Università degli Studi di Torino

Facoltà di Scienze Matematiche Fisiche e Naturali

Corso di Laurea in Informatica

Anno Accademico 2004/2005

Relazione di Tirocinio

Studio e implementazione di un algoritmo di ottimizzazione di indirizzi di rete

Supervisore: Prof. Franco Sirovich

Candidato: Daniele Depetrini

Sommario

Il metodo originario per l'indirizzamento di reti Internet.....5
Il metodo attuale.....6
Esigenze funzionali 7
Esigenze implementative.....7
Esigenze prestazionali.....7
Definizioni.....8
Proprietà di conflitto.....9
Proprietà di riduzione.....9
Formato degli elementi di rete.....10
Implementazione della proprietà di conflitto.....11
Implementazione delle proprietà di riduzione.....11
Genesi dell'algoritmo.....12
Criterio di confronto fra reti (Criterio di ordinamento).....12
Struttura base dell'algoritmo.....12
Parsing dell'input.....13
Strutture dati.....13
Proprietà base.....14
Sort degli array.....14
Pseudo codice 15
Precondizioni.....18
Analisi della funzione: idee base e invariante ciclo principale.....18
Analisi della funzione: correttezza.....20
Espansione di una rete.....21
Postcondizioni.....23
Esempi.....23
Pseudo codice.....25
Precondizioni.....27
Analisi della funzione: idee base e invariante ciclo principale.....27
Analisi della funzione: correttezza.....28
Postcondizioni.....30
Esempi.....31
Get_entries.....31
Sort_entries.....32
Apply_whitelist.....32
Optimize.....32
Print_addresses.....33
Visione Complessiva.....33
Scelte implementative.....34
Compilazione.....34

Interfaccia del programma....34
Formati in input....35
Utilizzo....35
Libreria....35
Open Source....35
Profiling....36
Utilizzo di memoria....37
Testing....37
Esecuzione di apply_whitelist....39
Esecuzione di expand....43
Esecuzione di optimize....47
cidrrmerge.h....52
cidrmerge-lib.c....55
cidrmerge.c....71
Makefile....86

1.Premessa: terminologia

- **ELEMENTO DI RETE**: un indirizzo di rete espresso in notazione prefissa o notazione con netmask (le due notazioni sono nella pratica equivalenti).
- **NOTAZIONE CON PREFISSO**: formalismo per esprimere un elemento di rete. Consiste in un'espressione del tipo “dotted/N”, dove dotted è un indirizzo IP in formato dotted quad e N è un numero intero.
- **NOTAZIONE CON NETMASK**: formalismo per esprimere un elemento di rete. Consiste in un'espressione del tipo “network/netmask”, dove sia network che netmask sono indirizzi IP espressi in formato dotted quad.
- **HOST**: un singolo indirizzo IP. La notazione con prefisso dell'elemento di rete corrispondente sarà espressa come “host/32”, dove host è l'indirizzo IP dell'host specificato in notazione dotted quad.
- **INSIEME DI HOSTS**: è un insieme di indirizzi IP (hosts). Può essere rappresentato da un numero diverso ma equivalente di elementi di rete.
- **RETE**: insieme di hosts espressi con un solo elemento di rete. La notazione prefissa dell'elemento di rete corrispondente sarà “dotted/N”, dove “N” è la lunghezza in bit del prefisso comune a tutti gli hosts dell'insieme, specificata in “dotted” . Nel caso l'insieme contenga solo un elemento si ricade nel caso “HOST”.
- **AGGREGAZIONE DI RETI**: operazione che consiste nel sostituire la rete R1 e la rete R2 una nuova rete R3 che contiene tutti e soli gli hosts contenuti in R1 e R2.
- **SOTTORETE**: una rete R1 si dice sottorete di una seconda rete R2 quando tutti gli hosts di R1 sono anche contenuti in R2.
- **SOTTORETE IMPROPRIA**: quando vale la condizione che R1 è sottorete di R2, ma anche il viceversa. In questo caso le due reti sono equivalenti e rappresentano lo stesso insieme di hosts.
- **SOTTORETE PROPRIA**: quando una sottorete non è impropria.
- **RETI IN CONFLITTO**: due reti sono in conflitto quando hanno uno o più hosts in comune. Una è allora necessariamente sottorete dell'altra.
- **RIDUZIONE**: operazione mediante la quale viene ridotto il numero di elementi di rete lasciando invariato l'insieme iniziale di hosts, espressi sotto forma di reti.
- **ESPANSIONE DI UNA RETE**: operazione che viene effettuata quando la rete della blacklist r1 è in conflitto con la rete della whitelist r2 e r2 è sottorete propria di r1. Consiste nel rimpiazzare r1 con un insieme di reti che rappresentano tutti e soli gli hosts di r1 non in comune con alcun elemento di white.

- **PUNTATORE LOGICO**: indirizzo dell'array che ha come prefisso un numero maggiore o uguale a EXPANDED_PREFIX. Individuato come elemento r, “punta” logicamente a r.prefix- EXPANDED_PREFIX elementi dell'array expanded a partire dalla posizione r.network.
- **ELEMENTO NON VALIDO**: elemento di un array di tipo “entry” con il campo prefisso avente valore INVALID_PREFIX. Tale elemento non rappresenta una rete valida e deve essere sempre ignorato. E' presente nell'array solo per evitare una compattazione, operazione costosa a livello computazionale.
- **INVALIDAZIONE**: operazione mediante la quale ad un certo elemento di un array di tipo “entry” viene assegnato un prefisso uguale a INVALID_PREFIX, rendendolo elemento non valido.
- **BLACKLIST**: elenco di reti in input rappresentate sotto forma di array di tipo “entry” che costituiscono l'insieme “I”. Sono le reti riconosciute come fonte di spam.
- **WHITELIST**: elenco di reti in input rappresentate sotto forma di array di tipo “entry” che costituiscono l'insieme “W”. Sono le reti che non devono mai essere presenti nella blacklist locale.
- **SPAM**: email non desiderate, che tipicamente pubblicizzano prodotti o servizi senza che vi sia stata una richiesta preliminare da parte del destinatario.
- **INVALID_PREFIX**: prefisso speciale utilizzato per marcare un elemento non valido. Vale 33.
- **EXPANDED_PREFIX**: prefisso non valido utilizzato per marcare che l'elemento dell'array è un puntatore logico. Vale 220.
- **TONETMASK(PREFIX)**: macro che trasforma il prefisso PREFIX in netmask esplicita, cioè in un intero senza segno in cui i più significativi PREFIX bits sono a 1 e i rimanenti a 0. E' implementata nel seguente modo:

  ```
  (((PREFIX)==0)?(0):(MAXINT<<(32-(PREFIX))))
  ```

2.Introduzione: il concetto di CIDR

Il metodo originario per l'indirizzamento di reti Internet

Il metodo originario usato in Internet per l'assegnazione degli indirizzi IP si basava sul concetto di **classe** di indirizzo e si attuava in base al numero di bit assegnato alla parte **netID** e alla parte **hostID** dell'indirizzo, utilizzando il seguente schema:

Classe di indirizzo	**Nr bit di netID**	**Nr bit di hostID**	**Range decimale del primo byte dell'indirizzo**
Classe A	8 bit	24 bit	1-126
Classe B	16 bit	16 bit	128-191
Classe C	24 bit	8 bit	192-223

Usando questo schema, si poteva arrivare a supportare al massimo:

- 126 reti di classe A che potevano includere fino a 16,777,214 hosts ognuna
- 64K reti di classe B che potevano includere fino a 65,534 hosts ognuna
- oltre due milioni di reti di classe C che potevano includere fino a 254 hosts ognuna
- alcuni range/classi riservati per multicast, broadcast, riservati per usi futuri

Questo metodo di indirizzamento faceva sorgere principalmente due problemi:

1. non era possibile ottenere altro che questi blocchi di indirizzi di rete, con conseguente spreco di indirizzi. Se, ad esempio, servivano 100 indirizzi, andavano sprecati i rimanenti 155
2. le tabelle di routing dei router avevano ormai raggiunto dimensioni enormi: siccome non c'era modo di aggregare le reti adiacenti, per ogni nuova rete era necessaria una nuova linea in tabella.

Il metodo attuale

L'introduzione del CIDR (Classless Inter-Domain Routing) si è resa necessaria a partire dal 1993 con l'esplosione del “fenomeno” Internet e il conseguente moltiplicarsi di reti in giro per il mondo.

Il metodo attuale di indirizzamento consiste nell'associare all'indirizzo della rete un campo ulteriore che specifica quanti (o quali, a seconda del metodo utilizzato) sono i

bit di rete. Per differenza si ottengono quelli hosts.

Due notazioni non sono equivalenti in linea teorica, ma si equivalgono nell'uso comune:

1. notazione prefissa: l'indirizzo di rete viene indicato come A.B.C.D/N. In essa A-D sono quattro decimali rappresentanti gli ottetti dell'indirizzo (notazione detta **dotted quad**), in cui sono indicati solo i bit di rete (i bit di hosts sono sempre a 0) . N è un numero decimale compreso fra 1 e 32, indicante quanti bit sono di rete (NN=32 indica che la rete è composta da un solo host).

   Esempio: 10.20.30.0/23

2. notazione con netmask: l'indirizzo di rete viene indicato come A.B.C.D E.F.G.H, dove A-D sono la dotted quad dell'indirizzo, in cui sono indicati solo i bit di rete (i bit di hosts sono sempre a 0) ed E-F sono la dotted quad della netmask. L'operazione binaria di AND fra un qualunque indirizzo IP e la netmask permette di ottenere l'indirizzo di rete "presente" nell'indirizzo IP.

   Esempio equivalente al precedente: 10.20.30.0 255.255.254.0

Da notare che la seconda notazione è più potente, in quanto permette di avere bit di rete anche dopo alcuni bit di hosts.

Nella pratica e per quanto stabilito dall'RFC 1518, la netmask deve specificare in ogni caso un prefisso di rete, quindi non ci sono differenze reali di espressività.

Nel seguito con il termine “notazione CIDR” si intenderà una notazione prefissa.

3.Esigenza iniziale: implementazione di blacklist

Esigenze funzionali

Per arginare il fenomeno dello spamming, si sono costituite in Internet degli enti che censiscono gli indirizzi IP/reti generanti spamming, ne pubblicano l'elenco (alcuni gratuitamente, altri a pagamento) e formano così una “blacklist”.

Tale elenco può essere utilizzato dai mail server, opportunamente istruiti, per effettuare un lookup via protocollo DNS-like: se l'IP di provenienza della connessione SMTP è presente nell'elenco della blacklist, la connessione verrà rifiutata.

Esigenze implementative

L'implementazione tipica richiesta dai servizi di posta elettronica di grandi dimensioni ha i seguenti requisiti:

- utilizzare diverse blacklist provenienti da diversi fornitori di informazioni e aggregarle in una sola, in modo tale da eseguire una sola lookup al DNS
- evitare che, per qualche motivo, un IP/rete di quelle “amiche”,la propria o quella di altri gestori conosciuti, finisca nelle blacklist importate dall'esterno, bloccando i propri clienti. Tale elenco di reti costituisce la “whitelist”

- Dare la possibilità al gestore del servizio di avere una propria whitelist che abbia la precedenza su tutte le blacklist.

Esigenze prestazionali

Le prestazioni, sia in termini di memoria utilizzata che in termini di tempo di esecuzione, sono molto importanti per questa applicazione perché la mole di dati da trattare è molto grande (milioni di elementi) e l'aggiornamento molto frequente delle blacklist obbliga ad una esecuzione frequente (l'esecuzione tipica avviene su base oraria).

Inoltre, avere a disposizione una blacklist finale minima (cioè con le reti aggregate) consente un notevole risparmio di memoria e CPU dei DNS server, oltre ad una maggiore maneggiabilità dei files della stessa.

4.Ricerca di una soluzione esistente

Verificate le esigenze di utilizzo, si è effettuata una ricerca in Internet sulla disponibilità di un software che implementasse quanto richiesto. La ricerca, tuttavia, ha dato esito negativo, non avendo trovato nessun software che soddisfacesse le esigenze di cui sopra.

Ci si è quindi concentrati nella ricerca, sia su Internet che in letteratura scientifica, di un algoritmo risolutivo del problema dato.

La ricerca ha portato alla luce che la risoluzione del problema della sola riduzione delle reti (e non della applicazione di una whitelist) è possibile e auspicabile (è definita opzionale) nei router che implementano il protocollo BGP4 (RFC 1771).

Nell'RFC stesso esiste una specifica che definisce ciò che può essere o non essere aggregato. La descrizione è ad alto livello e non definisce un metodo procedurale preciso con lo scopo di lasciare spazio agli implementatori.

E' noto che i router Cisco implementano tale algoritmo anche se, trattandosi di software commerciale, non è stato possibile reperire i dettagli a riguardo.

Si è quindi deciso di procedere a uno studio di fattibilità e all'implementazione originale di un algoritmo che soddisfi i requisiti posti.

5.Formalizzazione requisiti funzionali

Definizioni

Alla luce di quanto emerso come esigenza pratica, il problema che si vuole risolvere viene qui di seguito formalizzato.

Definiamo innanzi tutto il concetto di *rete* come *insieme di hosts espressi con un solo elemento di rete.*

Dato in input due insiemi di reti chiamati insieme "I"(la blacklist) e insieme W (la

whitelist; può essere vuoto), trovare un insieme "O" che soddisfi le seguenti condizioni:

1. *l'insieme* $O = I \cap \overline{W}$ *, cioè l'insieme O contiene tutte e sole le reti di I che non appartengono a W*
2. *O è l'insieme minimo, cioè se esiste un altro insieme* $O1 \neq O$ *che soddisfa la condizione 1, allora il numero di elementi di rete di O1 è maggiore di, o uguale a, quelli di O.*

Tra gli elementi degli insiemi di reti sono definite delle proprietà di **riduzione** che permettono di stabilire quando una rete può essere eliminata oppure quando due o più reti possono essere sostituite da una terza, per arrivare a soddisfare la condizione 2 (vedi paragrafo successivo).

Da notare che, dato un insieme di hosts, l'insieme di reti (e di elementi di rete) minimo corrispondente è unico.

Questa conseguenza è dovuta principalmente a due caratteristiche:

1. se due reti rappresentano lo stesso insieme di hosts, l'elemento di rete corrispondente è identico, cioè le due reti sono uguali
2. le proprietà di riduzione non sono influenzate dall'ordine di applicazione, cioè a prescindere dalla sequenza di applicazione si arriverà sempre alle stesse reti.

Dalla combinazione di queste due caratteristiche nasce l'univocità: la riduzione ci conduce obbligatoriamente ad un certo insieme di reti e le stesse sono rappresentate ognuna univocamente da un elemento di rete.

Proprietà di conflitto

Due reti R1 ed R2 sono in **conflitto** quando parte degli hosts di R1 appartiene ad R2 o viceversa.

Se R1 ed R2 appartengono rispettivamente alla blacklist e alla whitelist violano la condizione 1. del paragrafo precedente.

Siccome le reti sono definite tramite prefisso, come vedremo nel paragrafo "6. Idee implementative di base", due reti in conflitto sono necessariamente l'una sottorete dell'altra.

Proprietà di riduzione

Le proprietà fondamentali di riduzione sono:

1. **eliminazione delle sottoreti**: una rete R1 si dice sottorete di una seconda rete R2 quando tutti gli hosts di R1 sono anche contenuti in R2. Due reti uguali sono sottoreti (improprie) l'una rispetto all'altra. La sottorete R1 può quindi essere eliminata dall'insieme
2. **aggregazione**: operazione che consiste nel sostituire alla rete R1 e alla rete R2 una

nuova rete R3 che contiene tutti e soli gli hosts contenuti in R1 e R2.

Le proprietà fondamentali di riduzione vengono definite rispetto a due reti per questioni di comodità.

Un insieme di reti O è definito **minimo** quando non è più possibile applicare nessuno dei sopracitati criteri per nessuna coppia di elementi dell'insieme O.

L'ovvia conseguenza di questa definizione è che se O è vuoto o ha un solo elemento allora è un insieme minimo.

Da notare che le definizioni di cui sopra vanno applicate per passi successivi: anche le reti già soggette ad aggregazione possono venire eliminate o nuovamente aggregate fino all'ottenimento dell'insieme minimo.

6.Idee implementative base

Formato degli elementi di rete

Le reti espresse in notazione con prefisso sono nella forma “A.B.C.D/N”. La notazione con netmask è equivalente.

La notazione con prefisso è la più comoda ed efficiente da implementare sul computer, in quanto un elemento di rete è descritto da un intero senza segno di 32 bit e da un byte che rappresenta un numero senza segno. Il primo, che chiameremo “network”, rappresenta A.B.C.D e il secondo, detto “prefix”, contiene N.

E' definita una macro “TONETMASK(X)” che è in grado di trasformare il prefisso X in netmask esplicita, cioè in un intero senza segno in cui i più significativi X bits sono a 1 e i rimanenti a 0.

In seguito, quando si parlerà di netmask, si intenderà l'intero senza segno ottenuto applicando la macro TONETMASK al valore di prefix.

I bit di network in corrispondenza con bit a zero di TONETMASK(N) devono essere a zero; altrimenti la rappresentazione è illegale.

I due interi senza segno saranno così valorizzati:

- *network:* il valore binario di A viene salvato nei primi 8 bit dell'intero (i più significativi), quello di B nel secondo ottetto, C nel terzo e D nel quarto
- *prefix*: viene valorizzato direttamente con il valore di N.

Esempio:

Notazione con prefisso: `10.50.120.128/28`.
Rappr. intera network: 171079808
Rappr. intera netmask: 28
Rappr. binaria network: `00001010 00110010 01111000 10000000`
TONETMASK(28): `11111111 11111111 11111111 11110000`

Il vantaggio di questo metodo di memorizzazione è che tutte le sottoreti di una rete avranno un valore di rete `>=` del valore della rete a cui appartengono (secondo la definizione matematica di confronto fra numeri naturali).

Si verificheranno in seguito altre proprietà di questa rappresentazione.

Da notare che il programma sviluppato non funziona (c'è un controllo esplicito) se la dimensione dell'intero senza segno è diversa da 32 bit. Tutte le piattaforme testate soddisfano comunque tale requisito.

Nel seguito, quando si parlerà di intero, si intenderà intero senza segno, a meno che non sia diversamente specificato.

Implementazione della proprietà di conflitto

La notazione C-like della proprietà di conflitto sopra enunciata può essere così definita:

- `supermask=TONETMASK(r1.prefix)&TONETMASK(r2.prefix);`
  conflitto se`(r1.network&supermask)==(r2.network&supermask)`

L'implementazione definisce prima di tutto una netmask `supermask,` scegliendo la minore fra quella di r1 e r2 usate per verificare se i prefissi di rete risultanti dalla sua applicazione sono uguali, ovvero se r1 è sottorete di r2 o viceversa.

Esempio: r1.network=10.10.10.45, r1.prefix=32, r2.network=10.10.10.0, r2.prefix=24. In questo caso r1 e r2 sono in conflitto in quanto r1 è sottorete di r2.

Implementazione delle proprietà di riduzione

La notazione C-like delle proprietà di riduzione sopra enunciate può essere così definita:

1. **eliminazione delle sottoreti**: se

   `((r2.network & TONETMASK(r1.prefix) == r1.network)`

   allora r2 può essere eliminato

2. **aggregazione**: se

   `( (r1.prefix == r2.prefix) && (r3.prefix == (r1.prefix -1) ) && (r1.network & TONETMASK(r3.prefix)) == (r2.network & TONETMASK(r3.prefix)) == r3.network )`

   `allora` r3 può rimpiazzare sia r1 che r2.

La prima proprietà è implementata in maniera intuitiva: si verifica che r2 appartenga tutta a r1 confrontando i valori di rete a parità di netmask.

La seconda è più articolata: si esegue soltanto l'aggregazione di reti con netmask uguali, aggregandole a due a due quando fra due cambia solo l'ultimo bit di rete.

Questo metodo permette comunque di arrivare ad una aggregazione completa tramite step successivi.

Esempi:

- r1.network=10.10.10.0 e r1.prefix=24, r2.network=10.10.10.1 e r2.prefix=32.

  E' chiaro che, in questo caso, r1 contiene anche r2 e quindi r2 può essere omesso senza perdita di informazione.

- r1.network=10.10.10.0 e r1.prefix=24, r2.network=10.10.11.0 e r2.prefix=24.

  In questo caso si possono rimpiazzare sia r1 che r2 con r3.network=10.10.10.0 e con r3.prefix=23.

- Si veda il capitolo “16. Esempio di esecuzione” relativo alla funzione optimize per analizzare un caso di aggregazione per step successivi.

Come si può notare, il calcolo dei criteri di cui sopra è molto veloce perché consiste in confronti fra interi e in operazioni bit a bit.

Genesi dell'algoritmo

Enunciate le idee implementative base, ci si può domandare se sia possibile arrivare ad un algoritmo risolutivo eseguibile in tempo polinomiale.

La risposta è positiva, come vedremo nella successiva analisi della complessità.

Per arrivare a tale risultato, il problema da risolvere è stato distinto in due parti:

1. eliminare da I gli elementi che appartengono all'insieme W, cioè applicare la whitelist
2. ottenere un insieme O minimo, cioè applicare le proprietà di riduzione.

Criterio di confronto fra reti (Criterio di ordinamento)

Viene introdotto un concetto di ordinamento completo fra gli elementi degli insiemi (cioè fra le reti),

Dati $r1, r2 \in I$ r1 < r2 se:

1. r1.network < r2.network
2. r1.nework = r2.network e r1.prefix < r2.prefix

Con gli operatori di minore e uguale si intendono i normali criteri di confronto fra numeri naturali.

E' possibile di conseguenza dire che r1 = r2 se e solo se r1.network = r2.network e

r1.prefix = r2.prefix.

L'ordinamento degli elementi è fondamentale per consentire l'esecuzione in tempo lineare delle due procedure base dell'algoritmo.

Struttura base dell'algoritmo

Lo scheletro base è costituito dalle seguenti funzioni:

- parsing dell'input (**get_entries**)
- sort degli array (**sort_entries**)
- applicazione della whitelist per l'eliminazione delle reti in conflitto (**apply_whitelist**)
- calcolo dell'insieme minimo (**optimize**)
- stampa del risultato (**print_addresses**)

7.Implementazione e correttezza

L'implementazione di tale algoritmo, come da requisito, è stata eseguita in un'ottica di performance sia asintotica sia di tempo di esecuzione totale e utilizzo di memoria.

Per tale motivo, è stato scelto di utilizzare il C come linguaggio di programmazione, poiché, permettendo un controllo diretto delle risorse macchina, offre maggiori leve per ottenere un'ottimizzazione prestazionale efficace.

Ha ulteriormente contribuito a questa scelta la caratteristica che l'applicativo non richieda interfaccia grafica, ma si configuri come libreria “core” di altri programmi e possa venire richiamato in ambito di shell scripting.

In ogni caso non sono state trascurate le regole base di buona programmazione: divisione funzionale, analisi di correttezza, riutilizzabilità.

Vediamo nel dettaglio le strutture e le funzioni base, lasciando al seguito la discussione sul programma nel suo insieme.

Parsing dell'input

Il programma in input supporta un certo numero di formati, che verranno trattati in seguito nella discussione generale sul programma.

E' fondamentale, per la correttezza delle funzioni base del programma, l'effettuazione di un controllo formale sulla correttezza dei dati in input e, in particolare, che l'accoppiata network/prefix sia congrua.

A questo scopo, si verifica che nessun bit del "campo host” sia a 1. Ciò rende sempre vera l'invariante base per la correttezza del programma, trattata nel successivo paragrafo “Proprietà base”.

Strutture dati

Per la rappresentazione degli attributi “network” e “prefix” viene utilizzata una struct che definisce il tipo degli elementi degli array che compongono gli insiemi I, W e O.

Di seguito la definizione precisa in notazione C-like:

```
struct entry
{
     unsigned int network;
     unsigned char prefix;
};
```

Esempio 1: array non ordinato

Le colonne “Netmask”, “Network (dotted)” e “Netmask (dotted)” sono riportate in tabella a fine esemplificativo, ma all'interno del programma si fa uso dei soli campi “Network” e “Prefix”.

Pos. Array	**Network (dotted)**	**Network**	**Prefix**	**Netmask**
0	20.0.0.0	335544320	8	255.0.0.0
1	192.92.104.0	3227281408	22	255.255.252.0
2	10.20.21.0	169088256	24	255.255.255.0
3	10.20.20.128	169088128	25	255.255.255.128
4	1.2.3.4	16909060	32	255.255.255.255
5	10.20.20.0	169088000	25	255.255.255.128
6	202.2.4.16	3389129744	28	255.255.255.240
7	1.2.0.0	0	16	255.255.0.0
8	192.92.105.4	3227281668	30	255.255.255.252

Proprietà base

La proprietà base del programma è la seguente: se una rete X è sottorete di Y, si è verificata una di queste due condizioni:

- il valore di network di X è maggiore di quello di Y
- il valore di network di X è uguale a quello di Y e il valore di netmask di X è maggiore di quello di Y.

Inoltre, se la rete Z non è sottorete di X e il valore di network di Z è maggiore di quello di Z, allora essa sarà maggiore anche di tutte le sottoreti di X.

Sort degli array

Per il sorting degli array si era deciso inizialmente di utilizzare il quicksort standard incluso nel linguaggio C.

Successivamente, siccome il tempo impiegato dall'ordinamento era significativamente alto, si è introdotta la possibilità di utilizzare una versione modificata avente come base delle librerie standard da cui si differenzia per migliorie implementative. La funzione di confronto è più snella e può essere specificata inline.

Entrambi gli algoritmi si basano su una funzione di ordinamento custom “cmp_entry”, leggermente diversa a seconda dell'uso dell'una o dell'altra funzione, ma che rispecchia in ogni caso le specifiche di ordinamento sopra definite.

Vengono ordinati sia l'array con gli elementi di I che quello con gli elementi di W.

In base a come si è definito gli elementi dell'array, è facile verificare che, per stabilire se una rete è maggiore di un'altra, è sufficiente eseguire i confronti specificati tramite semplice confronto fra interi di 32 e 8 bits.

Questo rende l'operazione estremamente rapida, in quanto tutti i processori attuali la implementano in hardware.

Esempio 2: array ordinato

Pos. Array	**Network (dotted)**	**Network**	**Prefix**	**Netmask**
0	1.2.0.0	16908288	16	255.255.0.0
1	1.2.3.4	16909060	32	255.255.255.255
2	10.20.20.0	169088000	25	255.255.255.128
3	10.20.20.128	169088128	25	255.255.255.128
4	10.20.21.0	169088256	24	255.255.255.0
5	20.0.0.0	335544320	8	255.0.0.0
6	192.92.104.0	3227281408	22	255.255.252.0
7	192.92.105.4	3227281668	30	255.255.255.252
8	202.2.4.16	3389129744	28	255.255.255.240

NOTA: le posizioni degli array con prefix = INVALID_PREFIX sono considerate invalide e pertanto possono essere ignorate in fase di ordinamento.

8.Funzioni base: applicazione della whitelist

Pseudo codice

Le parti in grassetto-italico non sono espresse in C.

```
1.apply_whitelist(entry[] addr,entry[] white,bool do_sort)
2.{
3.   if (do_sort)
4.   {
5.         sort_entries(addr);
6.         sort_entries(white);
7.   }
8.
9.   i1=0;
10.  i2=0;
11.  expanded_position=0;
12.  while (i1<lenght(addr) && (i2<lenght(white))
13.  {
14.        if (addr[i1] è in conflitto con white[i2])
15.        {
16.              if (addr[i1] non è sottorete di white[i2])
17.              {
18.                    invalid=0; tmp=i1+1;
19.                    while (i1<lenght(addr) && addr[tmp] è
  sottorete di addr[i1])
20.                    {
21.                          addr[i1][tmp].prefix=INVALID_PREFIX;
22.                          tmp++;
23.                          invalid++;
24.                    }
25.                    len_white=numero di elementi di white in
  conflitto con addr[i1];
26.                    addr[i1].network=expanded_position;
27.
  addr[i1].prefix=EXPANDED_PREFIX+white[i2].prefix-addr[i1].prefix;
28.
  expand(expanded,addr[i1],&(white[i2]),len_white,white[i2].prefix-
  addr[i1].prefix,&expanded_position);
29.                    i1+=invalid+1;
30.                    i2+=len_white;
```

```
31.             }
32.             else
33.             {
34.                   addr[i1].prefix=INVALID_PREFIX;
35.                   i1++
36.             }
37.       }
38.       else
39.       {
40.             if (addr[i1] minore white[i2])
41.             {
42.                   i1++;
43.             }
44.             else
45.             {
46.                   i2++;
47.             }
48.       }
49.  }
50.}
51.
52.expand(entry[] expanded,entry to_expand,entry[] white,unsigned
  int len_white,int n,unsigned int *current_position)
53.{
54.  int first=*current_position, last=*current_position + n-1;
55.  struct entry tmp_1,tmp_2;
56.
57.  *current_position+=n;
58.  while(first<=last)
59.  {
60.       tmp_small.network=to_expand.network;
61.       tmp_small.prefix=to_expand.prefix+1;
62.       tmp_big.network=to_expand.network con l'ultimo bit di
  rete a 1;
63.       tmp_big.prefix=to_expand.prefix+1;
64.
65.       if (tmp_small è in conflitto con white[0])
66.       {
67.             if (tmp_big non è in conflitto con nessun elemento
  di white con indice > 0)
```

```
68.             {
69.                   expanded[last]=tmp_big;
70.             }
71.             else
72.             {
73.                   expand(expanded,tmp_big,elementi di white
  non in conflitto con tmp_big,white[0].prefix-
  tmp_big.prefix,current_position);
74.             }
75.             to_expand.network=tmp_small;
76.             last--;
77.       }
78.       else
79.       {
80.             if (tmp_small non è in conflitto con nessun
  elemento di white con indice > 0)
81.             {
82.                   expanded[first]=tmp_small
83.             }
84.             else
85.             {
86.                   expand(expanded,tmp_small,elementi di white
  non in conflitto con tmp_small,white[0].prefix-
  tmp_small.prefix,current_position);
87.             }
88.             first++;
89.             to_expand=tmp_big;
90.       }
91.  }
92.}
```

Precondizioni

La funzione prende in input due array di reti (di tipo struct entry *), uno chiamato *addr* che contiene le reti in input e un altro chiamato *white* che contiene le reti della whitelist.

La funzione apply_whitelist necessita dell'ordinamento degli elementi di entrambi.

Tale ordinamento deve essere eseguito prima dalla chiamata oppure può essere eseguito dalla funzione stessa, a seconda della valorizzazione dell'ultimo parametro do_sort.

Analisi della funzione: idee base e invariante ciclo principale

L'idea che sta alla base dell'algoritmo è la produzione in output di un array che contenga tutte le reti in input che non sono sottoreti di whitelist.

Per ragioni di performance tale risultato deve essere ottenuto senza una scansione multipla né dell'array addr né di quello white e senza utilizzare un array di appoggio in cui copiare tutti gli elementi in output, bensì modificando l'array in input ed utilizzando un array di appoggio, chiamato expanded, per ospitare le sole reti espanse.

Anche se gli array in output sono due, essi ne rappresentano logicamente uno solo, come vedremo sotto.

Si introducono gli indici i1 e i2, rispettivamente l'indice della posizione corrente dell'array addr (la blacklist) e dell'array white (la whitelist).

Si definisce il concetto di posizione allocata ma non valida quando il prefisso è uguale a INVALID_PREFIX, che permette di invalidare degli elementi senza dovere ricompattare l'array, un'operazione ovviamente molto costosa.

Gli elementi invalidi non vengono considerati ai fini di tutte le proprietà ed invarianti sotto descritte.

Si definisce inoltre il concetto di indirizzo di rete puntatore: tale posizione all'interno dell'array addr non definisce un vero elemento, bensì un puntatore logico ad una serie di reti che si trovano nell'array expanded.

Logicamente, gli elementi “puntati” si trovano inseriti a partire dalla posizione del puntatore stesso. In seguito, quando definiremo le proprietà degli elementi di addr, esse saranno valide sostituendo agli elementi puntatore tutti gli elementi puntati.

Il metodo di indirizzamento e di espansione verrà descritto in dettaglio fra poco nel paragrafo “Espansione di una rete”.

La scelta delle azioni da intraprendere viene effettuata utilizzando esclusivamente i valori di addr[i1] e di white[i2].

Affinché ciò sia possibile, l'ordinamento ha una funzione chiave: la decisione se avanzare di posizione nell'uno o nell'altro array oppure di espandere/invalidare una rete in conflitto é sempre presa guardando i valori delle sole posizioni correnti, senza curarsi di quello che viene prima o dopo.

L'invariante del ciclo principale della funzione può essere così definita in tre predicati:

1. L'array addr contiene fino alla posizione i1 (esclusa) l'insieme di hosts rappresentati nell'array in input fino alla posizione i1 stessa, tranne quelli che sono in conflitto con elementi in white.
2. [(addr[i1] > white[i2]) e (white[i2-1] <= addr[i1-1] <= white[i2])] oppure [(addr[i1] <= white[i2]) e (white[i2-1] <= addr[i1] <= white[i2])] (<= secondo il criterio di confronto fra reti). In caso l'indice sia

```
   negativo oppure sia oltre l'ultimo elemento degli
   array, la condizione corrispondente viene considerata
   valida.
3. Tutti gli elementi di addr e white sono ordinati
   secondo il criterio di confronto fra reti.
```

La proprietà 1 ci assicura che fino alla posizione i1 tutti gli hosts compresi nell'array white sono stati eliminati.

Ciò può avvenire invalidando l'elemento oppure ricorrendo all'espansione. Entrambe le operazioni creano una “anomalia” nell'array, che però è visto come serie di elementi contigui, utilizzando le accortezze descritte sopra.

La proprietà 2 serve a garantire che l'esplorazione di addr e white avvenga in maniera tale che tutti gli eventuali conflitti saranno rilevati: siccome la condizione necessaria affinché ci sia un conflitto è che l'elemento di addr sia sottorete dell'elemento di white o viceversa, essa assicura che addr[i1] e white[i2] siano sempre il più “vicino” possibile(dal punto di vista dell'ordinamento).

La proprietà 3 serve a garantire che l'array sia sempre ordinato, altra condizione necessaria per la correttezza.

Analisi della funzione: correttezza

Prima di tutto, qualora richiesto, viene eseguito il sorting degli array (linee 3-7).

In seguito, vengono inizializzati i due indici i1, i2 ed expanded_position a 0.

Quindi inizia il ciclo principale (linee 12-48): la condizione 1. è ovviamente valida all'inizio (i1 vale 0, quindi, essendo questa posizione esclusa e non essendo ancora stati presi in considerazione elementi dell'array di input, l'insieme vuoto soddisfa il predicato).

La condizione 2 è anch'essa ovviamente valida, in quanto i1 è stato appena inizializzato a 0 e la condizione è sempre vera in questo caso.

Anche la condizione 3. lo è in quanto, a seconda del valore del parametro do_sort, l'ordinamento è una precondizione oppure è eseguito dalla funzione stessa come primo passo e gli array sono rimasti invariati.

Della precondizione 3. puntualizzeremo in seguito solo il mantenimento dell'ordinamento di addr, in quanto l'array white non verrà mai modificato e rimarrà quindi sempre ordinato.

Alla linea 14 viene rilevato se esiste un conflitto nelle posizioni correnti di i1 e i2, che viene trattato in modo diverso, a seconda che:

- addr[i1] non sia sottorete di white[i2] (di conseguenza white[i2] è sottorete propria di add[i1]) e quindi addr[i1] abbia solo parte degli indirizzi in whitelist (per esempio addr[i1]=10.10.10.0/24 e white[i2]=10.10.10.1/32) (linee 17-20).
  In questo caso, viene espansa la rete addr[i1] che verrà sostituita con il puntatore logico agli elementi che si trovano nell'array expanded.

La logica che sta dietro questa operazione è la seguente: siccome addr[i1] ha solo parte degli indirizzi in whitelist, per esprimere tutti e solo gli hosts che non sono appartenenti a white[i2], si dovrà generare un numero di elementi di rete che va da 1 ad un massimo di 32. Per evitare di spostare gli elementi nell'array addr viene utilizzato un array di appoggio expanded.
Gli hosts nell'array expanded, oltre a non contenere elementi in white[i1], non conterranno neanche elementi di eventuali altre sottoreti in conflitto con addr[i1], in quanto expanded le prende in considerazione tutte e le esclude applicando nuovamente l'espansione, che è una funzione ricorsiva.
L'array expanded potrà quindi contenere altri puntatori logici che puntano a posizioni successive dello stesso.
L'espansione è fatta in modo tale da generare un sottoinsieme minimo: gli elementi espansi non saranno ulteriormente considerati nel prosieguo della funzione e neanche dalla funzione di ottimizzazione.
Bisogna però accertarsi che non esistano delle sottoreti di addr[i1] in posizioni immediatamente successive, che rappresenterebbero hosts duplicati e che renderebbero non vera l'affermazione precedente. Tali posizioni vengono invalidate (linea 34).
L'indice i1 verrà incrementato del numero degli elementi invalidati + 1, per consentire di andare oltre a tutti gli elementi scartati più l'elemento addr[i1] stesso.
I2 viene incrementato di tutti gli elementi di white in conflitto con addr[i1] (come minimo uno, essendo white[i2] sicuramente in conflitto).
Per le proprietà dell'espansione spiegate sopra (sia gli elementi addr[i1] che white[i2]sono stati completamente considerati) la condizione 1. dell'invariante continua a valere.
In virtù dell'ordinamento e della validità della condizione all'ingresso l'incremento di i1 e di i2 mantiene valida anche la condizione 2.
Inoltre, siccome gli elementi espansi sono ordinati, anche la 3 è mantenuta.

- addr[i1] sia sottorete (anche impropria) di white[i2]. In questo caso, viene semplicemente invalidata (linea 34) e viene incrementato i1 (linea 35).
  L'invariante continua a valere: addr[i1] era in confitto ma è stata invalidata, quindi l'incremento di i1 non la viola. Siccome gli elementi invalidi non vengono considerati, anche la condizione 2 continua a valere. Inoltre, un elemento invalido è per definizione sempre ordinato, quindi anche la terza condizione è rispettata.

Se, invece, le posizioni correnti non sono in conflitto (linee 40-47), allora verranno incrementati i1 o i2, secondo il criterio di ordinamento.

Il punto 1. dell'invariante continuerà a valere, in virtù del fatto che, per la proprietà dell'ordinamento enunciata prima, se una sottorete x, appartenente ad addr, di y appartenente a white esiste in addr allora x sarà “maggiore” (e quindi in una posizione successiva dell'array) rispetto a y. Sarà “minore” nel caso opposto.

Per la sopracitata scelta di avanzamento, anche il punto 2. sarà ancora valido.

Anche il punto 3. continuerà ovviamente a valere in quanto gli array non sono stati

modificati.

La condizione di terminazione è doppia:

- si sono analizzati tutti gli elementi di addr. In questo caso, la parte 1. dell'invariante è ovviamente soddisfatta all'uscita dal ciclo, in quanto in virtù dell'ordinamento tutte le reti in conflitto sono state prese in considerazione. La parte 2. è valida in virtù del metodo utilizzato per gli incrementi di i1 e i2. La parte 3. è valida in quanto l'ordinamento è sempre stato mantenuto ad ogni passo.
- si sono analizzati tutti gli elementi di white. Ovvio che non ci siano più conflitti (parte 1.), che la parte d2.valga (si può assumere white[i2] essere più grande di qualsiasi elemento addr quando i2 è oltre la lunghezza massima dell'array white) e che l'array addr sia ancora ordinato (parte 3.).

Espansione di una rete

L'espansione di una rete è un'operazione necessaria quando si verifica un conflitto fra una rete di addr e una di white e l'elemento di white è sottorete propria di quello di addr. Esempio: addr[i1]=10.10.10.0/24 e white[i2]=10.10.10.1/32.

In questo caso si è implementata una funzione che ha le seguenti caratteristiche:

1. produce in output un insieme di reti che comprende tutti e soli gli hosts che non erano in comune con tutte le reti di white in conflitto
2. produce in output un sottoinsieme minimo anche rispetto a tutte le altre reti eventualemente risultanti in output
3. produce in output un sottoinsieme ordinato.

Per arrivare a soddisfare la condizione 1. si è dovuto prendere in considerazione nella funzione stessa tutte le eventuali altre sottoreti di addr[i1] presenti in white.

Esempio:

addr[i1]=10.10.10.0/24, white[i2]=10.10.10.1/32, white[i2+1]=10.10.10.2/32, white[i2+2]=10.10.10.3/32.

L'idea della funzione è la seguente: dato un addr[i1] e un white[i2] in conflitto, si mette in to_expand il valore di addr[i1].

Qui comincia il ciclo: si calcolano due reti temporanee, entrambe con prefisso uguale a to_expand+1: tmp_small, che ha lo stesso indirizzo di rete di to_expand e tmp_big, che ha l'ultimo bit di rete a 1.

Esempio:

tmp_small=10.10.10.0/25, tmp_big= 10.10.10.128/25.

Dato che white[i2] è sottorete di to_expand, solo uno fra tmp_small e tmp_big sarà in conflitto con white[i2]: assumiamo che sia tmp_low (se fosse tmp_big basterebbe invertire le parti nelle considerazioni successive).

Se tmp_big non è in conflitto con nessun altro elemento di white, allora lo stesso verrà salvato senza ulteriori controlli (né al momento né successivamente) nell'array expanded.

Se tmp_big è in conflitto con qualche altro elemento di white, allora verrà richiamata nuovamente la funzione di espansione con parametri tmp_big e tutti gli elementi di white in conflitto con tmp_big stesso.

La funzione ripete poi il ciclo, mettendo in to_expand tmp_low fino a quando il prefisso di to_expand non è uguale a quello di white[i2]. In base a come abbiamo generato to_expand, a questo punto anche il valore di rete sarà identico e la funzione può quindi terminare.

L'ordinamento è ottenuto osservando che a pari prefisso tmp_big è più grande di tmp_low e che se per due cicli si salva nell'array tmp_big il primo sarà più grande del secondo in virtù del fatto che, come elemento to_espand, dopo il primo è stato utilizzato l'elemento tmp_low.

Dal punto di vista strettamente implementativo, siccome il numero di elementi espansi è normalmente maggiore di 1 e dal momento che si voleva evitare di inserirli nel mezzo dell'array (operazione che implicherebbe lo spostamento di tutti gli elementi successivi verso il basso), si è scelto di utilizzare un array di appoggio expanded che li contiene tutti (compresi quelli delle espansioni ricorsive).

Per essere in grado di inserire logicamente gli elementi espansi nel posto corretto (cioè al posto dell'elemento espanso) e in maniera efficiente, è stata introdotto il concetto di posizione puntatore.

La posizione puntatore ha le seguenti caratteristiche:

- il campo prefix contiene il valore EXPANDED_PREFIX+<numero di elementi espansi>
- il campo network contiene la posizione del primo elemento espanso nell'array addr.

Con le suddette informazioni è possibile ricostruire l'array di output completo, scorrendone gli elementi ed utilizzando all'evenienza gli array di expanded quando necessario. L'operazione ha un tempo lineare sul numero di elementi totali.

Da notare le conseguenze delle caratteristiche dell'espansione:

- siccome il sottoinsieme espanso è minimo, allora potrà essere ignorato in fase di funzione di ottimizzazione
- siccome il sottoinsieme espanso è ordinato, allora anche l'invariante di ordinamento di tutti gli elementi continua ad essere valida.

Date le conseguenze di cui sopra, le reti in expanded non verranno più considerate né nel seguito della funzione apply_whitelist né nella funzione di riduzione, bensì solamente in fase di stampa finale dell'output.

L'unione logica degli array addr ed expanded verrà in seguito indicata come array

addr+expanded.

Postcondizioni

Dall'invariante di ciclo all'uscita dalla funzione si avrà che:

1. l'array addr+expanded non contiene elementi in conflitto con alcun elemento di white
2. gli array addr, expanded, addr+expanded e white sono ordinati.

La postcondizione 2., per quanto riguarda il solo addr, verrà utilizzata per evitare un secondo ordinamento degli array nell'ottimizzazione (computazionalmente molto costoso), operazione che logicamente viene dopo.

Esempi

Esempio 3: array di whitelist ordinato

Pos. Array	Network (dotted)	Network	Prefix	Netmask
0	20.30.40.0	337520640	24	255.255.255.0
1	192.92.105.180	3227281844	32	255.255.255.255

Esempio 4: array addr dopo l'esecuzione della funzione apply_whitelist, usando gli elementi dell'esempio 1 come input e dell'esempio 3 come whitelist. Si ricorda che INVALID_NETMASK vale 33 e EXPANDED_PREFIX vale 220.

Pos. Array	Network (dotted)	Network	Prefix	Netmask
0	1.2.0.0	16908288	16	255.255.0.0
1	1.2.3.4	16909060	32	255.255.255.255
2	10.20.20.0	169088000	25	255.255.255.128
3	10.20.20.128	169088128	25	255.255.255.128

Pos. Array	Network (dotted)	Network	Prefix	Netmask
4	10.20.21.0	169088256	24	255.255.255.0
5	20.0.0.0	335544320	8	255.0.0.0
6	N/A	0	228	N/A
7	N/A	3227281668	33	N/A
8	202.2.4.16	3389129744	28	255.255.255.240

Esempio 5: array expanded dopo l'esecuzione della funzione apply_whitelist, usando gli elementi dell'esempio 1 come input e dell'esempio 3 come whitelist.

Pos. Array	Network (dotted)	Network	Prefix	Netmask
0	192.92.104.0	3227281408	24	255.255.255.0
1	192.92.105.0	3227281664	25	255.255.255.128
2	192.92.105.128	3227281792	27	255.255.255.224
3	192.92.105.160	3227281824	28	255.255.255.240
4	192.92.105.176	3227281840	30	255.255.255.252
5	192.92.105.184	3227281848	29	255.255.255.248
6	192.92.105.192	3227281856	26	255.255.255.192
7	192.92.106.0	3227281920	23	255.255.254.0

Nell'esempio 4 è da notare la posizione 6, che rappresenta un puntatore logico alla posizione 0 (da “network”) di expanded dell'esempio 5.

Il numero di elementi espansi è 8 (“prefix”-220).

La posizione 7 rappresenta un elemento invalidato.

L'array logico addr+expanded (degli esempi 4 e 5) costituisce l'output della funzione.

9.Funzioni base: ottimizzazione

Pseudo codice

Le parti in grassetto-italico non sono espresse in C.

```
1.optimize(entry[] addr,bool do_sort)
2.{
```

```
3.    i=0;
4.    cur=1;
5.
6.    if ( lenght (addr) <= 1)
7.    {
8.          return 0;
9.    }
10.
11.   if (do_sort)
12.   {
13.         sort_entries(addr);
14.   }
15.
16.   while ((addr[0].prefix == 0) && (cur < lenght(addr) ))
17.   {
18.         if (addr[cur].prefix != INVALID_PREFIX )
19.         {
20.               sposto addr[cur] in addr[0];
21.         }
22.         cur++;
23.   }
24.
25.   while (cur < lenght(addr) )
26.   {
27.         if ((addr[cur] != INVALID_NETMASK )&&(addr[cur] non è
  sottorete di addr[i])
28.         {
29.               if (addr[cur] e addr[i] sono aggregabili)
30.               {
31.                     if (i>0)
32.                     {
33.                           addr[cur] = aggregazione( addr[cur],
  addr[i]);
34.                           i--;
35.                     }
36.                     else
37.                     {
38.                           addr[i] = aggregazione(addr[cur] ,
  addr[i]);
39.                           cur++;
40.                     }
41.               }
```

```
42.                 else
43.                 {
44.                       i++;
45.                       addr[i] = addr[cur]
46.                       cur++;
47.                 }
48.           }
49.           else
50.           {
51.                 cur++;
52.           }
53.     }
54.
55.     return i+1;
56.}
```

Precondizioni

La funzione optimize prende in input l'array di tipo entry_list di nome addr.

La funzione optimize necessita dell'ordinamento di addr per il suo corretto funzionamento.

Tale ordinamento deve essere già stato eseguito prima dalla chiamata oppure può essere eseguito dalla funzione stessa, a seconda della valorizzazione del parametro do_sort.

Analisi della funzione: idee base e invariante ciclo principale

La scelta implementativa base della funzione è di eseguire l'aggregazione degli elementi in input senza creare un duplicato dei dati, costoso sia in termini di memoria occupata che di tempo di calcolo.

Gli indirizzi che corrispondono a puntatori logici sono trattati come normali elementi validi che non possono essere ridotti in nessun modo usando le proprietà illustrate in precedenza. Verranno pertanto semplicemente ricopiati nell'array di output.

Siccome si tratta di eseguire confronti fra reti nello stesso array, si sono utilizzati due indici che puntano ad elementi dello stesso:

- l'indice i, che punta all'ultimo elemento già analizzato
- l'indice cur, che punta al primo elemento da analizzare.

I due indici servono a partizionare l'array, definendo così due gruppi di reti:

- insieme **RES**: dall'indirizzo 0 all'indirizzo i (incluso). Sono tutti elementi validi (cioè con prefix != INVALID_PREFIX); costituiscono l'insieme minimo degli

elementi dell'array in input dalla posizione 0 alla posizione cur (addr[cur] escluso)

- insieme **TODO**: il range di elementi che va dall'indirizzo cur a fine array. Sono gli elementi ancora da analizzare.

Per differenza, gli elementi che vanno da i+1 a cur -1 sono posizioni dell'array originario che durante l'esecuzione si liberano poiché la procedura esegue la compattazione dell'array eliminando i buchi logici (posizioni invalide) e applicando le proprietà di riduzione.

L'insieme su cui si sta lavorando sarà quindi costituito dalla concatenazione di **RES** e **TODO**: il “buco” centrale non è da considerarsi né al fine dell'ordinamento né dei dati ivi contenuti.

Viste queste premesse, l'invariante del ciclo principale della funzione (linee 56-93) è così definita:

1. gli elementi di **RES** costituiscono il sottoinsieme minimo dell'insieme iniziale, che non comprende posizioni invalide
2. gli elementi di **RES** più l'elemento addr[cur] comprendono tutte le reti dell'array originale fino alla posizione cur
3. la concatenazione degli elementi di **RES** e **TODO** è ordinata secondo il criterio di ordinamento delle reti sopra descritto.

L'algoritmo si basa anche in questo caso sull'ordinamento, facendo leva sul fatto che, se esiste una sottorete di una rete nell'insieme di input, essa sarà nelle posizioni immediatamente successive.

Inoltre, se due reti sono aggregabili ma non sono sottoreti l'una dell'altra, allora avranno netmask uguale e saranno in posizioni logicamente consecutive (a meno di “buchi” dell'array), in quanto altrimenti sarebbero l'una sottorete dell'altra e si ricadrebbe nel caso precedente.

Analisi della funzione: correttezza

Quando in questo paragrafo si parlerà di ordinamento, si intenderà l'ordinamento degli elementi contenuti nell'insieme valido, cioè la concatenazione di **RES** e **TODO**, che costituiscono l'insieme di reti su cui si sta lavorando.

Gli indici principali i e cur sono inizializzati rispettivamente a 0 e 1 (linee 43 e 44).

Se l'array in input contiene 0 o 1 elementi, la funzione esce immediatamente, in quanto un array vuoto o con un elemento è, per definizione, un insieme minimo (da notare che è anche ordinato).

Alla linea 53, se necessario, viene eseguito il sorting (parametro in input do_sort).

Il ciclo che va dalle linee 56 a 63 serve per assicurare il predicato 1. dell'invariante

principale durante la prima esecuzione dello stesso.

Da notare che questa operazione rispetta l'ordinamento, in quanto i è e rimane a 0 e addr[0] viene riempito con il più piccolo elemento valido dell'array.

I predicati 1., 2. e 3. saranno soddisfatti alla prima esecuzione del ciclo principale

Il numero 1. è soddisfatto in quanto **RES** conterrà un solo elemento (addr[0]), che il ciclo precedente ha assicurato essere valido.

Il 2. è soddisfatto in quanto addr[i] contiene il primo elemento valido, mentre addr[cur] il primo da analizzare successivo a questo (rispetto all'input cambierà eventualmente solo la posizione dell'elemento addr[0]).

Il 3. è soddisfatto perché l'array originale o era ordinato come precondizione o è stato ordinato alla linea 53.

Passiamo ora all'analisi dell'esecuzione del ciclo principale.

La condizione alla linea 67 è doppia e, in entrambi i casi, ha come risultato di togliere dall'insieme **TODO** l'elemento in posizione cur, incrementandone il valore di 1, senza rimpiazzarlo con alcuno altro.

Tale operazione è lecita, in quanto sottoposta a due possibili condizioni:

- l'elemento in posizione cur è invalido.
  Ovvio che tutti i predicati siano ancora verificati in questo caso
- l'elemento in posizione cur è sottorete di quello in posizione i.
  Anche in questo caso, il fatto che l'elemento in posizione cur venga semplicemente scartato non viola l'invariante, in quanto **RES** rimane invariato (predicato 1.). Una sottorete di una rete può essere rimossa dall'insieme senza toglierne effettivamente elementi per la prima proprietà di riduzione (predicato 2.).
  Inoltre, togliere un elemento dell'array sicuramente non viola l'ordinamento (predicato 3.).

Passiamo ora all'analisi della condizione di aggregabilità di due reti (seconda proprietà di riduzione).

Due reti sono aggregabili se possono essere sostituite da una terza rete che contiene tutti e soli gli hosts di entrambe. Esempio: 10.10.10.0/24 e 10.10.11.0/24 sono aggregabili in 10.10.10.0/23.

Il fatto che due reti siano consecutive dal punto di vista degli hosts e abbiano la stessa netmask non è una condizione sufficiente. Ad esempio 10.10.10.1/32 e 10.10.10.2/32 non sono aggregabili.

La verifica dell'aggregabilità si trova alla linea 69.

Come detto sopra, se due reti sono aggregabili allora hanno sicuramente netmask uguale e sono in posizioni consecutive nel nostro array “logico”.

In particolare, ad un certo punto verranno a trovarsi una in addr[i] e l'altra in

addr[cur].

La rete aggregata avrà un bit di rete in meno e risulterà più piccola (sempre secondo l'ordinamento stabilito) delle reti da cui deriva, siccome le stesse sono sottoreti dell'aggregata.

Le azioni intraprese differiscono leggermente se l'indice i è uguale a 0 (e quindi l'insieme **RES** contiene solo addr[0]) oppure se no lo è.

- i > 0 (linee 73-74): in questo caso viene sovrascritta la posizione cur con l'aggregato delle due reti e viene decrementato l'indice i.
  Questa operazione ha come effetto di riconsiderare l'aggregato delle due reti (in quanto viene messo in una delle due posizioni soggette ad analisi) e di rimuovere da **RES** l'ultimo elemento, in quanto già compreso in addr[cur].
  Inoltre il decremento di i farà sì che venga riconsiderato il penultimo elemento di **RES** per verificare eventuali possibilità di aggregazione con addr[cur] (il nuovo elemento inserito) per garantire che l'insieme **RES** finale sia quello minimo.
  I predicati dell'invariante non vengono violati: il numero 1. è rispettato in quanto togliere un elemento a **RES** sicuramente non lo viola.
  Il numero 2. è altresì rispettato in quanto la rete aggregata è equivalente alle due che vengono di fatto soppresse e viene messa in addr[cur].
  Il numero 3. non viene violato in quanto, per la caratteristica dell'ordinamento, la rete inserita sarà minore delle due reti originarie, ma maggiore di tutte le precedenti (cioè di tutte le reti presenti nell'insieme **RES** privato dell'ultimo elemento).
- i = 0 (linee 78-79): in questo caso viene sovrascritta la posizione i con l'aggregato delle due reti e viene incrementato l'indice cur.
  La differenza rispetto al caso precedente é che, siccome l'insieme **RES** contiene solo l'elemento addr[0], non è ovviamente possibile che vi risiedano reti in posizione i-1 aggregabili con la nuova aggiunta.
  Viene quindi semplicemente sovrascritta la posizione 0 e scartata la posizione cur incrementando il contatore.
  I predicati dell'invariante continuano ad essere validi. Il numero 1,. in quanto **RES,** poiché contiene un solo elemento valido, è sicuramente un insieme ottimale.
  Il numero 2. in quanto la rete inserita in posizione 0 è equivalente a quelle in precedenza inserite in posizione i e cur. Il numero 3. in quanto la nuova rete inserita è sicuramente minore delle due precedenti e quindi, a maggior ragione, l'ordinamento è preservato.

Resta da analizzare il caso di “default”, cioè quando addr[cur] è valido, addr[cur] non è sottorete di addr[i] e addr[cur] non è aggregabile con addr[i]. In questo caso verrà incrementato i, copiato il contenuto di addr[cur] in addr[i] e verrà incrementato cur.

Da notare che se cur=i+1, questa operazione è inutile ma è si è preferito evitare il controllo ad ogni ciclo, siccome su input “normali” si tratta di un caso che se si verifica solo nei primi cicli di esecuzione del programma.

Queste operazioni soddisfano l'invariante perché, non essendo l'elemento addr[cur]

aggregabile in alcun modo con addr[i], la sua aggiunta a **RES** non viola il predicato 1. Il predicato 2. è altresì soddisfatto in quanto l'elemento in posizione cur è stato aggiunto all'insieme RES e quindi sicuramente è preservato.Il predicato 3. è pure soddisfatto in quanto l'elemento spostato viene inserito in coda a **RES** e cur viene incrementato.

La condizione di terminazione è banale: quando non ci sono più elementi nell'insieme **TODO**, il ciclo termina.

La funzione optimize, infine, ritorna alla riga 95 il numero di posizioni valide dell'array, cioè il numero di elementi che costituiscono l'insieme **RES**.

Postcondizioni

L'invariante del ciclo principale ci porta a dimostrare che all'uscita della funzione l'array addr ha le seguenti caratteristiche:

- dai predicati 1. e 2. si deduce che l'insieme **RES** di uscita conterrà l'insieme minimo degli elementi in input (considerando addr[cur] posizione non più da considerare all'uscita del ciclo). Non conterrà nessuna posizione invalida.
- dal predicato 3. si deduce che l'insieme **RES** sarà completamente ordinato (essendo l'insieme **TODO** vuoto all'uscita del ciclo).

Da notare che l'output della funzione consta non solo degli elementi dell'array addr, ma anche di tutti gli elementi dell'array expanded generati da apply_whitelist e non considerati da optimize in quanto già sottoinsieme minimo.

Esempi

Esempio 6: array addr dopo l'esecuzione della funzione di ottimizzazione, usando gli elementi dell'esempio 4 come input

Pos. Array	Network (dotted)	Network	Prefix	Netmask
0	1.2.0.0	16908288	16	255.255.0.0
1	10.20.20.0	169088000	23	255.255.254.0
2	20.0.0.0	335544320	8	255.0.0.0
3	N/A	0	228	N/A
4	202.2.4.16	3389129744	28	255.255.255.240

Il puntatore logico in posizione 3 punta agli elementi dell'esempio 5. La funzione optimize non considera tali elementi, essendo già stati ridotti dalla funzione apply_whitelist.

10.Complessità

La complessità di un programma deriva dalla funzione che ha l'**O** (“o grande”) maggiore fra tutte quelle che compongono il programma stesso.

Notazione: per N si intende il numero di elementi di I, per Z la somma degli elementi di W, per logo si intende il logaritmo in base 2, per M si intende il numero di reti aggiunte ad I a causa dell'espansione dovuta a conflitto, con T il numero delle reti aggregate e con P la percentuale di riduzione.

Riportiamo di seguito l'analisi delle funzioni base del programma.

Get_entries

La funzione legge gli input una linea alla volta e ne parsifica il contenuto per estrarre gli indirizzi in notazione con prefisso.

Se consideriamo l'operazione di parsing non dipendente dall'input (cosa che nel nostro caso è ragionevole), si può considerare il tempo necessario allo stesso costante O(1) e, quindi, far dipendere il tempo di esecuzione della funzione dal numero degli elementi da leggere.

La funzione avrà quindi complessità **O(N+Z)**.

Sort_entries

L'algoritmo si avvale dell'algoritmo quicksort. Lo studio di complessità di tale algoritmo si può trovare in letteratura.

La sua complessità nel caso medio, essendo I e W gli insiemi da ordinare separatamente, è **O(N * log (N)) + O(Z * log (Z))**.

Apply_whitelist

Si fa riferimento in questo paragrafo al metacodice presente nella descrizione dettagliata della funzione.

La funzione si avvale di due indici, i1 e i2, per avanzare rispettivamente nell'array della blacklist (addr) o della whitelist (white).

La funzione esegue i test di terminazione del ciclo principale e i check per la verifica dei conflitti in tempo O1.

La condizione di terminazione fa sì che, se si sono terminati gli elementi di W o di I, la funzione esce. Il caso migliore avrà quindi complessità min(O(W),O(N)).

Nei casi di non conflitto viene incrementato di una unità l'indice i1 o l'indice i2 Questo caso contribuisce quindi con complessità lineare rispetto alla somma degli elementi di I e W.

Una riflessione particolare merita l'espansione di una rete. Essa, come sappiamo, introduce X elementi nuovi nell'array expanded, con X che va da 1 a 32. Gli indici

vengono incrementati fino a passare oltre agli elementi espansi e a quelli invalidati.

Ai fini della complessità bisogna quindi considerare gli elementi espansi come semplice aggiunta lineare agli elementi in input.

Nel suo insieme la complessità media della funzione può essere quindi approssimata come **O(N+Z+M)**.

Optimize

La funzione utilizza due indici: i1 che rappresenta l'ultimo elemento di RES valido e il secondo cur che rappresenta il primo elemento di TODO.

La funzione viene richiamata sul solo array addr risultato dell'apply_whitelist. Quindi, rispetto all'insieme iniziale, non bisogna considerare gli elementi espansi in quanto optimize non li considera.

La funzione è composta di due cicli: il primo cerca la prima posizione valida nell'array incrementando l'indice cur. Offre quindi un contributo lineare alla complessità.

Il secondo ciclo (il principale) esegue i test di terminazione e i controlli per la verifica delle due proprietà di riduzione in tempo costante O(1).

Nel caso “normale” esegue la copia dell'elemento in posizione cur in posizione i (O (1)) ed incrementa entrambi i contatori. Questo caso ha quindi complessità O(N).

In caso l'elemento addr[cur] sia una sottorete, la posizione cur verrà invalidata e l'indice incrementato. Anche questo ramo da un contributo lineare.

L'ultimo caso, quello in cui due reti sono aggregabili, da origine ad un nuovo elemento da analizzare (messo in posizione cur) e ad un decremento dell'indice i. In questi casi, di numero T, l'aggregazione di per sé può essere considerata constante, ma si tornerà ad analizzare un elemento già analizzato in precedenza. Avviene quindi un incremento lineare della complessità totale della funzione.

Nel suo insieme la funzione ha perciò complessità **O(N+T).**

Print_addresses

La funzione si occupa di stampare tutti gli elementi dell'array ridotto.

Considerando costante il tempo necessario alla stampa del singolo elemento, è chiaro che il numero di elementi da stampare dipenderà dal numero delle reti rimaste dopo l'applicazione della whitelist e dopo la riduzione.

In questo caso verranno ovviamente considerati anche tutti gli elementi in expanded, in quanto facenti parte dell'output finale del programma.

Si può prendere in considerazione il fattore di riduzione esprimendolo come percentuale P. La complessità di questa funzione può essere espressa come **O((N+M) * P)**, P<=1.

Visione Complessiva

Riportiamo qui l'elenco delle funzioni con associata la complessità:

Funzione	Caso Migliore	Caso Peggiore	Caso Medio
get_enties	O(N+Z)	O(N+Z)	O(N+Z)
sort_entries	O(N * log (N)) + O(Z * log (Z))	O(N ^ 2) + O(Z ^ 2)	O(N * log (N)) + O(Z * log (Z))
apply_whitelist	min(O(Z),O(N))	O(N+Z) + O(M)	O(N+Z) + O(M)
optimize	O(N)	O(N+T)	O(N+T)
print_addresses	O((N+M) * P)	O((N+M) * P)	O((N+M) * P)

Dalla tabella si può notare come tutte le funzioni, a parte il sorting, hanno un tempo di esecuzione lineare, compresa la print_address, siccome P non dipende dalla numero degli elementi in input, bensì dalla loro “combinazione”.

E' importante anche rilevare che la funzione apply_whitelist, a seconda degli input, può avere un M significativamente grande, anche maggiore di N + Z.

Si può comunque dire che il programma nel suo insieme ha complessità asintotica

O(N * log (N)) +O(Z * log (Z)).

11.Il programma nel suo complesso

Scelte implementative

L'implementazione è stata fatta in rigoroso ANSI C per permettere la portabilità su diversi compilatori/sistemi (anche se, fino ad oggi, è stato testato con gcc e tcc).

I prototipi esportati nella libreria e i valori configurabili sono tutti definiti nel file “cidrmerge.h”.

E' fornito un Makefile contenente al suo interno una serie di definizioni che permettono di controllare alcune funzionalità del programma e di attivare il debugging.

A livello di formato dei dati sono stati utilizzati array allocati a bucket di dimensione configurabile e allargati/ristretti all'occorrenza con la primitiva C di “realloc”.

Compilazione

Se non è necessario eseguire il debug del programma, assumendo di avere a disposizione il gcc e lo gnu make installati e funzionanti sulla macchina (e nel path di esecuzione), per la sua compilazione sarà sufficiente eseguire nella directory con i sorgenti e il makefile il comando:

```
make
```

ed ottenere l'eseguibile “cidrmerge”.Per avere anche la libreria compilata , basta eseguire il comando:

```
make lib
```

Interfaccia del programma

Il programma non ha interfaccia utente e si configura come una utility command line da lanciare secondo necessità o da eseguire in maniera schedulata, eventualmente guidata da script o da altri programmi di controllo.

La sintassi del comando è la seguente:

```
./cidrmerge -h
Usage: cidrmerge [whitelist file] [NOOPTIMIZE]
```

L'input viene preso in parte su standard input (la blacklist) e la whitelist (opzionale) viene caricata dal file specificato come parametro.

Per avere un output non ottimizzato, ma solo ordinato ed eventualmente privato degli elementi in whitelist, è possibile specificare il flag NOOPTIMIZE.

L'output verrà fornito su standard output, gli errori su standard error.

Formati in input

Il programma supporta diversi tipi di formati in input, per essere in grado di importare i diversi file di blacklist diffusi in Internet e che, per vari motivi, esprimono lo stesso concetto base, ma con un formalismo diverso.

Di seguito l'elenco dei formati supportati dal programma, per importare, ad esempio, la rete 10.20.0.0/16. Le reti sono specificate una per ogni riga e sono seguite da un ritorno a capo:

- CIDR in formato prefisso: `10.20.0.0/16`
- hosts mancanti: `10.20`
- hosts indicati con *: `10.20.* o 10.20*`
- qualunque dei formati precedenti con alla fine un carattere “spazio” e testo libero:
  `10.20 testo che verrà ignorato`

Utilizzo

E' chiaro che con questa impostazione il programma si configura come una utility perfettamente integrabile con script di varia natura su sistemi unix-like.

Esso può essere utilizzato per il suo scopo originale, cioè per la fusione di più blacklist con l'applicazione di una whitelist, ma anche per utilizzi che esulano leggermente, quali:

- compattazione di tabelle di reti/ACL
- esclusione di uno o più hosts da una tabella di reti

- calcolo del complemento di un insieme di reti: se si da in input al programma come “blacklist” 0.0.0.0/0, rete che comprende tutti gli host indirizzabili da IPv4, si può ottenere in output il complemento minimo delle reti presenti nella whitelist.

Vengono inoltre fornite a corredo due utility per per convertire reti dal formato con prefisso a quello con netmask e viceversa (netmasktocidr.pl e cidrtonetmask.pl).

Libreria

Il programma esporta anche una libreria dinamica che contiene le funzioni base (apply_whitelist, optimize e sort_entries).

Tale libreria può essere usata nel caso si vogliano impiegare le funzioni in essa contenute in programmi esterni, che abbiano già un'interfaccia grafica e funzioni di input/output implementate.

Open Source

Il programma è stato rilasciato sotto la licenza GPL e reso disponibile sotto forma di codice sorgente su Internet sul sito: http://sourceforge.net/projects/cidrmerge/

Tale scelta è stata presa per condividere facilmente il software con chi ne può avere la necessità e per velocizzare il bug fixing ricorrendo all'aiuto degli utilizzatori.

A ogni nuovo rilascio chi si è iscritto alla mailing list riceve una email che gli permette di verificare l'opportunità dell'installazione.

Il programma è al momento fornito solo in formato codice sorgente, ma in futuro potrà essere rilasciato anche nei formati binari più diffusi (linux x86 e solaris sparc).

12.Profiling & Testing

Profiling

Il programma è in grado di tracciare nel file `timestamp.out` i tempi di esecuzione delle varie funzioni interne. Ciò è stato utile nell'eseguire un profiling delle funzioni stesse e nella loro ottimizzazione volta per volta per ottenere un tempo di esecuzione finale il più veloce possibile.

Di seguito è possibile vedere come si presenta il file di profiling eseguendo il programma su un intel centrino 1.4Gz e 512Mb di ram su un input preso da un sistema reale con circa 8.5 milioni di reti:

```
3 get_entries
2 sort_entries
0 apply_whitelist
0 optimize
1 print_addresses
blacklist: in input 8631142, expanded 10 final optimize
6539609
whitelist: total 305
```

```
Total execution time 6 seconds.
```

Siccome la granularità dei tempi è in secondi, per apprezzare meglio la distribuzione fra le varie funzioni la CPU è stata rallentata dell'87% (il minimo possibile) e il programma è stato rieseguito:

```
36 get_entries
18 sort_entries
2 apply_whitelist
2 optimize
6 print_addresses
blacklist: in input 8631142, expanded 10 final optimize
6539609
whitelist: total 305
Total execution time 64 seconds.
```

Da questo tracciato di esecuzione si nota ancora meglio come le funzioni core del programma (apply_whitelist e optimize) richiedano pochissimo tempo: solo 4 secondi su un tempo totale di 64. Le funzioni di input/output occupano invece la maggior parte del tempo di esecuzione: il 65% circa. La funzione di sorting (versione ottimizzata), nonostante sia quella con la complessità asintotica maggiore di tutte, non richiede moltissimo tempo: solo il 28% circa del totale.

Se però ripetiamo l'esecuzione con la funzione di sorting standard della libreria C presente sul PC su cui sono stati eseguiti i test, il risultato cambia drasticamente:

```
36 get_entries
87 sort_entries
2 apply_whitelist
3 optimize
6 print_addresses
blacklist: in input 8631142, expanded 10 final optimize
6539609
whitelist: total 305
Total execution time 134 seconds.
```

La durata della funzione di sorting si è incrementa quasi di cinque volte, da 18 a 87 secondi.

In questo caso essa richiede circa il 65% del tempo di esecuzione totale del programma sugli stessi input.

E' lampante il vantaggio della funzione ottimizzata rispetto a quella “originale”. Il link al sito dell'autore si trova nella sezione “Bibliografia”.

Probabilmente c'è ancora margine di miglioramento lavorando sulle funzioni di input/output (specialmente la get_entries), anche se molto lavoro è già stato fatto e il parsing dell'input necessita di tempo di calcolo per l'esecuzione del parsing e dei controlli formali sull'input.

Utilizzo di memoria

L'elemento base degli array è di tipo “struct entry” (struct composta da un intero ed un

char); non è una struttura con i campi allineati alla parola/macchina (in questo caso a 32 bit) e perciò a seconda delle architetture e del tipo di compilazione si comporta in modo diverso.

Il programma quindi, in base ai dati utilizzati per i test, eseguito su piattaforma 32 bit con le strutture allineate a 32 bit (con la conseguente perdita di 3 bytes per ogni posizione degli array), occupa circa 80MB di RAM.

Sulle piattaforme che supportano l'indirizzamento di interi anche non allineati alla parola macchina (tipo quelle con CPU Intel), si è potuto testare il programma utilizzando le strutture “compattate” in maniera da consumare il solo spazio utilizzato. In questo caso la memoria utilizzata scende a circa 50MB.

In caso di espansione di molte reti, la memoria occupata tende a crescere, ma, in generale, il programma ha mostrato di scalare in maniera adeguata ai tagli di memoria delle macchine attuali.

Testing

Il testing è stato eseguito inizialmente su insiemi con input limitati e mirati per poi passare all'utilizzo di dati reali.

Il programma è attualmente disponibile per il download via Internet ed esiste un beta tester “ufficiale” che ha provato tutte le release fino ad ora rilasciate su dati di produzione, facendo il confronto con le versioni precedenti per appurarne la non regressione e l'effettiva conformità di tutte le nuove funzionalità.

Le piattaforme su cui è stato testato ed utilizzato sono:

- Solaris SPARC
- Linux x86
- RISC IBM

Non ci sono comunque ostacoli teorici al fatto che il programma possa girare anche su altre piattaforme.Il programma, sui dati usati per i test, eseguito su piattaforma 32 bit con le strutture allineate a 32 bit (con la conseguente perdita di 3 bytes per ogni posizione degli array), occupa circa 80MB di RAM. Per le caratteristiche di implementazione, non dovrebbero esserci grossi ostacoli all'uso di altri (vedi “Scelte implementative”, cap.11).

13.Sviluppi futuri

Gli sviluppi futuri del programma potranno prendere diverse strade, a seconda delle esigenze degli utilizzatori e di eventuali integrazioni con software complementari.

Soprattutto nell'ottica di migliorare la funzione apply_whitelist si potrebbe pensare di utilizzare un diverso formato dei dati, al fine di evitare l'utilizzo di un array di appoggio ed eseguire la funzione sulla stessa struttura.

Interessanti da questo punto di vista sono i btrie (normali e quelli di tipo

“PATRICIA”), ma l'impatto generale sul programma deve essere ancora valutato.

I tempi di esecuzione e l'utilizzo di memoria della versione attuale si sono comunque rivelati adeguati per tutti gli utilizzi di produzione, anche su grosse installazioni, per cui non c'è una richiesta immediata di ulteriori miglioramenti.

Un interessante sviluppo ai fini dell'utilizzo dell'algoritmo per le blacklist è la possibilità di considerare dei record descrittivi (record di tipo TXT) che vengono utili per riferimento.

14.Ringraziamenti

Si ringraziano Cataldo Cigliola e Mario Caruso per il testing di tutte le versioni del programma in un ambiente di produzione.

Si ringrazia inoltre Michael Tokarev per aver reso disponibile a tutti la funzione di sorting da lui ottimizzata che è stata integrata nel programma e ha portato grossi benefici di performance.

15.Bibliografia

- RFC 1518, ftp://ftp.rfc-editor.org/in-notes/rfc1518.txt
- RFC 1771, ftp://ftp.rfc-editor.org/in-notes/rfc1771.txt
- QSORT, http://www.corpit.ru/mjt/qsort.html

16.Esempio di esecuzione

Lo scopo dei seguenti esempi è di mostrare passo per passo come il programma agisce sull'input, con il fine di capire meglio l'algoritmo.

Verranno mostrati gli indici correnti e i dati negli array all'inizio della ciclo; pertanto l'effetto delle operazioni eseguite ad ogni step sarà visibile nella tabella dello step successivo.

Esecuzione di apply_whitelist

Si utilizzeranno i dati in input degli esempi 2 e 3.

Ogni tabella rappresenta la stampa degli array passo passo, con evidenziato l'elemento di posizione corrente e con il commento dei punti salienti.

L'espansione verrà commentata passo passo nel prossimo paragrafo.

addr	white

STEP 0

i1	network	prefix	i2	network	prefix
0	1.2.0.0	16	0	25.30.40.0	24
1	1.2.3.4	32	1	192.92.105.180	30
2	10.20.20.0	25			
3	10.20.20.128	25			
4	10.20.21.0	24			
5	20.0.0.0	8			
6	192.92.104.0	22			
7	192.92.105.4	30			
8	202.2.4.16	28			

STEP 1

i1	network	prefix	i2	network	prefix
0	1.2.0.0	16	0	25.30.40.0	24
1	1.2.3.4	32	1	192.92.105.180	30
2	10.20.20.0	25			
3	10.20.20.128	25			
4	10.20.21.0	24			
5	20.0.0.0	8			
6	192.92.104.0	22			
7	192.92.105.4	30			

i1	network	prefix		i2	network	prefix
8	202.2.4.16	28				

STEP 2

i1	network	prefix		i2	network	prefix
0	1.2.0.0	16		0	25.30.40.0	24
1	1.2.3.4	32		1	192.92.105.180	30
2	10.20.20.0	25				
3	10.20.20.128	25				
4	10.20.21.0	24				
5	20.0.0.0	8				
6	192.92.104.0	22				
7	192.92.105.4	30				
8	202.2.4.16	28				

STEP 3

i1	network	prefix		i2	network	prefix
0	1.2.0.0	16		0	25.30.40.0	24
1	1.2.3.4	32		1	192.92.105.180	30
2	10.20.20.0	25				
3	10.20.20.128	25				
4	10.20.21.0	24				
5	20.0.0.0	8				
6	192.92.104.0	22				
7	192.92.105.4	30				
8	202.2.4.16	28				

STEP 4

i1	network	prefix		i2	network	prefix
0	1.2.0.0	16		0	25.30.40.0	24
1	1.2.3.4	32		1	192.92.105.180	30
2	10.20.20.0	25				
3	10.20.20.128	25				

i1	network	prefix		i2	network	prefix
4	10.20.21.0	24				
5	20.0.0.0	8				
6	192.92.104.0	22				
7	192.92.105.4	30				
8	202.2.4.16	28				

STEP 5

i1	network	prefix		i2	network	prefix
0	1.2.0.0	16		0	25.30.40.0	24
1	1.2.3.4	32		1	192.92.105.180	30
2	10.20.20.0	25				
3	10.20.20.128	25				
4	10.20.21.0	24				
5	20.0.0.0	8				
6	192.92.104.0	22				
7	192.92.105.4	30				
8	202.2.4.16	28				

STEP 6

i1	network	prefix		i2	network	prefix
0	1.2.0.0	16		0	25.30.40.0	24
1	1.2.3.4	32		1	192.92.105.180	30
2	10.20.20.0	25				
3	10.20.20.128	25				
4	10.20.21.0	24				
5	20.0.0.0	8				
6	192.92.104.0	22				
7	192.92.105.4	30				
8	202.2.4.16	28				

STEP 7

La posizione addr[6] è in conflitto con la posizione white[1] e white[1] è sottorete propria di addr[6]; si procede quindi all'espansione dell'array. Si invalida inoltre addr[7] in quanto sottorete di addr[6]. I1 viene quindi incrementato di 2 (per andare oltre all'elemento espanso e all'elemento invalidato) e i2 viene incrementato di 1.

Di conseguenza alla fine di questo step i1=8 e i2=2.

i1	network	prefix		i2	network	prefix
0	1.2.0.0	16		0	25.30.40.0	24
1	1.2.3.4	32		1	192.92.105.180	30
2	10.20.20.0	25				
3	10.20.20.128	25				
4	10.20.21.0	24				
5	20.0.0.0	8				
6	192.92.104.0	22				
7	192.92.105.4	30				
8	202.2.4.16	28				

STEP 8

i2=2 e quindi la funzione esce dal ciclo principale.

i1	network	prefix		i2	network	prefix
0	1.2.0.0	16		0	25.30.40.0	24
1	1.2.3.4	32		1	192.92.105.180	30
2	10.20.20.0	25				
3	10.20.20.128	25				
4	10.20.21.0	24				
5	20.0.0.0	8				
6	0	228				
7	192.92.105.4	33				
8	202.2.4.16	28				

L'array addr dello step 8 costituisce l'array addr in output della funzione apply_whitelist.

Esecuzione di expand

Di seguito i passi dell'espansione di addr[6] (preso dall'array in input di

apply_whitelist) rispetto all'elemento in conflitto nella whitelist white[1].

La routine esegue l'espansione di 8 elementi, siccome white[1]-addr[6]=30-22=8. La funzione eseguirà sempre un numero di step uguale al numero di elementi da espandere.

La variabile up (indicata in giallo) viene inizializzata a 0, la variabile bottom (indicata in blu) viene inizializzata a 7. Nel caso assumano lo stesso valore, nella tabella sottostante verrà utilizzato il colore rosso.

STEP 0

	network	**Prefix**
0		
1		
2		
3		
4		
5		
6		
7		

STEP 1

	network	**Prefix**
0		
1		
2		
3		
4		
5		
6		
7	192.92.106.0	23

STEP 2

	network	**Prefix**
0	192.92.104.0	24

	network	Prefix
1		
2		
3		
4		
5		
6		
7	192.92.106.0	23

STEP 3

	network	Prefix
0	192.92.104.0	24
1	192.92.105.0	25
2		
3		
4		
5		
6		
7	192.92.106.0	23

STEP 4

	network	Prefix
0	192.92.104.0	24
1	192.92.105.0	25
2		
3		
4		
5		
6	192.92.105.192	26
7	192.92.106.0	23

STEP 5

	network	Prefix
0	192.92.104.0	24
1	192.92.105.0	25
2	192.92.105.128	27
3		
4		
5		
6	192.92.105.192	26
7	192.92.106.0	23

STEP 6

	network	Prefix
0	192.92.104.0	24
1	192.92.105.0	25
2	192.92.105.128	27
3	192.92.105.160	28
4		
5		
6	192.92.105.192	26
7	192.92.106.0	23

STEP 7

	network	Prefix
0	192.92.104.0	24
1	192.92.105.0	25
2	192.92.105.128	27
3	192.92.105.160	28
4		
5	192.92.105.184	29
6	192.92.105.192	26
7	192.92.106.0	23

STEP 8

	network	Prefix
0	192.92.104.0	24
1	192.92.105.0	25
2	192.92.105.128	27
3	192.92.105.160	28
4	192.92.105.176	30
5	192.92.105.184	29
6	192.92.105.192	26
7	192.92.106.0	23

La funzione esce dal ciclo e termina dopo lo step 8 in quanto up > bottom.

Come si può notare l'array è ordinato e non è riducibile.

Esecuzione di optimize

La funzione optimize viene eseguita sull'array addr di output della funzione apply_whitelist.

L'elemento che si trova all'indice i viene evidenziato in giallo, mentre quello all'indice cur viene evidenziato in blu.

Di conseguenza l'insieme degli elementi da inizio array fino alla posizione gialla costituisce l'insieme **RES,** mentre l'insieme che va dalla posizione rossa fino a fine array è il **TODO**.

Man mano che cur si distanzia da i, gli elementi che sono in mezzo vengono cancellati dalla tabella (anche se il programma non li invalida) per evidenziare che le posizioni sono libere dal punto di vista logico.

Gli elementi stampati ad ogni passo corrispondono a quelli presenti nell'array all'inizio del ciclo. I commenti sono relativi all'esecuzione dello stesso: di conseguenza il risultato dell'esecuzione si vedrà in tabella nello step seguente.

STEP 0

L'elemento addr[1] è compreso in addr[0]. Verrà quindi semplicemente incrementato il contatore cur senza modificare addr[0].

	network	prefix
0	1.2.0.0	16
1	1.2.3.4	32
2	10.20.20.0	25
3	10.20.20.128	25

	network	prefix
4	10.20.21.0	24
5	20.0.0.0	8
6	0	228
7	192.92.105.4	33
8	202.2.4.16	28

STEP 1

	network	prefix
0	1.2.0.0	16
1		
2	10.20.20.0	25
3	10.20.20.128	25
4	10.20.21.0	24
5	20.0.0.0	8
6	0	228
7	192.92.105.4	33
8	202.2.4.16	28

STEP 2

addr[1] e addr[3] sono aggregabili. Siccome i > 0, l'indirizzo aggregato viene messo in posizione addr[3] e viene decrementato di 1 i.

	network	prefix
0	1.2.0.0	16
1	10.20.20.0	25
2		
3	10.20.20.128	25
4	10.20.21.0	24
5	20.0.0.0	8
6	0	228
7	192.92.105.4	33
8	202.2.4.16	28

STEP 3

	network	prefix
0	1.2.0.0	16
1		
2		
3	10.20.20.0	24
4	10.20.21.0	24
5	20.0.0.0	8
6	0	228
7	192.92.105.4	33
8	202.2.4.16	28

STEP 4

addr[1] e addr[4] sono aggregabili. Siccome i > 0, l'indirizzo aggregato viene messo in posizione addr[4] e viene decrementato di 1 i.

	network	prefix
0	1.2.0.0	16
1	10.20.20.0	24
2		
3		
4	10.20.21.0	24
5	20.0.0.0	8
6	0	228
7	192.92.105.4	33
8	202.2.4.16	28

STEP 5

	network	prefix
0	1.2.0.0	16
1		

	network	prefix
2		
3		
4	10.20.20.0	23
5	20.0.0.0	8
6	0	228
7	192.92.105.4	33
8	202.2.4.16	28

STEP 6

	network	prefix
0	1.2.0.0	16
1	10.20.20.0	23
2		
3		
4		
5	20.0.0.0	8
6	0	228
7	192.92.105.4	33
8	202.2.4.16	28

STEP 7

L'elemento addr[6] è un puntatore logico. Verrà pertanto semplicemente trattato come una posizione valida non in conflitto con addr[2].

	network	prefix
0	1.2.0.0	16
1	10.20.20.0	23
2	20.0.0.0	8
3		
4		
5		
6	0	228

	network	prefix
7	192.92.105.4	33
8	202.2.4.16	28

STEP 8

addr[7] è invalidato, viene semplicemente incrementato cur.

	network	prefix
0	1.2.0.0	16
1	10.20.20.0	23
2	20.0.0.0	8
3	0	228
4		
5		
6		
7	192.92.105.4	33
8	202.2.4.16	28

STEP 9

	network	prefix
0	1.2.0.0	16
1	10.20.20.0	23
2	20.0.0.0	8
3	0	228
4		
5		
6		
7		
8	202.2.4.16	28

STEP 10

cur ha superato la lunghezza dell'array, la funzione esce dal ciclo principale. Sotto l'addr finale in output. Le posizione valide sono le prime 5 (da 0 a 4).

	network	prefix
0	1.2.0.0	16
1	10.20.20.0	23
2	20.0.0.0	8
3	0	228
4	202.2.4.16	28
5		
6		
7		
8		

L'output finale del programma nel suo complesso è dato dall'array dello step 10 di optimize più l'array di output della funzione expand.

17.Codice sorgente completo

cidrrmerge.h

```
1  #ifndef CIDRMERGE
2  #define CIDRMERGE
3
4  #define MAXLINE 1024
5  #define BUFFER 8192
6  /* Allocate BUCKET array entries per time */
7  #define BUCKET 100000
8  #define MAXCIDR 23
9
10/*Special prefix values */
11#define INVALID_PREFIX 33
12#define EXPANDED_PREFIX 220
13
14#define MASK1 0xff000000
```

```
15#define MASK2 0xff0000
16#define MASK3 0xff00
17#define MASK4 0xff
18
19#ifndef MAXINT
20#define MAXINT 0xffffffff
21#endif
22
23
24struct entry
25{
26      unsigned int network;
27      #ifdef CHAR_PREFIX
28      unsigned char prefix;
29      #else
30      unsigned int prefix;
31      #endif
32}entry_t;
33
34#define TONETMASK(PREFIX) (((PREFIX)==0)?(0):(MAXINT<<(32-(PREFIX))))
35
36/*
37      Function execute the reduction of entry array addr.It returns the number of entries in the final array.
38      PARAM: addr array: contains entries to be optimized
39      PARAM: len: number of entries into addr array.
40      PARAM: do_sort: if 0, sort will NOT be perfomed. Be carefull: optimizing a non-sorted input will produce unpredictable results
41      NOTE3: resulting array may contains entries with prefix bigger than EXPANDED_PREFIX. Those are logical pointers: expanded_list[entry.network] (array returned by apply_whitelist function) is the starting position and the leght is entry.prefix- EXPANDED_PREFIX
```

```
42*/
43unsigned int optimize(struct entry *addr,unsigned int len,int do_sort);
44
45/*
46      Remove from array *entry_list occurence of array white.
47      Returns number of element into output array expanded_list (that can be
  reassigned to allow array extension).
48      Function allocate more memory if needed. It does NOT deallocate any if the
  result is smaller.
49      PARAM: *entry_list: pointer to input/output param entry list.
50      PARAM: **expanded_list: pointer to output param entry list, used to store
  expanded networks.
51      PARAM: *white: white list entry array
52      PARAM: len1: number of entry into entry_list in input
53      PARAM: len2: number of entry into white_list in input
54      PARAM: size_expanded: return paramether that contains memory size of the
  output expanded_list array
55      PARAM: do_sort: if 0, sort will NOT be perfomed on both entry_list and
  white. Be carefull: optimizing a non-sorted input will produce unpredictable
  results
56      NOTE1: both arrays entry_list and white have to be sorted with sort_entries()
  function BEFORE calling this.function OR paramether do_sort needs to be set to
  non zero. Unpredictable results if not
57      NOTE2: resulting array may contains entries with INVALID_PREFIX value
  prefix. This means entry is invalid and it doesn't have to be taken into account.
58      NOTE3: resulting array may contains entries with prefix bigger than
  EXPANDED_PREFIX. Those are logical pointers: expanded_list[entry.network]
  is the starting position and the leght is entry.prefix- EXPANDED_PREFIX
59      NOTE4: resulning array is ordered apart from invalid entries (the ones with
  INVALID_PREFIX value in prefix)
60*/
61unsigned int apply_whitelist(struct entry *addr,struct entry **expanded_list,struct
  entry *white,unsigned int len1,unsigned int len2,unsigned int *size_expanded,int
  do_sort);
62
```

```
63/*
64      Execute entries sorting using quicksort.
65      PARAM: addr array: contains entries to be sorted
66      PARAM: len: number of entries into addr array.
67*/
68void sort_entries(struct entry *addr,unsigned int len);
69
70#endif
```

cidrmerge-lib.c

```
71#include <stdio.h>
72#include <stdlib.h>
73#include <string.h>
74
75#include "cidrmerge.h"
76
77#ifdef LIBRARY_DEBUG
78/* Using those functions from cidrmerge.c for
  debugging purposes only */
79void print_address(FILE *file,unsigned int net,
  unsigned char pref);
80void print_addresses(FILE *f,struct entry *addr,int
  size,struct entry *expanded,int level);
81#endif
82/*
83   Exectute one entry comparation. a1 is bigger than a2
  if:
84   - a1.network is bigger than a2.network
85   - a1.network is equal to a2.network but a1.prefix is
  bigger than a2.prefix
86   PARAM: a1 first entry
87   PARAM: a2 second entry
88   RETURN VALUE: it returns -1 if a1<a2, 0 if a1=a2 and
  1 if a1>a2
89*/
90#ifndef OPTIMIZED_SORT
```

```
91typedef int(*cmp_type)(const void *,const void*);
92
93static int cmp_entry(struct entry *a1, struct entry
  *a2)
94{
95   if ((*a1).network<(*a2).network)
96   {
97        return -1;
98   }
99   else if ((*a1).network>(*a2).network)
100  {
101       return 1;
102  }
103/* Network are ==, so we compare netmasks*/
104  else if ((*a1).prefix<(*a2).prefix)
105  {
106       return -1;
107  }
108  else if ((*a1).prefix>(*a2).prefix)
109  {
110       return 1;
111  }
112  else
113  {
114       return 0;
115  }
116}
117#endif
118
119/*
120  Please see cidrmerge.h
121*/
122void sort_entries(struct entry *addr,unsigned int
  len)
123{
124  #ifndef OPTIMIZED_SORT
```

```
125  /*standard C quicksort*/
126  qsort(addr,len,sizeof(struct
  entry),(cmp_type)cmp_entry);
127  #else
128  /*Optimized quicksort. Thanks to Michael Tokarev*/
129  #include "optimized-sort.h"
130  #define cmp_entry(a1,a2) (((a1)->network<(a2)-
  >network) || (((a1)->network==(a2)->network) && ((a1)-
  >prefix<(a2)->prefix)))
131
132  if (len>1)
133  {
134       /*printf("SORT: addr[0] %d len
  %d\n",addr[0].network,len);*/
135       QSORT(struct entry,addr,len,cmp_entry);
136  }
137  #endif
138}
139
140#ifdef INLINE
141__inline
142#endif
143static void expand(struct entry
  **expanded_list,struct entry to_expand,struct entry
  *white,unsigned int len_white,unsigned int n,unsigned
  int *current_position,unsigned int *size_expanded)
144{
145  int
  first=*current_position,last=*current_position+n-1;
146  int first_conflict,last_conflict,i;
147  struct entry tmp_small,tmp_big;
148  struct entry *result=*expanded_list;
149
150  *current_position+=n;
151
152  if (*current_position>*size_expanded)
153  {
```

```
154       /* we need to increase expanded_list array size
  */
155       #ifdef LIBRARY_DEBUG
156       printf("BEFORE realloc: size_expanded: %u, MEM:
  %u\n", *size_expanded,(*size_expanded)*sizeof(struct
  entry));
157       #endif
158
159       *size_expanded+=BUCKET;
160       #ifdef LIBRARY_DEBUG
161       printf("AFTER realloc size_expanded: %u,MEM:
  %u\n", *size_expanded,(*size_expanded)*sizeof(struct
  entry));
162       #endif
163
164
  result=*expanded_list=realloc(*expanded_list,(*size_exp
  anded)*sizeof(struct entry));
165       if (result==NULL)
166       {
167            fprintf(stderr,"Error reallocating %u
  bytes\n",(*size_expanded)*sizeof(struct entry));
168            exit(1);
169       }
170  }
171
172  #ifdef LIBRARY_DEBUG
173  printf("\nTO EXPAND n %u:\n",n);
174
  print_address(stdout,to_expand.network,to_expand.prefix
  );
175  printf("START WHITE len_white %u:\n",len_white);
176  print_addresses(stdout,white,len_white,NULL,0);
177  printf("END   WHITE\n");
178  #endif
179
180  while (first<=last)
181  {
```

```
182
 tmp_small.prefix=tmp_big.prefix=to_expand.prefix+1;
183       tmp_small.network=to_expand.network;
                                        /*last network
 bit set to 0 */
184       tmp_big.network=to_expand.network+(0x1<<(32-
 tmp_small.prefix));    /*last network bit set to 1 */
185
186       if ( (white[0].network &
 TONETMASK(tmp_small.prefix) ) == tmp_small.network )
187       {
188            /* conflicting whith tmp_small */
189            i=1;
190            /* search for first conflicting position
 with tmp_big*/
191            while((i<len_white) && !
 ( (white[i].network & TONETMASK(tmp_big.prefix) ) ==
 tmp_big.network ))
192            {
193                 i++;
194            }
195            first_conflict=i;
196            /* search for last conflicting position */
197            while((i<len_white) && ( (white[i].network
 & TONETMASK(tmp_big.prefix) ) == tmp_big.network ))
198            {
199                 i++;
200            }
201            last_conflict=i;
202            /* check if there is at least one
 conflicting network into whitelist (apart from the
 first one)*/
203            if (first_conflict!=len_white)
204            {
205                 if
 (white[first_conflict].prefix!=tmp_big.prefix)
206                 {
207
208                      #ifdef LIBRARY_DEBUG
```

```
209                     printf("RE-CONFLICT BIG first %u
  last %u\n",first_conflict,last_conflict);
210
  print_address(stdout,tmp_big.network,tmp_big.prefix);
211                     #endif
212
213
  result[last].network=*current_position;
214
  result[last].prefix=EXPANDED_PREFIX+white[first_conflic
  t].prefix-tmp_big.prefix;
215
216
  expand(expanded_list,tmp_big,&white[first_conflict],las
  t_conflict-first_conflict,white[first_conflict].prefix-
  tmp_big.prefix,current_position,size_expanded);
217                 }
218                 else
219                 {
220                     #ifdef LIBRARY_DEBUG
221                     printf("INVALIDATING BIG
  POSITION %u\n",last);
222                     #endif
223
  result[last].prefix=INVALID_PREFIX;
224                 }
225             }
226             else
227             {
228                 #ifdef LIBRARY_DEBUG
229                 printf("VALID BIG position
  %u:\n",last);
230
  print_address(stdout,tmp_big.network,tmp_big.prefix);
231                 #endif
232                 result[last].network=tmp_big.network;
233                 result[last].prefix=tmp_big.prefix;
234             }
235             to_expand.network=tmp_small.network;
```

```
236           to_expand.prefix=tmp_small.prefix;
237           last--;
238      }
239      else
240      {
241           /* conflicting with tmp_big */
242           i=1;
243           /* search for first conflicting position
 */
244           while((i<len_white) && !
 ( (white[i].network & TONETMASK(tmp_small.prefix) ) ==
 tmp_small.network ))
245           {
246                i++;
247           }
248           first_conflict=i;
249           /* search for last conflicting position */
250           while((i<len_white) && ( (white[i].network
 & TONETMASK(tmp_small.prefix) ) == tmp_small.network ))
251           {
252                i++;
253           }
254           last_conflict=i;
255
256           /* check if there is at least one
 conflicting network into whitelist (apart from the
 first one)*/
257           if (first_conflict!=len_white)
258           {
259                if
 (white[first_conflict].prefix!=tmp_small.prefix)
260                {
261                     #ifdef LIBRARY_DEBUG
262                     printf("RE-CONFLICT LOW first %u
 last %u\n",first_conflict,last_conflict);
263
 print_address(stdout,tmp_small.network,tmp_small.prefix
 );
```

```
264                     #endif
265
266
 result[first].network=*current_position;
267
 result[first].prefix=EXPANDED_PREFIX+white[first_confli
 ct].prefix-tmp_small.prefix;
268
269
 expand(expanded_list,tmp_small,&white[first_conflict],l
 ast_conflict-
 first_conflict,white[first_conflict].prefix-
 tmp_small.prefix,current_position,size_expanded);
270                 }
271                 else
272                 {
273                     #ifdef LIBRARY_DEBUG
274                     printf("INVALIDATING LOW
 POSITION %u\n",first);
275                     #endif
276
 result[first].prefix=INVALID_PREFIX;
277                 }
278             }
279             else
280             {
281                 #ifdef LIBRARY_DEBUG
282                 printf("VALID LOW position
 %u:\n",first);
283
 print_address(stdout,tmp_small.network,tmp_small.prefix
 );
284                 #endif
285
286
 result[first].network=tmp_small.network;
287
 result[first].prefix=tmp_small.prefix;
288             }
```

```
289           to_expand.network=tmp_big.network;
290           to_expand.prefix=tmp_big.prefix;
291           first++;
292      }
293
294      #ifdef LIBRARY_DEBUG
295      printf("first %d last %d\n",first,last);
296      #endif
297  }
298
299  #ifdef LIBRARY_DEBUG
300  printf("EXPAND END. Local position %d, current
  position %u:\n",n,*current_position);
301  printf("\n");
302  #endif
303}
304
305/*
306  Please see cidrmerge.h
307*/
308unsigned int apply_whitelist(struct entry
  *addr,struct entry **expanded_list,struct entry
  *white,unsigned int len1,unsigned int len2,unsigned int
  *size_expanded,int do_sort)
309{
310  unsigned int i1,i2,invalid;
311  unsigned int supermask,tmp,expanded_index=0;
312  struct entry tmp_entry;
313
314  #ifdef LIBRARY_DEBUG
315  unsigned int step=0;
316  printf("START apply_whitelist\n");
317  #endif
318  if (do_sort)
319  {
320      #ifdef LIBRARY_DEBUG
```

```
321       printf("Sorting entries\n");
322       #endif
323       sort_entries(addr,len1);
324       sort_entries(white,len2);
325  }
326
327  i1=i2=0;
328  while ((i1<len1)&&(i2<len2))
329  {
330       #ifdef LIBRARY_DEBUG
331       printf("STEP=%u I1=%d I2=%d\n",step++,i1,i2);
332       #endif
333       #ifdef LIBRARY_DEBUG_FULL
334       print_addresses(stdout,addr,len1,NULL,0);
335       #endif
336
337
  supermask=TONETMASK(addr[i1].prefix)&TONETMASK(white[i2
  ].prefix);
338
339       if
  ((addr[i1].prefix<=32)&&(addr[i1].network&supermask)==(
  white[i2].network&supermask))
340       {
341            #ifdef LIBRARY_DEBUG
342            printf("CONFLICT\n");
343
  print_address(stdout,addr[i1].network,addr[i1].prefix);
344
  print_address(stdout,white[i2].network,white[i2].prefix
  );
345            #endif
346            if (addr[i1].prefix<white[i2].prefix)
347            {
348                 /*we have to expand the network*/
349
350                 /*invalidate all addr[i1] subnetworks
```

```
 */
351                invalid=i1+1;
352                while
 ((invalid<len1)&&(addr[invalid].network&supermask)==(wh
 ite[i2].network&supermask))
353                {
354                     /*address is already present in
 the expanded network, just drop it*/
355                     #ifdef LIBRARY_DEBUG
356                     printf("INVALIDATING ");
357
 print_address(stdout,addr[invalid].network,addr[invalid
 ].prefix);
358                     #endif
359
 addr[invalid].prefix=INVALID_PREFIX;
360
361                     invalid++;
362                }
363
364                invalid-=i1+1; /*invalid represents
 the number of invalidated positions*/
365
366                #ifdef LIBRARY_DEBUG
367                printf("INVALID %u LEN1
 %u\n",invalid,len1);
368                #endif
369
370                tmp=i2;
371                i2+=1;
372                while ( (i2<len2) &&
 ((addr[i1].network&supermask)==(white[i2].network&super
 mask)) )
373                {
374                     i2++;
375                }
376
377                #ifdef LIBRARY_DEBUG
```

```
378                 printf("EXPAND expanded_index %u
  whitelist elements: %u num expand
  %d\n",expanded_index,i2-tmp,white[tmp].prefix-
  addr[i1].prefix);
379                 #endif
380
381                 tmp_entry.network=addr[i1].network;
382                 tmp_entry.prefix=addr[i1].prefix;
383
384
  addr[i1].prefix=EXPANDED_PREFIX+white[tmp].prefix-
  addr[i1].prefix;
385                 addr[i1].network=expanded_index;
386                 /* expanded positions are clean and
  optimized */
387
  expand(expanded_list,tmp_entry,&(white[tmp]),i2-
  tmp,white[tmp].prefix-
  tmp_entry.prefix,&expanded_index,size_expanded);
388
389                 i1+=invalid+1;
390
391                 #ifdef LIBRARY_DEBUG
392                 printf("END MAIN EXPAND
  expanded_index %u i1: %u i2:
  %u\n",expanded_index,i1,i2);
393                 #endif
394
395           }
396           else
397           {
398                 /*just invalidating entry i1*/
399                 #ifdef LIBRARY_DEBUG
400                 printf ("INVALIDATING ");
401
  print_address(stdout,addr[i1].network,addr[i1].prefix);
402                 #endif
403                 addr[i1].prefix=INVALID_PREFIX;
404                 i1++;
```

```
405           }
406
407       }
408       else if ((addr[i1].prefix==INVALID_PREFIX)||
  (addr[i1].network&supermask)<=(white[i2].network&superm
  ask))
409       {
410           i1++;
411       }
412       else
413       {
414           i2++;
415       }
416  }
417  #ifdef LIBRARY_DEBUG
418  printf("END apply_whitelist\n");
419  #endif
420  return expanded_index;
421}
422
423/*
424  Please see cidrmerge.h
425*/
426unsigned int optimize(struct entry *addr,unsigned int
  len,int do_sort)
427{
428  unsigned int i,cur;
429  unsigned int tmp_net;
430
431  #ifdef LIBRARY_DEBUG
432  unsigned int step=0;
433  printf("START optimize\n");
434  #endif
435
436  i=0;   /*pointer to last valid position*/
437  cur=1; /*pointer to next addr to analize*/
```

```
438
439  if (len <= 1)
440  {
441       /* empty or sigle element array is optimized by
  definition.*/
442       return len;
443  }
444
445  if (do_sort)
446  {
447       sort_entries(addr,len);
448  }
449
450  /*Find first valid address and move it to first
  position*/
451  while ((addr[0].prefix==INVALID_PREFIX) &&
  (cur<len))
452  {
453       #ifdef LIBRARY_DEBUG
454       printf ("SEARCH FIRST I: %d CUR: %d\n",i,cur);
455       #endif
456       if (addr[cur].prefix!=INVALID_PREFIX)
457       {
458            addr[0].network=addr[cur].network;
459            addr[0].prefix=addr[cur].prefix;
460            addr[cur].prefix=INVALID_PREFIX;
461       }
462       cur++;
463  }
464
465  while (cur<len)
466  {
467       #ifdef LIBRARY_DEBUG
468       printf ("STEP: %u I: %d CUR:
  %d\n",step++,i,cur);
469       #endif
```

```
470       #ifdef LIBRARY_DEBUG_FULL
471       print_addresses(stdout,addr,len,NULL,0);
472       #endif
473
474       /*check for expanded networks, they can never
 conflicts*/
475       if (addr[cur].prefix>=EXPANDED_PREFIX)
476       {
477            #ifdef LIBRARY_DEBUG
478            printf ("COPY EXPANDED I: %d CUR:
 %d\n",i,cur);
479            #endif
480            i++;
481            addr[i].network=addr[cur].network;
482            addr[i].prefix=addr[cur].prefix;
483            cur++;
484            while ((addr[i].prefix>=EXPANDED_PREFIX)
 && (cur<len))
485            {
486                 #ifdef LIBRARY_DEBUG
487                 printf ("COPY ADDR[CUR] FOR EXPANDED
 I: %d CUR: %d\n",i,cur);
488                 #endif
489                 if (addr[cur].prefix!=INVALID_PREFIX)
490                 {
491                      i++;
492                      if (cur != i)
493                      {
494
 addr[i].network=addr[cur].network;
495
 addr[i].prefix=addr[cur].prefix;
496
 addr[cur].prefix=INVALID_PREFIX;
497                      }
498                 }
499                 cur++;
```

```
500           }
501      }
502      else
503      {
504           /*If this test will fail we just skip
  addr[cur]*/
505           if
  ((addr[cur].prefix<=32)&&((addr[cur].network&TONETMASK(
  addr[i].prefix))!=addr[i].network))
506           {
507                tmp_net=TONETMASK(addr[i].prefix-1);
508
509                if
  ( (addr[i].prefix==addr[cur].prefix) &&
  ( (addr[i].network&tmp_net) ==
  (addr[cur].network&tmp_net) ) )
510                {
511                     #ifdef LIBRARY_DEBUG
512                     printf ("COLLAPSE I: %d CUR:
  %d\n",i,cur);
513                     #endif
514
515                     if (i>0)
516                     {
517
  addr[cur].prefix=addr[i].prefix-1;
518                          addr[cur].network&=tmp_net;
519                          i--;
520                     }
521                     else
522                     {
523
  addr[i].prefix=addr[i].prefix-1;
524                          addr[i].network&=tmp_net;
525                          cur++;
526                     }
527                }
528                else
```

```
529                 {
530                      i++;
531
532
  addr[i].network=addr[cur].network;
533                      addr[i].prefix=addr[cur].prefix;
534
535                      cur++;
536                 }
537            }
538            else
539            {
540                 #ifdef LIBRARY_DEBUG
541                 printf ("SKIP CUR: %d\n",cur);
542                 #endif
543
544                 cur++;
545            }
546       }
547  }
548
549  #ifdef LIBRARY_DEBUG
550  printf("END optimize\n");
551  #endif
552
553  return i+1;
554}
555
```

cidrmerge.c

```
556#include <stdio.h>
557#include <stdlib.h>
558#include <string.h>
559#include <math.h>
```

```
560#include "cidrmerge.h"
561
562#ifdef TIMESTAMP
563#include <time.h>
564#endif
565
566static char *version="1.5.1";
567static char *author="Daniele Depetrini
  (depetrini@libero.it)";
568
569void print_address(FILE *file,unsigned int net,
  unsigned char pref);
570
571/* static array to speedup prefix print */
572static char *dotted[]={
573                                      "0","1","2","3","
  4","5","6","7","8","9", \
574
  "10","11","12","13","14","15","16","17","18","19", \
575                                      "20","21","22","2
  3","24","25","26","27","28","29", \
576                                      "30","31","32","3
  3","34","35","36","37","38","39", \
577                                      "40","41","42","4
  3","44","45","46","47","48","49", \
578                                      "50","51","52","5
  3","54","55","56","57","58","59", \
579                                      "60","61","62","6
  3","64","65","66","67","68","69", \
580                                      "70","71","72","7
  3","74","75","76","77","78","79", \
581                                      "80","81","82","8
  3","84","85","86","87","88","89", \
582                                      "90","91","92","9
  3","94","95","96","97","98","99", \
583                                      "100","101","102"
  ,"103","104","105","106","107","108","109", \
584
  "110","111","112","113","114","115","116","117","118","
  119", \
```

```
585                                     "120","121","122"
  ,"123","124","125","126","127","128","129", \
586                                     "130","131","132"
  ,"133","134","135","136","137","138","139", \
587                                     "140","141","142"
  ,"143","144","145","146","147","148","149", \
588                                     "150","151","152"
  ,"153","154","155","156","157","158","159", \
589                                     "160","161","162"
  ,"163","164","165","166","167","168","169", \
590                                     "170","171","172"
  ,"173","174","175","176","177","178","179", \
591                                     "180","181","182"
  ,"183","184","185","186","187","188","189", \
592                                     "190","191","192"
  ,"193","194","195","196","197","198","199", \
593                                     "200","201","202"
  ,"203","204","205","206","207","208","209", \
594
  "210","211","212","213","214","215","216","217","218","
  219", \
595                                     "220","221","222"
  ,"223","224","225","226","227","228","229", \
596                                     "230","231","232"
  ,"233","234","235","236","237","238","239", \
597                                     "240","241","242"
  ,"243","244","245","246","247","248","249", \
598                                     "250","251","252"
  ,"253","254","255"
599                                     };
600
601#ifdef INLINE
602__inline
603#endif
604static unsigned char collect_octet(char *line,int
  current_position,int *end_position)
605{
606       unsigned int number=0;
607
608       *end_position=current_position;
```

```
609       while (line[current_position] >='0' &&
  line[current_position] <='9')
610       {
611            number*=10;
612            number+=line[current_position]-48;
613            current_position++;
614       }
615
616       /*update return pointer only if it's a valid
  octect*/
617       if (number<256)
618       {
619            *end_position=current_position;
620       }
621
622       return number;
623}
624/*
625  Parse a text string in CIDR format
626*/
627static int retrieve_line(char* line,struct entry
  *res)
628{
629  int i,n;
630  int end=-1,stop=0;
631
632  i=0;
633  n=24;
634  res->network=0;
635  res->prefix=INVALID_PREFIX;
636  while ((n>=0) && !stop)
637  {
638       #ifdef INPUT_DEBUG
639       printf("Before collect i:%d end:%d
  line[i]:%c\n",i,end,line[i]);
640       #endif
```

```
641
642       res->network|=collect_octet(line,i,&end)<<n;
643
644       #ifdef INPUT_DEBUG
645       printf("After collect i:%d end:%d
  line[end]:%c\n",i,end,line[end]);
646       #endif
647
648       if (end==i)
649       {
650            if (line[end-1]=='.')
651            {
652                 n+=8;
653            }
654            else
655            {
656                 break;
657            }
658       }
659       switch (line[end])
660       {
661            case '.':
662                 if (n>0)
663                 {
664                      i=end+1;
665                 }
666                 else
667                 {
668                      stop=1;
669                 }
670                 break;
671            case '/':
672                 if (n==0)
673                 {
674                      i=end+1;
```

```
675                     #ifdef INPUT_DEBUG
676                     printf("Before collect prefix
 i:%d end:%d line[i]:%c\n",i,end,line[i]);
677                     #endif
678                     res-
 >prefix=collect_octet(line,i,&end);
679                     if ((line[end] != '\n') &&
 (line[end] != ' '))
680                     {
681                          res->prefix=INVALID_PREFIX;
682                     }
683                     #ifdef INPUT_DEBUG
684                     printf("After collect prefix
 i:%d end:%d prefix:%d\n",i,end,res->prefix);
685                     #endif
686                }
687
688                stop=1;
689                break;
690           case ' ':
691                /*no break here*/
692                fprintf(stderr,"WARNING: not
 considering characters after space in line %s",line);
693           case '*':
694                if ((line[end]=='*') &&
 (line[end+1] != '\n'))
695                {
696                     res->prefix=INVALID_PREFIX;
697                     n=32;
698                }
699           case '\0':
700                line[end]='\n';
701                line[end+1]='\0';
702           case '\n':
703                if (n<24)
704                {
705                     res->prefix=32-n;
```

```
706                 }
707                 stop=1;
708                 break;
709             default:
710                 stop=1;
711                 break;
712         }
713         n-=8;
714  }
715  #ifdef INPUT_DEBUG
716  printf("Final prefix: %d\n",res->prefix);
717  #endif
718  if (res->prefix>32)
719  {
720         return 0;
721  }
722
723  /*   final sanity check, very important: library
  function will not work if this costraint is not
  enforced */
724  if ((res->network&TONETMASK(res->prefix))!=res-
  >network)
725  {
726         return 0;
727  }
728
729  return 1;
730}
731
732#ifdef INLINE
733__inline
734#endif
735static int STRCPY(char *dst,char *src)
736{
737  int i=0;
738
```

```
739  while (src[i] != '\0')
740  {
741       dst[i]=src[i];
742       i++;
743  }
744  return i;
745}
746
747#ifdef OUTPUT_VERBOSITY
748int counter=0;
749#endif
750void print_address(FILE *file,unsigned int net,
  unsigned char pref)
751{
752  char buf[MAXCIDR+2];
753  int pos=0;
754#ifdef OUTPUT_VERBOSITY
755  fprintf(file,"N: %d\tVNet: %u\tMask:
  %d.%d.%d.%d\t",counter++,net,TONETMASK(pref)>>24,(TONET
  MASK(pref)&MASK2)>>16,(TONETMASK(pref)&MASK3)>>8,TONETM
  ASK(pref)&MASK4);
756
757/*     fprintf(file,"N: %d\tMask: %d.%d.%d.%d\t
  ",counter++,TONETMASK(pref)>>24,(TONETMASK(pref)&MASK2)
  >>16,(TONETMASK(pref)&MASK3)>>8,TONETMASK(pref)&MASK4);
758*/
759#else
760  if (pref!=INVALID_PREFIX)
761  {
762#endif
763       pos+=STRCPY(buf+pos,dotted[net>>24]);
764       buf[pos++]='.';
765       pos+=STRCPY(buf+pos,dotted[(net&MASK2)>>16]);
766       buf[pos++]='.';
767       pos+=STRCPY(buf+pos,dotted[(net&MASK3)>>8]);
768       buf[pos++]='.';
769       pos+=STRCPY(buf+pos,dotted[net&MASK4]);
```

```
770       buf[pos++]='/';
771       pos+=STRCPY(buf+pos,dotted[pref]);
772       buf[pos++]='\n';
773
774       buf[pos]='\0';
775       fputs(buf,file);
776
777#ifndef OUTPUT_VERBOSITY
778  }
779#endif
780}
781
782#ifdef INLINE
783__inline
784#endif
785unsigned int print_address2(unsigned int net,
  unsigned char pref,char *buf,unsigned int pos)
786{
787  if (pref!=INVALID_PREFIX)
788  {
789       pos+=STRCPY(buf+pos,dotted[net>>24]);
790       buf[pos++]='.';
791       pos+=STRCPY(buf+pos,dotted[(net&MASK2)>>16]);
792       buf[pos++]='.';
793       pos+=STRCPY(buf+pos,dotted[(net&MASK3)>>8]);
794       buf[pos++]='.';
795       pos+=STRCPY(buf+pos,dotted[net&MASK4]);
796       buf[pos++]='/';
797       pos+=STRCPY(buf+pos,dotted[pref]);
798       buf[pos++]='\n';
799  }
800  return pos;
801}
802
803int get_entries(FILE *f,struct entry ** addr,unsigned
  int *size)
```

```
804{
805  unsigned int i;
806  char *res;
807  char line[MAXLINE];
808
809  i=*size=0;
810  while ((res=fgets(line,MAXLINE,f)))
811  {
812       if (*size<=i)
813       {
814            *size+=BUCKET;
815            *addr=(struct entry
  *)realloc(*addr,sizeof(entry_t)*(*size));
816            if (addr==NULL)
817            {
818                 fprintf(stderr,"Error allocating %u
  bytes\n",sizeof(entry_t)*(*size));
819                 exit(1);
820            }
821
822       }
823
824       if (!retrieve_line(line,&((*addr)[i])))
825       {
826            fprintf(stderr,"Invalid line %s",line);
827       }
828       else
829       {
830            i++;
831       }
832  }
833  return i;
834}
835
836void print_addresses(FILE *f,struct entry *addr,int
  size,struct entry *expanded,int level)
```

```
837{
838  int i=0,pos=0;
839  char buf[BUFFER];
840
841  if (level>256)
842  {
843       /*Sanity ckeck */
844       fprintf(stderr,"Array too nested: internal
  error, aborting. Please report this issue with input
  files to depetrini@libero.it\n");
845       exit(1);
846  }
847
848  #ifdef OUTPUT_VERBOSITY
849  counter=0;
850  printf ("++TOPRINT: %u LEVEL %d++\n",size,level);
851  #endif
852
853  while (i<size)
854  {
855       if ((addr[i].prefix>=EXPANDED_PREFIX) &&
  expanded)
856       {
857            #ifdef OUTPUT_VERBOSITY
858            printf("EXPANDED position %d,num expanded
  %d, logical position %d\n",i,addr[i].prefix-
  EXPANDED_PREFIX,addr[i].network);
859            #endif
860            buf[pos]='\0';
861            fputs(buf,f);
862            pos=0;
863
  print_addresses(f,&(expanded[addr[i].network]),addr[i].
  prefix-EXPANDED_PREFIX,expanded,level+1);
864       }
865       else
866       {
```

```
867           if (addr[i].prefix>=EXPANDED_PREFIX)
868           {
869                printf("EXPANDED position: ");
870 print_address(f,addr[i].network,addr[i].prefix);
871           }
872           else
873           {
874                #ifdef OUTPUT_VERBOSITY
875 print_address(f,addr[i].network,addr[i].prefix);
876                #else
877 pos=print_address2(addr[i].network,addr[i].prefix,buf,p
 os);
878                if (pos>BUFFER-MAXCIDR-2)
879                {
880                     buf[pos]='\0';
881                     fputs(buf,f);
882                     pos=0;
883                }
884                #endif
885           }
886      }
887
888      i++;
889 }
890 if (pos>0)
891 {
892      buf[pos]='\0';
893      fputs(buf,f);
894 }
895
896 #ifdef OUTPUT_VERBOSITY
897 printf ("--END PRINT--\n");
898 #endif
```

```
899}
900
901int main (int argc, char *argv[])
902{
903  struct entry *addr=NULL;
904  struct entry *white_list=NULL;
905  struct entry *expanded_list=NULL;
906  unsigned int len1,len2,len_expanded=0;
907  unsigned int size1=0,size2=0,size_expanded=0;
908  FILE *WHITE;
909#ifdef TIMESTAMP
910  time_t now,start,prev;
911  FILE *PROFILE;
912  unsigned int in_input=0;
913#endif
914
915  if ((argc >1)&&(strcmp(argv[1],"-v")==0))
916  {
917       printf("%s by %s\n",version,author);
918       exit(0);
919  }
920
921  if ((argc >1)&&(strcmp(argv[1],"-h")==0))
922  {
923       printf("Usage: cidrmerge [whitelist file]
  [NOOPTIMIZE]\n");
924       exit(0);
925  }
926
927  /*sanity check*/
928  if (sizeof(len1)!=4)
929  {
930       fprintf(stderr,"Incompatible integer size (in
  bit): current system %d, supported 32.
  Aborting.\n",sizeof(unsigned int)*8);
931       exit (1);
```

```
932  }
933
934#ifdef TIMESTAMP
935  PROFILE=fopen("timestamp.out","w");
936  if (!PROFILE)
937  {
938       fprintf(stderr,"Error opening profile file
  timestamp,out, aborting\n");
939       exit (1);
940  }
941  start=prev=time(&now);
942  fprintf(PROFILE,"START %s",ctime(&start));
943#endif
944
945  len1=get_entries(stdin,&addr,&size1);
946
947  len2=0;
948
949  if (argc >1)
950  {
951       if ((WHITE=fopen(argv[1],"r")))
952       {
953
  len2=get_entries(WHITE,&white_list,&size2);
954            fclose(WHITE);
955       }
956       else
957       {
958            fprintf(stderr,"Error opening
  %s\n",argv[1]);
959            exit(1);
960       }
961  }
962
963#ifdef TIMESTAMP
964  in_input=len1;
```

```
965  prev=now;
966  now=time(&now);
967  fprintf(PROFILE,"%d get_entries\n", (int)(now-
  prev));
968#endif
969
970  sort_entries(addr,len1);
971  sort_entries(white_list,len2);
972#ifdef TIMESTAMP
973  prev=now;
974  now=time(&now);
975  fprintf(PROFILE,"%d sort_entries\n", (int)(now-
  prev));
976#endif
977
978  if (len2>0)
979  {
980
  len_expanded=apply_whitelist(addr,&expanded_list,white_
  list,len1,len2,&size_expanded,0);
981#ifdef TIMESTAMP
982       prev=now;
983       now=time(&now);
984       fprintf(PROFILE,"%d apply_whitelist\n",
  (int)(now-prev));
985#endif
986  }
987
988  if (argc < 3)
989  {
990       len1=optimize(addr,len1,0);
991#ifdef TIMESTAMP
992       prev=now;
993       now=time(&now);
994       fprintf(PROFILE,"%d optimize\n", (int)(now-
  prev));
995#endif
```

```
996  }
997
998  #ifdef LIBRARY_DEBUG
999  printf ("FINAL RESULT\n");
1000 #endif
1001
1002 print_addresses(stdout,addr,len1,expanded_list,0);
1003
1004 free(addr);
1005 if (len2>0)
1006 {
1007      free(white_list);
1008 }
1009 if (expanded_list)
1010 {
1011      free(expanded_list);
1012 }
1013
1014#ifdef TIMESTAMP
1015 prev=now;
1016 now=time(&now);
1017 fprintf(PROFILE,"%d print_addresses\n", (int)(now-
  prev));
1018#endif
1019
1020#ifdef TIMESTAMP
1021 now=time(&now);
1022 fprintf(PROFILE,"Blacklist: in input %u, expanded %u
  final optimize %u\n",in_input,len_expanded,len1);
1023 fprintf(PROFILE,"Whitelist: total %u\n",len2);
1024 fprintf(PROFILE,"Total execution time %d
  seconds.\n", (int)(now-start));
1025 fprintf(PROFILE,"END %s",ctime(&now));
1026 fclose(PROFILE);
1027#endif
1028
```

```
1029 return 0;
1030}
```

Makefile

```
1031#Compiler command
1032CC=gcc
1033
1034#Usual gcc flags
1035CC_FLAGS=-O3 -fPIC -g -c -Wall
1036#Enforce ansi compatibility
1037CC_FLAGS+= -ansi -pedantic
1038#Pack struct: this will reduce significantly memory
  usage (- 30%), but some platform cannot handle not
  aligned to word integers (SIGBUS will be generated in
  this case). It can cause incompatibility with code
  compiled without this flag too and program to be slower
  in some cases (never verified anyway), To test on
  target system.
1039#CC_FLAGS+=-fpack-struct
1040#cpu dependant, configure accordingly to final
  running machine. It can increase significantly
  performances
1041#CC_FLAGS+=-march=pentium4
1042#Profiling
1043#CC_FLAGS+=-pg
1044
1045#standard link flags
1046LINK_FLAGS=
1047#Linker profiling flags
1048#LINK_FLAGS+=-pg
1049
1050#Standard defines
1051STD_DEFINES=-D OPTIMIZED_SORT -D INLINE -D
  CHAR_PREFIX
1052#STD_DEFINES+=-D TIMESTAMP
1053#All debug defines. For program debugging only.
1054#DEBUG_DEFINES= -D LIBRARY_DEBUG_FULL -D
  LIBRARY_DEBUG -D INPUT_DEBUG -D OUTPUT_VERBOSITY
```

```
1055#DEBUG_DEFINES+=-D LIBRARY_DEBUG
1056#DEBUG_DEFINES+=-D LIBRARY_DEBUG_FULL
1057#DEBUG_DEFINES+=-D OUTPUT_VERBOSITY
1058
1059all :   cidrmerge
1060
1061lib:           liboptimize.so.1.5.1
1062
1063cidrmerge :         cidrmerge.o cidrmerge-lib.o
1064          ${CC} ${LINK_FLAGS} -o cidrmerge
  cidrmerge.o cidrmerge-lib.o
1065
1066cidrmerge.o :  cidrmerge.c cidrmerge.h
1067          ${CC} ${STD_DEFINES} ${DEBUG_DEFINES}
  ${CC_FLAGS} -c cidrmerge.c
1068
1069cidrmerge-lib.o :   cidrmerge-lib.c cidrmerge.h
1070          ${CC} ${STD_DEFINES} ${DEBUG_DEFINES}
  ${CC_FLAGS} -c cidrmerge-lib.c
1071
1072liboptimize.so.1.5.1 :   cidrmerge-lib.o
1073          ${CC} -shared -Wl,-soname,liboptimize.so.1
  -o liboptimize.so.1.5.1 cidrmerge-lib.o -lc -ldl
1074          ln -fs liboptimize.so.1.5.1
  liboptimize.so.1
1075          ln -fs liboptimize.so.1.5.1 liboptimize.a
1076
1077clean :
1078 rm -f *.o core
1079 rm -f cidrmerge liboptimize.so.1.5.1
  liboptimize.so.1 liboptimize.a
1080 rm -f timestamp.out gmon.out
```