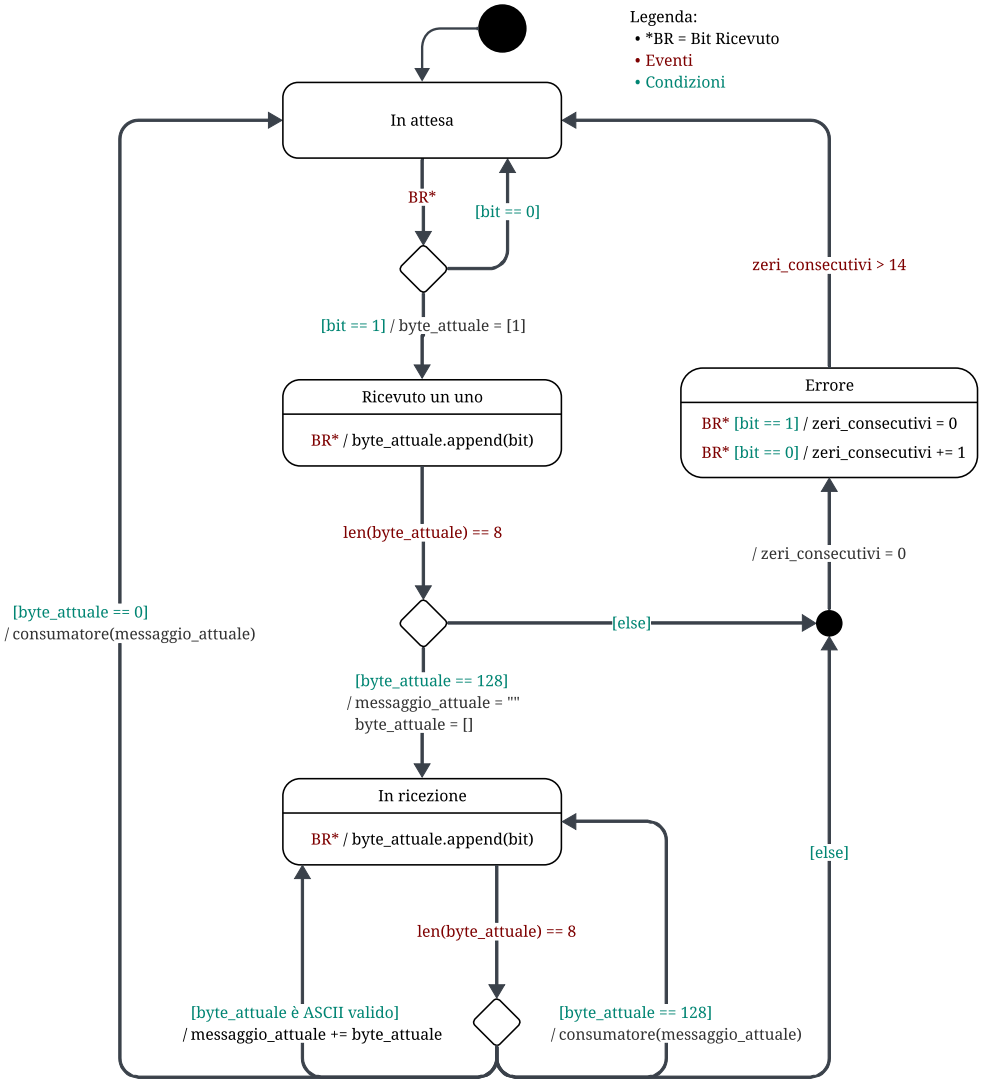


Figura 6.2: Il diagramma degli stati della classe `PaddingClient.Receiver`.

[9]La sequenza di zeri più lunga possibile all'interno di un messaggio con un singolo errore rilevabile è di 13, causata dalla ricezione di un *byte* (errato) `11000000` seguito da un byte (tecnicamente valido) con valore 1.

6.5 Analisi del traffico

Analizziamo il traffico generato da questo *PoC*, quando è utilizzato per trasferire i messaggi della conversazione descritta in sezione 4.4.

Dato l'elevato numero di richieste `GET` generate dai client di Alice e Bob, applicare il solo filtro di visualizzazione "`http2`" non permetterebbe di analizzare chiaramente lo scambio. Scegliamo quindi di concentrarti su un singolo verso della comunicazione: attraverso il filtro

```
(ip.src == 10.20.0.20 and http2.headers.method)
                      or
(ip.src == 10.10.0.10 and http2.headers.status)
```

selezioniamo soltanto le richieste inviate dal client di Bob (indirizzo `10.20.0.20`) e le risposte del server di Alice (indirizzo `10.10.0.10`). In questo modo, andiamo ad isolare il canale nascosto che va da Alice a Bob, permettendo un'analisi più agevole del flusso di *bit* che trasporta.

Per distinguere i *frame* che fanno uso di *padding* è possibile applicare il filtro `http2.flags.padded == True`. Colorando diversamente i messaggi che soddisfano questa condizione, si ottiene la visualizzazione riportata in fig. 6.3. In questo modo, è immediato riconoscere i messaggi che veicolano *bit* nascosti con valore 1. Nella figura, in particolare, è riportata la prima occorrenza di un tale messaggio, e possiamo vedere che la sequenza di *bit* trasmessi corrisponde proprio all'inizio del primo messaggio inviato da Alice a Bob. Infatti, in seguito alla trasmissione del preambolo (10000000_2), si può riconoscere la sequenza che codifica il carattere "H" (01001000_2), ovvero la prima lettera del messaggio "Hai notato quando cambia la guardia?".

Analizzando il messaggio che trasporta il primo *bit* nascosto di valore 1, vediamo (in fig. 6.4) che contiene un *frame* di tipo `DATA` con il *flag* `PADDED` impostato a 1, e il campo `Pad Length` pari a zero.

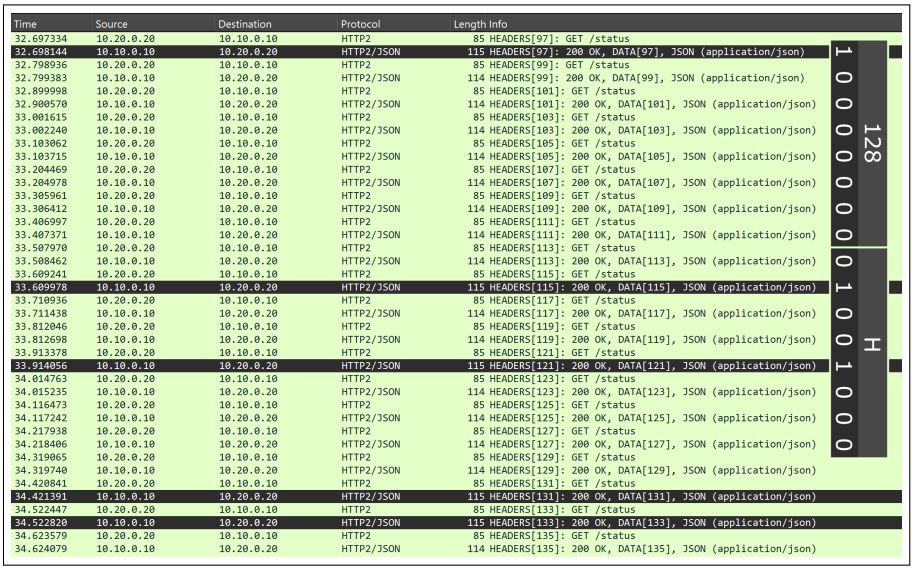


Figura 6.3: Le risposte inviate dal *server* di Alice al *client* di Bob. In nero i messaggi con il *flag* `PADDED` impostato; a destra è mostrata la corrispondente informazione nascosta.

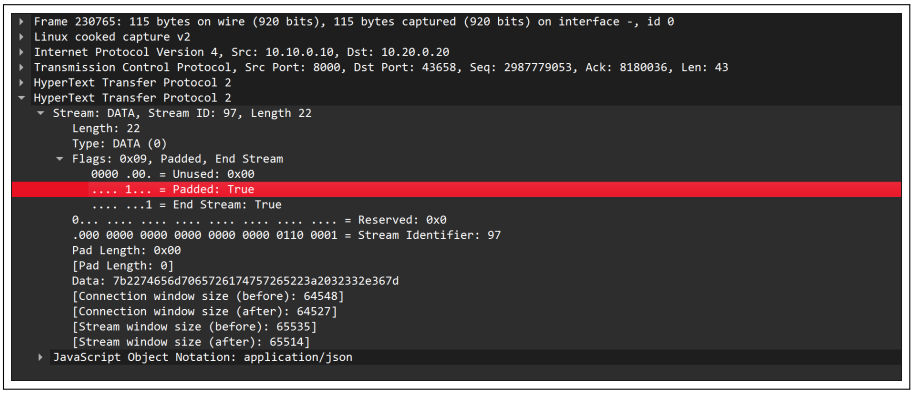


Figura 6.4: Il messaggio che trasporta il primo *bit* di informazione nascosta.

"È solo quando ci troviamo davanti un problema che possiamo iniziare a cercarne la soluzione."

Karl Popper, *Objective Knowledge: An evolutionary approach*

Capitolo 7

Conclusione

Nel lavoro che si va a concludere abbiamo esplorato il tema dei canali nascosti di rete, concentrandoci in particolare sui canali nascosti di memorizzazione. L'integrazione dei fondamenti teorici con la formulazione di un semplice metodo di analisi e l'implementazione di alcuni esempi pratici hanno contribuito a raggiungere una buona comprensione di un fenomeno che riveste crescente importanza nel panorama della sicurezza informatica. Abbiamo iniziato introducendo il concetto di canale nascosto, analizzando in che modo e per quali ragioni questo possa essere utilizzato al fine di trasmettere informazioni sfuggendo alla rilevazione. Sono stati quindi esaminati alcuni dei contributi più significativi in tema di classificazione delle tecniche conosciute. Questo ha posto le fondamenta teoriche per il successivo lavoro sperimentale, la cui prima fase è consistita nell'identificazione dei requisiti di un ambiente controllato per lo studio dei canali nascosti di rete. Dopo aver progettato e realizzato una possibile soluzione, sono stati condotti alcuni semplici test di comunicazione nascosta al suo interno.

7.1 Una progressione crescente

Si è visto come l'atto di nascondere informazione nei campi di un protocollo di rete da parte del mittente, e la loro estrazione da parte del ricevitore rappresentino i due passi necessari per la realizzazione di un canale nascosto di memorizzazione [Els24]. Abbiamo descritto e implementato una progressione di tre tecniche appartenenti

a questa categoria, basate su diverse versioni del protocollo *HTTP*. Per ciascun esempio, è stata condotta ed analizzata una comunicazione di prova all'interno del *testbed* realizzato appositamente. Vediamo ora cosa si può trarre dal confronto tra queste tecniche.

Diversi protocolli, tra cui *HTTP*, supportano l'aggiunta di estensioni all'intestazione minima, permettendo di trasferire informazioni opzionali al bisogno [ZAB07]. Nel capitolo 4 abbiamo visto come sia semplice sfruttare questo fatto per inserire informazione nascosta all'interno di un *header* raramente utilizzato in *HTTP/1.1*. Si tratta di un approccio diretto e facilmente rilevabile, a dimostrazione di come la semplicità di implementazione spesso si traduca in scarsa furtività.

Nel capitolo 5 abbiamo quindi proposto un metodo alternativo in grado di sfuggire ad un'ispezione visiva non eccessivamente approfondita, che sfrutta la particolare formulazione delle specifiche di *HTTP* per realizzare una funzionalità imprevista (la comunicazione nascosta) attraverso un comportamento esplicitamente consentito. Nonostante possa non risultare evidente (in quanto il *PRBR*[1] è rimasto sostanzialmente arbitrario), è avvenuta in questo passaggio una riduzione dell'efficienza di trasmissione del canale. Infatti, siamo passati dal trasmettere direttamente i caratteri che compongono il messaggio, al trasmettere un singolo *bit* per ogni carattere "*whitespace*" aggiuntivo. Ciò non è altro che una conseguenza particolare del principo generale per cui furtività, capacità e robustezza rappresentano un trilemma nel campo dei canali nascosti, e massimizzare uno di questi aspetti porta a penalizzare gli altri [Cav21].

Questo è reso ancora più evidente nel capitolo 6, in cui viene implementata una tecnica che gode di una furtività molto maggiore ma è severamente limitata in quanto a capacità di trasmissione, potendo al più codificare un singolo *bit* di informazione nascosta per ogni *frame* inviato. La necessità di un tale compromesso è una conseguenza del fatto che molte delle problematiche che consentivano di realizzare una comunicazione nascosta nelle prime versioni di *HTTP* siano state prese in considerazione ed eliminte negli sviluppi più recenti dello standard. Ma per ogni vulnerabilità mitigata, ne viene introdotta una nuova, e facendo uso di funzionalità inedite di *HTTP/2* abbiamo mostrato come sia comunque possibile nascondere in-

[1] Vedi sezione 6.4.1.

formazione sfruttando la natura duplice di una funzionalità introdotta proprio per risolvere un problema di sicurezza.

7.2 Una continua evoluzione

Nonostante sia comunque possibile impiegare diverse tecniche per la realizzazione di *CC* in *HTTP/2*[DM17], l'intento dei progettisti del protocollo è sicuramente quello vincente. Infatti, la complessità dei *Network Covert Channel* è in costante aumento: nuovi schemi di costruzione sono stati proposti, alcuni dei quali esulano dalle classificazioni e dagli impianti teorici esistenti [WSZK25]. Inoltre, le caratteristiche degli ambienti di rete emergenti, come lo *streaming* multimediale, la *blockchain* e *IPv6*, offrono condizioni convenienti per la proliferazione di canali ancora più difficili da individuare rispetto a quelli tradizionali [TXLG20]. Ci si aspetta che le future comunicazioni nascoste saranno più sofisticate, multi-livello e in grado di sfruttare la complessa interazione tra componenti *hardware* e *software* [Cav21].

Mitigare la minaccia dei *Covert Channel*s in un panorama così incerto comporta intrinsecamente dei compromessi. L'obiettivo ideale di eliminare completamente un canale non è sempre raggiungibile senza interrompere servizi legittimi o degradare eccessivamente la qualità percepita dagli utenti. Invece, le contromisure spesso mirano a limitare la capacità di trasmissione del canale nascosto, rendendolo meno utile per l'attaccante, piuttosto che a eliminarlo del tutto [Cav21]. Risulta chiara la necessità di considerare la minaccia rappresentata dai *CC* già nelle fasi iniziali della progettazione di nuovi protocolli, servizi e architetture di rete: il principio della "*security-by-design*" mira a prevenire lo sfruttamento di ambiguità o difetti di progettazione che possono essere utilizzati per instaurare una comunicazione nascosta [Cav21, CM24].

L'identificazione di aspetti in comune alla base della creazione dei *CC* è di certo l'approccio più promettente per guidare la progettazione in questo senso [WZFH14, ZAB07]; nondimeno, il primo passo nella risoluzione di un problema è sempre quello di venire a conoscenza del problema stesso. Il presente lavoro, pur nella sua semplicità, ha voluto contribuire alla comprensione di questo tema fornendo esempi concreti di come principi teorici si manifestino nella pratica ed

evidenziando la necessità di un approccio olistico che consideri tanto gli aspetti tecnici quanto quelli di design nella corsa alle armi contro i canali nascosti di rete.

Bibliografia

[Bar11] Adam Barth. HTTP state management mechanism. *RFC*, 6265:1–37, 2011.

[Bc] Cory Benfield and contributors. hyper-h2: Http/2 state-machine based protocol stack. `https://github.com/python-hyper/h2`. Accessed: 2025-05-15.

[Bis22] Mike Bishop. HTTP/3. *RFC*, 9114:1–57, 2022.

[BPT15] Mike Belshe, Roberto Peon, and Martin Thomson. Hypertext transfer protocol version 2 (HTTP/2). *RFC*, 7540:1–96, 2015.

[Cav21] Luca Caviglione. Trends and challenges in network covert channels countermeasures. *Applied Sciences*, 11(4), 2021.

[CM24] Luca Caviglione and Wojciech Mazurczyk. You can't do that on protocols anymore: Analysis of covert channels in IETF standards. *IEEE Netw.*, 38(5):255–263, 2024.

[Cra98] Scott Craver. On public-key steganography in the presence of an active warden. In David Aucsmith, editor, *Information Hiding, Second International Workshop, Portland, Oregon, USA, April 14-17, 1998, Proceedings*, volume 1525 of *Lecture Notes in Computer Science*, pages 355–368. Springer, 1998.

[CZJ+24] Zhuo Chen, Liehuang Zhu, Peng Jiang, Can Zhang, Feng Gao, and Fuchun Guo. Exploring unobservable blockchain-based covert channel

for censorship-resistant systems. *IEEE Trans. Inf. Forensics Secur.*, 19:3380–3394, 2024.

[CZN+20] Krzysztof Cabaj, Piotr Zórawski, Piotr Nowakowski, Maciej Purski, and Wojciech Mazurczyk. Efficient distributed network covert channels for internet of things environments†. *J. Cybersecur.*, 6(1), 2020.

[DM17] Biljana Dimitrova and Aleksandra Mileva. Steganography of hypertext transfer protocol version 2 (http/2). *Journal of Computer and Communications*, 05(05):98–111, 2017.

[Els24] Muawia Elsadig. *Network Covert Channels*. IntechOpen, April 2024.

[FGM+99] Roy T. Fielding, Jim Gettys, Jeffrey C. Mogul, Henrik Frystyk Nielsen, Larry Masinter, Paul J. Leach, and Tim Berners-Lee. Hypertext transfer protocol - HTTP/1.1. *RFC*, 2616:1–176, 1999.

[FNR22a] Roy T. Fielding, Mark Nottingham, and Julian F. Reschke. HTTP semantics. *RFC*, 9110:1–194, 2022.

[FNR22b] Roy T. Fielding, Mark Nottingham, and Julian F. Reschke. HTTP/1.1. *RFC*, 9112:1–46, 2022.

[FR14] Roy T. Fielding and Julian F. Reschke. Hypertext transfer protocol (HTTP/1.1): semantics and content. *RFC*, 7231:1–101, 2014.

[fS18] International Organization for Standardization. ISO/IEC 27000:2018 — Information technology — Security techniques — Information security management systems — Overview and vocabulary, 2018. Section 3.28.

[GKS18] Waldemar Graniszewski, Jacek Krupski, and Krzysztof Szczypiorski. Somsteg - framework for covert channel, and its detection, within HTTP. *J. Univers. Comput. Sci.*, 24(7):864–891, 2018.

[Goo24] Google. Https encryption on the web, 2024.

[HI96] Theodore G. Handel and Maxwell T. Sandford II. Hiding data in the OSI network model. In Ross J. Anderson, editor, *Information Hiding, First International Workshop, Cambridge, UK, May 30 - June 1, 1996, Proceedings*, volume 1174 of *Lecture Notes in Computer Science*, pages 23–38. Springer, 1996.

[KBR+20] Mike Kosek, Leo Blöcher, Jan Rüth, Torsten Zimmermann, and Oliver Hohlfeld. Must, should, don't CARE: TCP conformance in the wild. In Anna Sperotto, Alberto Dainotti, and Burkhard Stiller, editors, *Passive and Active Measurement - 21st International Conference, PAM 2020, Eugene, Oregon, USA, March 30-31, 2020, Proceedings*, volume 12048 of *Lecture Notes in Computer Science*, pages 122–138. Springer, 2020.

[Kwe06] Zbigniew Kwecka. Application layer covert channel analysis and detection, 01 2006.

[Lam73] Butler W. Lampson. A note on the confinement problem. *Commun. ACM*, 16(10):613–615, 1973.

[MDS23] Ivan Miketic, Krithika Dhananjay, and Emre Salman. Covert channel communication as an emerging security threat in 2.5d/3d integrated systems. *Sensors*, 23(4):2081, 2023.

[Mil99] Jonathan K. Millen. 20 years of covert channel modeling and analysis. In *1999 IEEE Symposium on Security and Privacy, Oakland, California, USA, May 9-12, 1999*, pages 113–114. IEEE Computer Society, 1999.

[MP14] Aleksandra Mileva and Boris Panajotov. Covert channels in TCP/IP protocol stack - extended version-. *Central Eur. J. Comput. Sci.*, 4(2):45–66, 2014.

[MPB24] Robin Marx, Barry Pollard, and Chris Böttger. *HTTP*, chapter 18. HTTP Archive, 2024.

[oD85] Department of Defense. *Department of Defense Trusted Computer System Evaluation Criteria*, page 1–129. Palgrave Macmillan UK, 1985.

[PAK99] Fabien A. P. Petitcolas, Ross J. Anderson, and Markus G. Kuhn. Information hiding-a survey. *Proc. IEEE*, 87(7):1062–1078, 1999.

[Pos80] Jon Postel. Dod standard transmission control protocol. *RFC*, 761:1–88, 1980.

[PR15] Roberto Peon and Hervé Ruellan. HPACK: header compression for HTTP/2. *RFC*, 7541:1–55, 2015.

[ROL12] Ruben Rios, Jose A. Onieva, and Javier Lopez. Hide_dhcp: Covert communications through network configuration messages. In Dimitris Gritzalis, Steven Furnell, and Marianthi Theoharidou, editors, *Information Security and Privacy Research*, pages 162–173, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[Row97] Craig H. Rowland. Covert channels in the TCP/IP protocol suite. *First Monday*, 2(5), 1997.

[RSA78] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, 1978.

[Shi07] Robert W. Shirey. Internet security glossary, version 2. *RFC*, 4949:1–365, 2007.

[Sim83] Gustavus J. Simmons. The prisoners' problem and the subliminal channel. In David Chaum, editor, *Advances in Cryptology, Proceedings of CRYPTO '83, Santa Barbara, California, USA, August 21-24, 1983*, pages 51–67. Plenum Press, New York, 1983.

[SM06] Gaurav Shah and Andrés Molina. Keyboards and covert channels. In Angelos D. Keromytis, editor, *Proceedings of the 15th USENIX Security Symposium, Vancouver, BC, Canada, July 31 - August 4, 2006*. USENIX Association, 2006.

[SM07] K A Scarfone and P M Mell. *Guide to Intrusion Detection and Prevention Systems (IDPS)*. 2007.

[TB22] Martin Thomson and Cory Benfield. HTTP/2. *RFC*, 9113:1–78, 2022.

[TXLG20] Jing Tian, Gang Xiong, Zhen Li, and Gaopeng Gou. A survey of key technologies for constructing network covert channel. *Secur. Commun. Networks*, 2020:8892896:1–8892896:20, 2020.

[WSZK25] Steffen Wendzel, Tobias Schmidbauer, Sebastian Zillien, and Jörg Keller. DYST (did you see that?): An amplified covert channel that points to previously seen data. *IEEE Trans. Dependable Secur. Comput.*, 22(1):614–631, 2025.

[WZFH14] Steffen Wendzel, Sebastian Zander, Bernhard Fechner, and Christian Herdin. A pattern-based survey and categorization of network covert channel techniques. *CoRR*, abs/1406.2901, 2014.

[ZAB07] Sebastian Zander, Grenville J. Armitage, and Philip Branch. A survey of covert channels and countermeasures in computer network protocols. *IEEE Commun. Surv. Tutorials*, 9(1-4):44–57, 2007.

[ZCC00] Elizabeth D. Zwicky, Simon Cooper, and D. Brent Chapman. *Building internet firewalls - internet and web security (2. ed.)*. O'Reilly, 2000.

[ZY09] Cai Zhiyong and Zhang Yong. Entropy based taxonomy of network convert channels. In *2009 2nd International Conference on Power Electronics and Intelligent Transportation System (PEITS)*, page 451–455. IEEE, December 2009.

Ringraziamenti

Vorrei esprimere la mia sincera gratitudine a tutti coloro che hanno contribuito alla realizzazione di questo lavoro. In particolare, ringrazio il professor Franco Callegati per il suo supporto durante tutto il processo di stesura della tesi. La sua esperienza e i suoi consigli sono stati una guida indispensabile per completare il mio primo lavoro di ricerca e scrittura. Un ringraziamento speciale va anche a chi ha speso il proprio tempo a leggere e commentare le bozze di questa tesi, contribuendo a migliorarne la chiarezza e la qualità.

Ringrazio i miei amici, i miei familiari, e soprattutto la mia partner per il supporto costante in questo periodo impegnativo. Il loro incoraggiamento è stato fondamentale a mantenere alta la mia motivazione. Ringrazio chi ha saputo tenermi un posto nonostante la distanza, e chi ha reso facile trovarne uno nuovo. Le esperienze di amicizia sincera che ho vissuto mi danno il coraggio di osare, sapendo che ci sarà sempre qualcuno a guardarmi le spalle.

Infine, ringrazio i miei genitori, per avermi sempre sostenuto e incoraggiato, fornendomi ogni opportunità per crescere e imparare. Senza il loro supporto, non sarei qui oggi.