Terza parte del laboratorio, ispirata da Ken Thompson. Pubblicò un articolo nel 1984 sulla rivista più prestigiosa dell’ ACM per celebrare un premio assegnato a un suo amico (il turing Award). In esso fu libero di dire tutto quello che voleva e scelse di mettere in guardia rispetto alle misure di sicurezza informatiche che si stavano evolvendo ma stavano anche diventando più deboli (se lui avesse messo nel sistema Unix una backdoor non ci sarebbe stato modo di accorgersene). In questo articolo spiegò quindi tutto l’occorrente per trattare un tale malware, presentando una serie di esempi. Noi ci concentreremo sul primo:

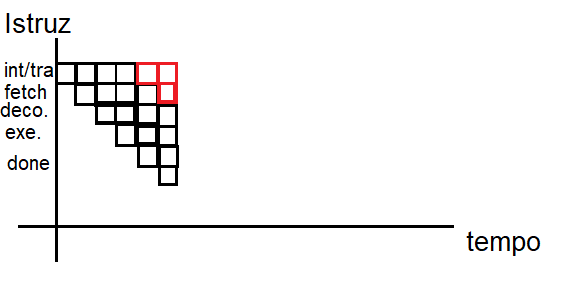
Esso è un programma abbastanza complesso e affronta il seguente problema. Bisogna poter stampare una stringa di caratteri in più modi diversi. Viene definita tale stringa che possa essere stampata la prima volta come array di caratteri e la seconda come stringa vera e propria. La sfida è scrivere un programma che generi in output il codice sorgente dello stesso programma. La sfida è causata dal fatto che in genere il codice sorgente è sempre più lungo dell’output, ma stampando più volte e in modi diversi il contenuto di una stessa variabile si può aggirare questo ostacolo. Per semplificarci la vita il prof ha organizzato la soluzione in maniera semiautomatica. Un programma è una parte del codice di Ken Thomson, mentre un altro prende in input tale pezzo di programma e manda in output il risultato finale (è spezzata la parte di inizializzazione). Si tratta quindi di comporre il file facendo una concatenazione delle due parti. L’output finale sarà quindi un programma a sua volta. La consegna alla fine è comunque quella di completare i programmi che sono incompleti.

Abbiamo visto i principi di Denning: Virtualizzazione, Anello Debole, Economia/Ridondanza e Minimo Privilegio.  
Guardiamo il minimo privilegio. Immaginiamo di avere il nostro sistema in cui gli utenti accedono alle risorse attraverso il Kernel, dopo che ne ha verificato il livello di autorità (e se hanno diritto di svolgere tale operazione). Questo nucleo quindi filtra le richieste. Il principio del minimo privilegio dice come fare per organizzare il sistema di accesso alle risorse. L’idea è che bisogna identificare un insieme minimale di permessi da dare agli utenti: essi devono essere in grado di fare solo le cose che sono a loro fondamentali. Questo è il principio grazie al quale si può garantire al segretezza e la integrità delle informazioni: se un utente non ha stretta necessità di leggere un certo file non c’è motivo di dargli il privilegio di lettura su tale file, in questo modo si garantisce che l’informazione non venga divulgata inutilmente. Un esempio simile si può fare in scrittura: se si permette a un utente di modificare un file solo se non può farne a meno si garantisce meglio l’integrità dei dati presenti tra le risorse. Supponiamo che gli utenti siano gli elementi di un sistema di calcolo e le risorse siano effettivamente l’intero file system/l’accesso alla rete ecc. Se io vado a vedere le necessità dei vari componenti noto che l’uomo deve avere accesso in lettura e scrittura al file system, così come deve avere accesso alla connessione di rete. Tuttavia, non si può dare l’accesso ad ogni utente a ogni risorsa del sistema, come se ne esce? La risposta è che basta pensare all’utente non come a una persona unica, ma come dei programmi mandati in esecuzione dalla persona, che abbiano uno scopo predefinito.   
Questo metodo funziona solo se è valida l’idea del cambio di contesto. Per esempio, se io voglio accedere alla posta elettronica, in quel momento è necessario l’accesso alla rete, però in quel momento non serve andare a modificare i file. Dopo aver letto un e-mail magari si ha da modificare un file di codice e allora si chiude l’app di posta e si apre un editor di testo : così si ha un cambio di contesto, perché si è passati dal programma che necessitava dell’accesso alla rete a un altro che invece ha bisogno solo di modifica del file system.  
Ai due programmi di esempio, quindi, possono essere dati privilegi diversi. Il privilegio minimo è quindi più piccolo rispetto al darlo a tutti gli utenti in maniera uguale. Un’applicazione corretta di questo principio di Denning è alla base della sicurezza dei Sistemi. Uno dei nostri obiettivi sarà vedere come fare a realizzare ciò.

Abbiamo visto esempi di processori costruiti con architettura Von Neumann, architettura ancora oggi presa d’esempio (con notevoli varianti, sia chiaro). Tutto questo è stato propedeutico per l’introduzione di un esempio completo di sistema di calcolo. Ma prima di ciò parliamo di PRESTAZIONI del sistema di calcolo.

Abbiamo già visto che un modo per velocizzare le prestazioni dei programmi è agire sul cosiddetto “collo di bottiglia” della memoria RAM (il collegamento tramite memorie associative cache tra Processore e RAM). Infatti il Bus dati ha delle limitazioni sulla sua banda e può trasportare solo una certa quantità di dati a una certa velocità, le memorie Cache aiutano a ridurre il carico che va a pesare sul Bus fornendo “in anticipo” le informazioni necessarie al processore. È possibile aumentare ulteriormente la velocità del processore (a parità di frequenza di clock)? La risposta è sì, e una delle tecniche più efficaci che sono state sviluppate è la tecnica di Pipelining. Questo termine vuol dire costruire macchine che possano eseguire più istruzioni contemporaneamente. Supponiamo di partire da un processore che è basato su un ciclo di esecuzione delle istruzioni in 4 fasi. Abbiamo: fetch, decode, execute, (verify) interrupt/trap. Ogni istruzione richiede queste quattro fasi. Si ci chiede se si è in una situazione di errore, se si può procedere si va a prendere il contenuto di un registro-istruzione dalla memoria, si decodifica il contenuto del registro e si esegue l’istruzione. A questo punto ci sono istruzioni semplici e istruzioni complicate (che richiedono più cicli di clock). Quindi 1 ciclo di clock viene usati per la verifica, 1 o più cicli vengono usati per recuperare il codice dell’istruzione, 1 ciclo viene usato per la decodifica e 1 ciclo o più vengono usati per eseguire l’istruzione. Il minimo numero di cicli di clock per istruzione è quindi 4. La tecnica di Pipelining permette di agire a questo punto. Ipotizziamo di avere dispositivi fisici separati che fanno il controllo di queste operazioni. Abbiamo quindi 4 dispositivi, uno per fase. Tuttavia tali dispositivi non verrebbero usati contemporaneamente (durante il primo si utilizzerebbe solo uno, poi un altro, poi un altro ancora), la Pipeline risolve ciò. L’idea è quella di considerare una sequenza di istruzioni successive: potremmo pensare di fare sempre tutte le 4 fasi in parallelo (e far lavorare contemporaneamente i 4 dispositivi del processore) e ogni singola istruzione deve passare attraverso le 4 fasi. Quindi ogni istruzione passa prima alla verifica, poi al fetch, poi alla decodifica e alla esecuzione. Quando la prima istruzione finisce di usare il primo dispositivo e inizia a usare il secondo, il primo diventa libero e inizia ad occuparsi della verifica di int/trap per la seconda istruzione. Dopo due cicli di clock, la prima istruzione sarà passata al terzo dispositivo e nei primi due ci saranno la seconda e la terza istruzione, e così via.

Facciamoci un diagramma che rappresenti il numero di istruzioni eseguite in funzione del tempo



Se si tiene conto che alcune istruzioni possono richiedere più cicli per essere eseguite si possono aggiungere ulteriori componenti che si occupino di allungare la Pipeline e fornire cicli di clock in più per lo svolgimento di istruzioni complesse (per esempio si potrebbero avere 6 componenti). Così dopo la 4° (o 6° a seconda dei casi) istruzione il processore sarà in grado di eseguire un’istruzione per ciclo di clock (è chiaramente molto più veloce della variante che usa almeno 4 cicli per ciascuna istruzione). Il miglioramento risulta tanto più è evidente quanto più è lungo il programma che vogliamo eseguire. Purtroppo, i programmi che mandiamo in esecuzione non sono completamente sequenziali: le istruzioni di controllo cambiano il contenuto del registro contatore di programma, che viene utilizzato durante la fase di fetch. Dunque, l’esecuzione di un’istruzione comporta la modifica di un valore di nel PC ma che può avvenire dopo che è avvenuta già la fase di fetch di un’istruzione che sarebbe dovuta essere saltata: questo è un errore. Quindi bisogna tener conto di queste situazioni per aggiustare il sistema di Pipeline. Una soluzione possibile è di annullare/svuotare la pipeline che viene dopo (cioè le istruzioni che sono venute prima) il componente che esegue il fetch (esempio: l’istruzione 2 ordina di saltare alla n17. Fatto l’execute di i2, sono già stati effettuati i fetch di i3 e i4, dunque bisogna svuotare i componenti che effettuano decodifica ed esecuzione per i3 e i4 per evitare la condizione di errore). L’istruzione di salto può essere riconosciuta sia nella fase di decodifica che in quella di esecuzione (tuttavia il primo caso vale solo se il salto non è condizionale). Se il salto è condizionale può essere riconosciuto solo durante l’esecuzione (perché è necessario il valore del registro di stato).   
Un’alternativa, per evitare di dover svuotare la Pipeline è quella di cambiare il significato delle istruzioni di salto (non condizionale). Si è capito che è un salto nella decodifica, mentre si è già fatto il fetch di un’altra istruzione. Per non perdere il ciclo di clock i progettisti si son detti: no, non è un errore, è una feature. Si è inventato il salto ritardato, il salto quindi dovrà sempre avvenire al termine dell’istruzione successiva: a ogni istruzione di salto viene associato un delay slot (che rappresenta l’istruzione successiva che viene eseguita lo stesso). Così facendo si risparmia un ciclo di clock, perché se l’istruzione successiva al salto era sempre stata pensata per essere eseguita allora è accettabile, e anzi conveniente, che quell’istruzione venga eseguita comunque dopo aver ricevuto l’ordine di salto (quindi ogni programmatore dovrà tener conto del delay slot quando scrive if statements, ma anche no perché ci pensa il compilatore). Quindi un processore senza salto ritardato avrà le istruzioni I1, I2, I3, JUMP, I17… mentre un processore che ha il salto ritardato avrà I1, I2, JUMP, I3, I17,… La differenza tra i due è che il primo dovrà svuotare una già caricata I4. L’inversione dell’istruzione di salto e l’ultima istruzione da eseguire in ogni caso è in genere effettuata dal compilatore.

In caso l’istruzione prima di una di salto sia un’altra istruzione di salto, o nel caso in cui tale istruzione sia la prima in assoluto, si utilizza l’istruzione NOP (No Operation) che va ad occupare il Delay Slot: si spreca un ciclo di clock non facendo niente ma si garantisce la correttezza di esecuzione del codice (ciclo di clock che verrà “recuperato” risparmiando alla successiva operazione di salto). Se si mette (e si può mettere) un’istruzione diversa da NOP nel delay slot si guadagna un ciclo di clock nell’esecuzione del programma.

Questa tecnica funziona per il salto non condizionale ma non funziona per il salto condizionale (perché lì la distanza tra istruzioni è almeno 2). Per poter gestire le istruzioni di salto condizionale bisognerebbe inserire due delay slot, ma questo non è stato fatto perché si ha la sensazione che prevarrebbe l’uso dei NOP. Quindi la comunità informatica ha accettato di implementare solo la soluzione parziale che vale solo per i salti incondizionati, procedendo con la parziale (o totale) pulizia della Pipeline in caso di salto condizionale (Ecco perché i controlli tramite costrutti condizionali, come gli if else, sono “costosi” dal punto di vista dei tempi di esecuzione).   
Il software, quindi, ha un grande impatto sulla velocità di esecuzione dei programmi. Inoltre, un processore moderno senza un compilatore adeguato farà schifo.