Um Núcleo de Segurança Distribuído para Suporte a Protocolos Tolerantes a Intrusões [1]

Pan Jieke[1], Miguel Correia[2], Nuno Ferreira Neves[2], Paulo Veríssimo[2]

[1] Siemens, SA
Rua Irmãos Siemens, 1
2720-093 Amadora, Portugal

[2] LASIGE, Faculdade de Ciências da Universidade de Lisboa
Campo Grande, Edifício C6
1749-016 Lisboa, Portugal

pan.jieke@siemens.com {mpc,nuno,pjv}@di.fc.ul.pt

Resumo

A tolerância a faltas desde há muitos anos que trata do problema de projectar sistemas confiáveis usando componentes que podem falhar. Recentemente a aplicação deste paradigma no âmbito da segurança começou a suscitar atenção sob a designação de *tolerância a intrusões*. Este artigo insere-se no âmbito da investigação nesta área. Descreve o projecto de um núcleo distribuído seguro chamado ChongDong com base num pacote de segurança para Linux, o LIDS. O ChongDong é depois usado para suportar a execução de um protocolo tolerante a intrusões que permite a um conjunto de processos chegarem a acordo sobre um valor. O desempenho desse protocolo é avaliado.

1 Introdução

A dependência da nossa sociedade em relação a sistemas informáticos distribuídos tem crescido de forma explosiva nos últimos anos. No entanto, esta evolução tecnológica é acompanhada por novos problemas e desafios, entre os quais os relacionados com a segurança informática.

As soluções adoptadas na perspectiva clássica da segurança têm um cariz fundamentalmente de *prevenção*, sendo o objectivo o de construir sistemas "perfeitos", sem vulnerabilidades que possam ser exploradas por piratas informáticos, quer directamente, quer através de *malware* (p.ex., *worms*). No entanto a realidade mostra que esse objectivo é dificilmente atingível em sistemas complexos, e que os sistemas vivem num ciclo permanente: vulnerabilidade descoberta, sistemas atacado, vulnerabilidade remendada (*patched*).

A *tolerância a faltas* é uma disciplina que se foi desenvolvendo paralelamente à da *segurança* ao longo dos anos. Os objectivos de ambas são muito semelhantes: construir sistemas "de confiança", que se comportem de acordo com a sua especificação. No entanto, na tolerância a faltas, sem esquecer a prevenção,

[1]Este trabalho foi parcialmente suportado pela FCT através do projecto POSI/EIA/60334/2004 (RITAS) e do Large-Scale Informatic Systems Laboratory (LASIGE).

parte-se do princípio de que mesmo as componentes da melhor qualidade podem falhar e que, caso falhem, o sistema como um todo tem de continuar operacional.

Este paradigma da tolerância a faltas tem sido aplicado no âmbito da segurança sob a designação de *tolerância a intrusões*. Apesar da área ter surgido já há muitos anos [1] só mais recentemente ganhou visibilidade, sobretudo graças ao programa americano OASIS [2] e ao projecto europeu MAFTIA [3, 4]. Um pressuposto fundamental desta abordagem é o de que os ataques e as intrusões podem ser considerados como sendo *faltas*, englobadas na categoria das faltas arbitrárias ou *bizantinas* [5]. Na tolerância a intrusões, em vez de se tentar prevenir todas as intrusões, assume-se à partida que algumas delas irão ter sucesso, mas os sistemas são construídos de modo a que elas sejam toleradas.

O artigo segue uma aproximação que se baseia em enriquecer o sistema "normal" no qual todos os processos e a comunicação podem ser atacados, com um núcleo de segurança distribuído. Este núcleo é uma componente de um tipo que tem sido denominado *wormhole* [6]. Este tipo de componentes pretende lidar com algum tipo de incerteza em sistemas distribuídos. Neste contexto, considera-se que existe incerteza em termos de segurança, ou seja, que podem existir ataques e intrusões nas componentes do sistema. O *wormhole*, pelo contrário, é uma componente distribuída que não manifesta essa incerteza, ou seja, que é segura. Assim, ao contrário das operações normais fornecidas pelo sistema, o *wormhole* pode oferecer um conjunto de operações seguras que devolvem resultados fiáveis e correctos. O *wormhole* tem necessariamente de oferecer uma funcionalidade muito limitada, para que a sua segurança possa ser certificada usando algum tipo de metodologia formal.

O *wormhole* cuja concretização é descrita neste artigo é chamado ChongDong[2]. O sistema que utiliza o ChongDong é composto por um conjunto de máquinas que usam uma rede para comunicarem e cooperam entre si. As máquinas executam diverso *software*, incluindo um sistema operativo, bibliotecas, servidores de suporte (p.ex., servidores de autenticação), entre outros. Adicionalmente, as máquinas contêm um ChongDong local. Os ChongDongs locais são interligados por um canal de comunicação concretizado através de uma rede *Ethernet* privada e separada do canal de comunicação das aplicações. Cada máquina que contém um ChongDong local tem dois adaptadores de rede: um para a comunicação "normal" e outro para a comunicação privada do ChongDong. A arquitectura do sistema encontra-se na representada na figura 1.

O ChongDong é tratado como qualquer outro *software* do sistema e as aplicações utilizam os seus serviços sempre que necessário. A sua segurança é garantida pelos mecanismos de controlo de acesso e protecção fornecidos pelo sistema *LIDS (Linux Intrusion Detection System)*[3], um *patch* de segurança do sistema operativo Linux [7].

O artigo descreve também a concretização de um *protocolo de consenso tolerante a intrusões* baseado no ChongDong. O problema do consenso consiste em fazer com que um conjunto de processos distribuídos que escolhem um valor

[2] ChongDong significa *wormhole* em chinês...

[3] O nome é enganador. O LIDS não é realmente um IDS.

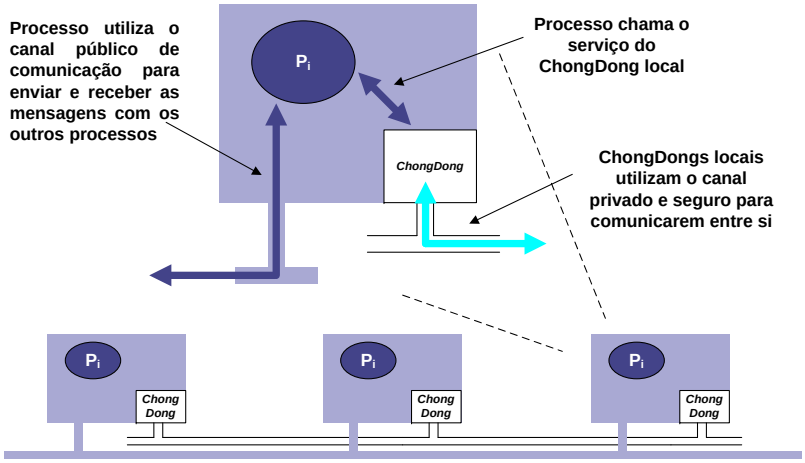

Figura 1: Arquitectura de um sistema que utiliza o ChongDong. A parte da aplicação é representada em cor escura e o ChongDong é representado em cor branca.

inicial, cheguem a acordo sobre um único dos valores escolhidos. O consenso é um problema fundamental em sistemas distribuídos, que tem grande interesse teórico e prático, pois é equivalente a diversos outros problemas, como a difusão de mensagens com ordem total [8]. No entanto, é importante notar que o ChongDong não se destina a suportar protocolos de consenso tolerantes a intrusões. Pelo contrário, o protocolo de consenso apresentado server apenas para ilustrar como o ChongDong pode ser usado para concretizar protocolos tolerantes a intrusões. Diversos serviços distribuídos tolerantes a intrusões poderiam ser construídos com base no ChongDong [9].

O problema do consenso é insolúvel de forma determinista em sistemas sobre os quais não se façam hipóteses temporais [10]. A realização de hipóteses temporais sobre tempos de processamento ou comunicação num sistema sujeito a ataques é muito arriscada, já que um ataque de negação de serviço pode muitas vezes quebrar esse tipo de hipóteses. No artigo todas as hipóteses temporais necessárias ao sistema estão encapsuladas no ChongDong que, como vimos, é projectado de forma a ser seguro. O protocolo de consenso usa apenas um serviço do ChongDong para realizar determinado passo da sua execução, sendo de resto executado no sistema "normal" inseguro.

As contribuições do artigo são as seguintes:

- é descrito o projecto de uma componente distribuída segura da classe dos *wormholes* com o suporte do LIDS;
- é mostrado como essa componente pode ser usada para concretizar um protocolo de consenso tolerante a intrusões;

- o desempenho do protocolo de consenso é avaliado.

2 Trabalho Relacionado

Um paradigma fundamental em segurança é o de controlo de acesso. Os mecanismos deste tipo são geralmente divididos em duas classes: DAC e MAC. O controlo de acesso discricionário (*DAC, Discretionary Access Control*) é o mais comum: cada utilizador pode especificar quem pode aceder aos recursos que lhe pertencem e como esse acesso pode ser feito (p.ex., só para leitura). No caso do controlo de acesso imperativo (*MAC, Mandatory Access Control*), os utilizadores têm permissões de acesso estabelecidas administrativamente.

Várias extensões e pacotes apareceram recentemente com o objectivo de suportar controlo de acesso MAC em Linux. Muitos desses pacotes são baseados na proposta IEEE 1003.1e (Posix.1e), entretanto retirada [11]. As *capabilities* fazem parte do próprio núcleo do Linux e permitem controlar o acesso de qualquer processo aos recursos do sistema, mesmo que este tenha privilégios de *superuser* [12]. O pacote LSM (*Linux Security Modules*) é um *patch* para Linux que suporta a concretização de módulos de controlo de acesso [13]. O *Security-Enhanced Linux*, desenvolvido pela *US National Security Agency*, é um conjunto de *patches* para o núcleo do Linux que suportam MAC [14].

Neste artigo é usado um pacote denominado LIDS para proteger o ChongDong local [7]. O LIDS concretiza MAC e define um conjunto de capacidades mais extenso e com controlo mais fino do que o do Linux original. Uma solução alternativa para proteger o *wormhole* seria colocar a sua parte local dentro de um módulo de *hardware*, como um coprocessador seguro ou um microcomputador PC-104. Essa solução alternativa é no entanto menos interessante sob o ponto de vista da possível distribuição do código por outros investigadores interessados em usá-lo.

O trabalho em *wormholes* para tolerância a intrusões teve início no projecto MAFTIA, tendo sido desenvolvidos, entre outros, protocolos de difusão fiável [15] e de consenso [16]. Estes protocolos eram baseados num *wormhole* denominado TTCB [17] e não conseguiam evitar inteiramente as hipóteses temporais sobre o sistema. Além disso, a TTCB considerava hipóteses de tempo mais fortes do que o ChongDong, já que era síncrona, ou tempo-real. A TTCB foi usada para obter uma solução genérica para a concretização de serviços distribuídos tolerantes a intrusões eficiente em termos de número de réplicas necessárias [18]. Foi também mostrado que é necessário usar um *wormhole* para suportar recuperação proactiva em sistemas assíncronos [19]. O conceito de *wormhole* foi ainda usado para lidar com a incerteza em termos de tempo [20].

Para além do protocolo de consenso tolerante a intrusões baseado na TTCB, vários foram propostos na literatura. O protocolo original de Lamport et al. era baseado em comunicação síncrona, uma hipótese que não fazemos neste artigo [5]. Outras aproximações foram usadas para concretizar este tipo de protocolos, como por exemplo a aleatoridade [21], a sincronia parcial [22] e os detectores de falhas [23].

3 O Sistema LIDS

Com a crescente popularidade do Linux na Internet, cada vez são encontradas mais vulnerabilidades nas suas aplicações, serviços e até no próprio núcleo [24]. Muitas dessas vulnerabilidades permitem que os piratas informáticos penetrem nos sistemas e, muitas vezes, obtenham privilégios de *superuser* (ou *"root"*). Com o modelo de protecção do actual sistema Linux (e dos restantes Unixes), uma vez obtidos os privilégios de *superuser*, o pirata passa a ter controlo sobre todos os recursos do sistema, podendo inclusive modificar o funcionamento do núcleo através da inserção de módulos nesse mesmo núcleo (*loadable kernel modules*), como fazem muitos *rootkits*[4]. Este poder ilimitado do *superuser* é o principal problema que o LIDS pretende resolver.

3.1 Objectivos do LIDS

O LIDS foi escrito por Xie Huagang e Philippe Biondi. Como já ficou dito, é um *patch* de segurança para o núcleo do Linux. Fornece ferramentas de administração que permitem aumentar a segurança do próprio núcleo através da concretização de *Mandatory Access Control (MAC)*. Quando o LIDS está activo, o acesso a ficheiros seleccionados, às operações de administração do sistema ou da rede, à memória, aos dispositivos de I/O, entre outros, podem ser proibidos até mesmo ao *superuser*. As protecções de cada recurso são definidos pelo administrador do sistema, sendo as operações reservadas ao administrador protegidas usando uma palavra de passe. O LIDS utiliza e extende as capacidades do Linux para controlar todo sistema.

Concretizando, o LIDS fornece as seguintes protecções:

- Ficheiros. O LIDS pode proteger ficheiros e directorias de modo a não permitir que um utilizador, mesmo que *superuser*, efectue alterações não autorizadas. A protecção é feita através da modificação das chamadas ao sistema no núcleo. Cada vez que haja acesso a um ficheiro são verificados o nome do ficheiro e o modo de protecção. Se estiver protegido contra um determinado modo de acesso, as operações correspondentes não são efectuadas.

- Processos. Em Linux, enquanto um processo estiver a correr no sistema, deve existir uma entrada na directoria */proc* com o nome igual ao *pid* do processo. Utilizando o comando *ps* consegue-se obter informação sobre os processos correntes e, através do comando *kill*, consegue-se terminar processos. Com os mecanismos do LIDS pode-se tornar um processo invisível, ininterrompível e insensível aos sinais do sistema.

- Núcleo. O LIDS fornece uma funcionalidade chamada *kernel sealing*. Só é permitida a inserção de módulos ou outras configurações antes do *kernel sealing*.

[4] Conjunto de ferramentas usado por um atacante para manter o acesso após intrusão num sistema.

O LIDS fornece ainda funcionalidades como a detecção de *port scans*[5] e a resposta a intrusões através da modificação de regras de uma *firewall* local. Este tipo de mecanismos não vão ser usados no artigo.

3.2 Utilização do LIDS

O LIDS permite proteger os recursos do sistema de duas formas. Primeiro, permite limitar o que todo e qualquer processo pode fazer, mesmo que tenha privilégios de *superuser*, através das capacidades. Por exemplo, se for desactivada a capacidade *CAP_SETUID*, qualquer processo fica impossibilitado de fazer *setuid*, ou seja, de modificar os seus *user IDs* real e efectivo [25]. Se for desactivada a capacidade *CAP_SYS_RAWIO*, para dar mais um exemplo, deixa de ser possível fazer acesso *raw* a dispositivos de I/O.

O LIDS permite também proteger ficheiros e processos específicos usando listas de controlo de acesso, ACLs. Mais uma vez, o objectivo é o de permitir ou não o acesso a determinados recursos do sistema de forma administrativa, ou seja, controlo de acesso MAC. No entanto, as ACLs são usadas para fazer essa definição de forma fina, para cada recurso. A definição de ACLs é realizada através da ferramenta *lidsconf*, que actua sobre um sujeito e um objecto. O sujeito é um programa, geralmente um binário ou um *shell script*. O objecto é o recurso ao qual o sujeito pode pretender aceder, por exemplo, ficheiros, directorias, capacidades e *sockets*. Seguem-se alguns exemplos de configuração de ACLs:

- *lidsconf -A -s programa1 -o ficheiro1 -j READONLY*
  O *lidsconf* adiciona (-A) um controlo de acesso que dá ao sujeito (-s) *programa1* o privilégio de leitura ao objecto (-o) *ficheiro1*. O *programa1* não pode eliminar nem alterar *ficheiro1*, mesmo que esteja a correr com privilégios de *superuser*.

- *lidsconf -A -o directoria -j READONLY*
  Como o sujeito não está definido, a regra é aplicada a todos os utilizadores e processos. É dado apenas o privilégio de leitura ao objecto *directoria*, ou seja, a todos os ficheiros nela incluídos.

- *lidsconf -A -s /bin/httpd -o CAP_NET_BIND_SERVICE 80 -j GRANT*
  O sujeito *httpd* (servidor Apache) recebe a capacidade CAP_NET_BIND _SERVICE para se poder ligar ao porto 80. Assume-se que a capacidade CAP_NET_BIND_SERVICE está desactivada globalmente, logo nenhum programa pode fazer *bind* a portos arbitrários.

3.3 Segurança do LIDS

Como já referimos, têm surgido nos últimos anos diversas vulnerabilidades no núcleo do Linux [24]. Algumas dessas vulnerabilidades permitem contornar os

[5] Tentativa de visualizar os portos abertos numa máquina, ou conjunto de máquinas, através da rede.

mecanismos de segurança fornecidos por esse núcleo. Sendo assim, é pertinente discutir se o LIDS é mais seguro do que o Linux. Do que foi visto até agora fica claro que o que o LIDS permite é a definição de políticas de segurança mais finas do que o Linux. Para esse efeito oferece um modelo mais complexo e com mais mecanismos do que o Linux original, sobretudo através do controlo de acesso MAC. No entanto isso não nos diz nada sobre a segurança do LIDS em si.

Os primeiros trabalhos em controlo de acesso identificaram a necessidade desses mecanismos serem concretizados num *monitor de referência*, uma componente que deve obedecer a três propriedades: completude (o monitor não pode ser contornado), isolamento (não pode ser corrompido) e verificabilidade (tem de ser formalmente verificável) [26]. A arquitectura do LIDS não inclui um monitor de referência, embora o LIDS concretize mecanismos de controlo de acesso semelhantes ao de um monitor de referência. O LIDS em si é apenas um *patch* de segurança do núcleo do Linux. Basicamente é código de melhoramento espalhado ao longo do núcleo e não uma componente encapsulada e que verifica as três propriedades indicadas.

Esta limitação do LIDS traduz-se numa limitação da concretização corrente do ChongDong, mas não põe em causa o modelo de *wormhole* seguro. Este é um modelo genérico que pode ser realizado de diversas formas, por exemplo em hardware para aplicações comerciais com requisitos exigentes.

4 O Wormhole ChongDong

A arquitectura de um sistema tolerante a intrusões que utilize o ChongDong pode ser dividida em duas partes (v. figura 1). Localmente existe um módulo ChongDong que corre em cada máquina. As aplicações ou processos, que potencialmente executam protocolos tolerantes a intrusões, utilizam um canal público de comunicação para efectuar o envio e a recepção das mensagens entre si. Se necessário comunicam também com o ChongDong existente localmente.

O principal serviço fornecido pelo ChongDong é um serviço de acordo distribuído seguro chamado *ChongDong_ TBA*. Quando um conjunto de processos pretende executar esse serviço, cada um deles chama localmente a função *ChongDong_ TBA* passando como argumento um valor inicial, sendo depois retornado a todos um resultado. O ChongDong fornece ainda um serviço que permite a cada processo obter um identificador único perante o *wormhole*. Esse serviço tem de ser chamado por todos os processos que pretendam invocar o *ChongDong_ TBA*.

4.1 API do ChongDong

A programação de aplicações que utilizem o ChongDong é feita mediante o recurso a uma biblioteca que define a API do ChongDong. A interface do serviço de *atribuição de identificador* consiste na seguinte função (em pseudo-código):

$$result, eid \leftarrow getEid()$$

A função retorna uma estrutura que contém dois elementos. O *eid* é o identificador atribuído pelo ChongDong e *result* indica se a atribuição correu correctamente ou não. A cada processo é atribuído um identificador único. Esta unicidade é garantida através da concatenação do endereço *MAC* (*Media Access Control*) da máquina onde corre o ChongDong com um valor aleatório (não repetido).

A interface do serviço *ChongDong_ TBA* é a seguinte:

$$error, value, prop\_ok \leftarrow ChongDong\_TBA(eid, elist, cid, quorum, value)$$

Os primeiros três argumentos da função são basicamente utilizados para identificação: *eid* é o identificador do processo que invoca o *ChongDong_ TBA*; *elist* é uma lista de identificadores dos processos envolvidos no *ChongDong_ TBA* e *cid* representa o identificador do consenso que vai ser executado. O *quorum* define o número mínimo de processos que devem chamar o *ChongDong_ TBA* antes de este fazer realmente acordo sobre os valores propostos. Este valor pode ser, por exemplo, $2f + 1$, em que f é o número máximo de processos que podem falhar. Finalmente, *value* é o valor proposto por cada processo. O valor é geralmente um *hash* criptográfico de dados de maior dimensão, já que *value* tem apenas 160 bits na concretização actual. Diferentes instâncias do serviço *ChongDong_ TBA* são identificadas por conjuntos de *(elist, cid, quorum)* diferentes.

O resultado retornado pela função é uma estrutura que contém os seguintes campos: *error* que indica se o *ChongDong_ TBA* foi bem executado ou não; *value* que contém o valor resultante do acordo; e *prop_ ok* que indica o número total dos processos que propuseram o valor resultante.

O valor retornado pelo serviço *ChongDong_ TBA* é o valor proposto mais vezes pelos processos que participaram activamente no serviço, ou seja, os processos que chamaram a função a tempo de o seu valor ser levado em conta. Assim, este serviço faz uma espécie de acordo sobre valores de tamanho limitado, os já referidos 160 bits.

4.2 Concretização do serviço *ChongDong_ TBA*

Temos usado o termo *processo* para denominar aquilo que executa o código da aplicação ou protocolo tolerante a intrusões. Um processo nesse contexto pode ser diversas coisas, como um processo no sentido clássico usado em sistemas operativos, um objecto distribuído (p.ex., no sentido usado no CORBA), ou outro tipo de componente de *software*. O ChongDong é concretizado através de um *processo*, não nesse sentido abstracto que temos vindo a usar mas no sentido de um processo Linux. Esse processo recebe pedidos dos outros processos através de um mecanismo de comunicação entre processos (*sockets Unix*). A comunicação através desses *sockets* é encapsulada nas chamadas feitas à API introduzida acima, de forma a que a aplicação apenas veja uma chamada a uma função.

A concretização da função *ChongDong_ TBA* dentro de um ChongDong local está apresentada em pseudo-código nos algoritmos 1 e 2, um para cada um

dos fluxos de execução (*threads*) do processo. Ao serem chamadas as funções *ChongDong_ TBA* nos ChongDongs locais, estes trocam entre si todos os valores propostos e executam um protocolo de consenso tolerante a faltas por paragem. Através da execução deste protocolo de consenso obtém-se o valor acordado que é devolvido aos processos que invocaram o serviço.

Esta utilização de um protocolo de *consenso* dentro do ChongDong pode causar alguma confusão, já que atrás foi referido que o artigo apresentaria um protocolo de consenso *tolerante a intrusões*. Na realidade o problema resolvido é o mesmo mas os tipos de faltas levados em conta são diferentes. O protocolo de consenso executado dentro do ChongDong não precisa de tolerar ataques e intrusões, apenas faltas de paragem (*"crashes"*), pois o ChongDong é construído para ser seguro. Assim, o protocolo usado é relativamente simples. Já o protocolo de consenso tolerante a intrusões precisa de tolerar todo o tipo de comportamento malicioso por partes dos processos, logo o problema é mais complicado e é esse que tratamos neste artigo.

Algoritmo 1 Protocolo ChongDong_TBA (tarefa principal)

```
Initialization: {Variáveis partilhadas pelas duas tarefas}
 1: TBABackup ← ∅                                          {Tabela dos TBAs atrasados}
 2: LocalTBA ← ∅                                           {Tabela dos TBAs Locais}
 3: VPropose ← ∅  {Tabela c/uma entrada por cada TBA para o qual haja quorum hashes}
 4: VResult ← ∅    {Tabela com os TBAs que foram decididos pelo algoritmo de consenso}

When ChongDong_TBA(eid, elist, cid, quorum, value) is called do

 5: if (elist, cid, quorum) ∈ TBABackup then
 6:    return TBABackup.getPropose(elist, cid, quorum)
 7: end if

 8: if exec-cons(elist, cid, quorum) then                  {o consenso já está a correr?}
 9:    loop
10:       if (elist, cid, quorum) ∈ VResult then
11:          return VResult.getTBA(elist, cid, quorum)
12:       end if
13:    end loop
14: end if

15: multicast(eid, elist, cid, quorum, value)
16: LocalTBA ← LocalTBA ∪ (eid, elist, cid, quorum, value)

17: loop
18:    if deliver(eid, elist, cid, quorum, value) then
19:       LocalTBA ← LocalTBA ∪ (eid, elist, cid, quorum, value)
20:    end if
21:    if (elist, cid, quorum) ∈ VResult then
22:       return ← VResult.getTBA(elist, cid, quorum)
23:    end if
24: end loop
```

O protocolo do *ChongDong_ TBA* começa por fazer algumas inicializações, como se pode ver no algoritmo 1. No início da execução, é verificado se existe ou

não uma entrada na tabela *TBABackup* com o mesmo *(elist, cid, quorum)*, pois a tabela *TBABackup* serve para guardar todos os resultados de execuções do serviço que já terminaram (linhas 5-7). Se existir, é retornado imediatamente o valor decidido, já que o processo não participou na execução do serviço na altura em que outros processos o fizeram.

Se não existir uma entrada com *(elist, cid, quorum)* na tabela *TBABackup* mas consenso relativo a essa execução do serviço já estiver a correr, o ChongDong local fica à espera do resultado da execução (linhas 8-14). Quando o consenso terminar, o valor decidido é retornado e o protocolo termina a sua execução.

Se nenhuma das duas situações se verificarem, o ChongDong local difunde a informação sobre a chamada ao serviço para todos os outros ChongDongs e adiciona essa informação à tabela *LocalTBA* (linhas 15-16). Em seguida fica à espera da entrega das mensagens enviadas pelos outros ChongDongs. Quando receber as mensagens dos outros ChongDongs, guarda a informação na tabela *LocalTBA* (linhas 17-24).

Algoritmo 2 Protocolo ChongDong_TBA (tarefa secundária)

```
 1: loop
 2:   while not LocalTBA.hasQuorum() do
 3:     wait()
 4:   end while

 5:   VPropose ← LocalTBA.tbasWithQuorum()
 6:   VResult ← early-consensus(VPropose)
 7:   LocalTBA.clean(VPropose)

 8:   for all TBA ∈ VResult do
 9:     TBABackup ← TBA
10:   end for
11: end loop
```

A tarefa secundária está geralmente bloqueada à espera de que haja pelo menos uma entrada na tabela *LocalTBA* para a qual haja *quorum* valores propostos por diferentes processos (algoritmo 2, linhas 2-4). Quando tal acontecer, a tarefa arranca o algoritmo de consenso tolerante a faltas por paragem *early-consensus* [27]. O valor a propor é um vector com uma entrada por cada instância do serviço *ChongDong_ TBA* para o qual já haja um *quorum* de $2f+1$ mensagens (linhas 5-6). Quando o algoritmo de consenso devolve todas as instâncias do serviço *ChongDong_ TBA* que tenham sido decididas pelo consenso, limpa as entradas na tabela *LocalTBA* para evitar repetição da execução do protocolo de consenso (linha 7). As instâncias do serviço *ChongDong_ TBA* decididas são adicionadas a *TBABackup* (linhas 8-10).

Como já referimos, Fischer, Lynch e Paterson provaram que o problema de consenso não tem solução determinista num sistema assíncrono sujeito a paragem de processos [10]. Mais tarde, Chandra e Toueg mostraram que o problema de consenso era solúvel se o sistema fosse extendido com um detector de falhas não fiável [28]. O algoritmo de *early-consensus* é inspirado no algoritmo fornecido neste último artigo, mas termina em menor número de ciclos

de envio de mensagens [27]. O algoritmo precisa de um detector de falhas que foi concretizado usando um esquema simples de envio de *heartbeats*, ou seja, de mensagens a dizer "estou vivo". Cada ChongDong local envia um *heartbeat* periodicamente para cada um dos outros. Se um ChongDong local fica determinado tempo sem receber um *heartbeat* de outro, considera-o suspeito de ter falhado. Estes detectores são não fiáveis na exacta medida em que se podem enganar nas suspeitas. No entanto o algoritmo funciona bem apesar desses enganos. Note-se que para concretizar o detector de falhas é necessária alguma sincronia, se bem que muito fraca. No entanto essa sincronia fica encapsulada no ChongDong.

4.3 Tornar o ChongDong seguro

A parte do ChongDong existente em cada máquina localmente foi tornada segura usando os mecanismos de controlo de acesso fornecidos pelo LIDS. O LIDS fornece duas ferramentas que permitem efectuar a sua administração e configuração: *lidsadm* e *lidsconf*. A *lidsadm* é uma ferramenta que permite administrar o LIDS. Permite realizar operações tais como activar ou desactivar o LIDS, efectuar o *kernel sealing* e visualizar o estado do LIDS. A ferramenta *lidsconf* é utilizada para configurar as ACLs. Estas duas ferramentas são utilizadas pelo administrador do LIDS, ao qual é atribuída uma palavra chave de administração.

Os aspectos mais importantes que devem ser levados em conta na protecção do ChongDong local são:

- A obtenção do identificador do processo do ChongDong (*pid*) deve ser inacessível para os utilizadores do sistema, incluindo o *superuser*. O processo do ChongDong não deve aparecer na tabela dos processos (correndo *ps*) nem na directoria */proc*.
- O processo do ChongDong deve ser protegido de modo a que não seja possível enviar-lhe sinais.
- O processo do ChongDong não deve ser interrompível, isto é, mesmo que executando o comando *kill -9* pelo *superuser* o ChongDong não é terminado.
- O código do ChongDong deve ser protegido contra modificação.

Essas protecções são forçadas através dos seguintes comandos:

- Esconder o processo do Chongdong:
  lidsconf -A -s ./ChongDong -o CAP_HIDDEN -j GRANT
- Proteger o processo do ChongDong de modo aos sinais do sistema não serem enviados ao processo:
  lidsconf -A -s ./ChongDong -o CAP_PROTECTED -j GRANT
- Tornar ininterrompível o processo do ChongDong:
  lidsconf -A -s ./ChongDong -o CAP_KILL_PROTECTED -j GRANT

- Proteger todo o código do ChongDong contra modificação:
  lidsconf -A -o ./ChongDong/ -j READONLY

A rede privada do ChongDong tem também de ser protegida através dos seguintes comandos:

- Desactivar a interface da rede e torná-la apenas acessível para o ChongDong
  lidsadm -S – -CAP_NET_ADMIN
  lidsconf -A -s ./ChongDong -o CAP_NET_ADMIN -j GRANT

- Desactivar a possibilidade de efectuar broadcast pela rede e torná-la apenas acessível para o ChongDong
  lidsadm -S – -CAP_NET_BROADCAST
  lidsconf -A -s ./ChongDong -o CAP_NET_BROADCAST -j GRANT

- Desactivar o serviço de *binding* para portos menores que o 1024 e torná-la apenas acessível para o ChongDong
  lidsadm -S – -CAP_NET_BIND_SERVICE
  lidsconf -A -s ./ChongDong -o CAP_NET_BIND_SERVICE -j GRANT

Até ao momento, o LIDS ainda não fornece o mecanismo de controlo de acesso para os dispositivos de rede. Mais especificamente, não é possível esconder a placa de rede ou dar autorização de acesso a apenas alguns processos. Por isso não foi possível efectuar a protecção total ao canal de comunicação do ChongDong. Esta funcionalidade está a ser acrescentada ao LIDS a pedido dos autores do artigo, sendo de prever que esteja disponível a médio prazo. Na realização actual os ChongDongs começam a sua operação autenticando-se mutuamente e estabelecendo chaves de sessão. Estas chaves são depois usadas para proteger a integridade da comunicação através de *Message Authentication Codes* (MACs) [29].

Para aumentar a confiança na segurança do ChongDong, foi feita uma análise estática do seu código [30]. Como toda a concretização do sistema foi feita em linguagem C++, foi utilizada uma ferramenta de análise estática de código para essa linguagem chamada *Flawfinder*[6]. Esta ferramenta detecta os problemas de segurança mais conhecidos do C++. Alguns exemplos são: os problemas relacionados com *buffer overflows*, como o uso das funções strcpy(), strcat(), gets(), sprintf(), e scanf() que não efectuam a verificação do tamanho do *buffer*; os problemas relacionados com o formato das *strings* (p.ex., printf(), snprintf() e syslog()); problemas relacionados com execuções de programas (p.ex., família exec(), system(), popen()); e problemas relacionados com a aquisição de números aleatórios fracos (random()).

O *Flawfinder* produz uma lista de potenciais vulnerabilidades ordenadas por gravidade. Os riscos mais graves são mostrados no topo da lista. O nível de risco não depende só das funções, mas também dos valores dos parâmetros da função.

[6] http://www.dwheeler.com/flawfinder/

Por exemplo, em muitos contextos o uso de uma *string* constante envolve menor risco do que um string variável.

A execução do *Flawfinder* com o código do ChongDong gerou uma lista de 59 possíveis vulnerabilidades, todas elas chamadas a *memcpy*. Como a função memcpy não efectua a verificação do tamanho do *buffer* de origem, se o tamanho deste for maior do que o do *buffer* de destino pode existir uma vulnerabilidade a ataques *buffer overflow*. Todas as 59 utilizações suspeitas do *memcpy* foram verificadas manualmente mas como os tamanhos dos *buffers* são sempre verificados antes de execução da função memcpy() essas vulnerabilidades na realidade não existem.

5 Consenso Tolerante a Intrusões

Sendo o objectivo do ChongDong suportar a execução de *protocolos tolerantes a intrusões*, foi concretizado um protocolo com essa propriedade que é significativo e que tem interesse prático: um protocolo de *consenso* [31][7]. Como foi referido atrás, os processos que correm o protocolo utilizam uma rede pública de comunicação para efectuar o envio e a recepção das mensagens entre si e comunicam com o ChongDong existente em cada máquina local para obter o valor de acordo distribuído. Este funcionamento encontra-se representado na figura 1. A comunicação através da rede pública é protegida usando canais seguros, por exemplo, conexões SSL [32]. Estes canais garantem apenas a integridade da comunicação, encontrando-se a cifra que suporta confidencialidade desligada.

O problema de consenso pode ser definido em termos das seguintes propriedades:

- *Validade.* Se todos os processos correctos propuserem o mesmo valor v, então qualquer processo correcto que decida, decide o valor v.
- *Acordo.* Todos os processos correctos decidem o mesmo valor.
- *Terminação.* Qualquer processo correcto acaba por decidir.

O protocolo é executado num ambiente assíncrono no qual os processos com intrusões se podem comportar de forma arbitrária. Por, exemplo, os processos nos quais ocorram intrusões podem tentar fazer conluio para quebrar as propriedades do consenso. Assim, o objectivo é garantir que os processos correctos conseguem decidir num valor com a presença de um subconjunto de processos maliciosos. Parte-se da hipótese de que no máximo f processos podem ser maliciosos e que o número total de processos N verifica $N \geq 3f + 1$. Por exemplo, pode-se assumir que não há intrusões em mais do que 2 processos ($f = 2$) e que o número de processos é $N = 7$. Este tipo de hipóteses é incontornável neste

[7] Convém recordar que o consenso que se discute aqui é diferente do que foi referido no capítulo anterior. O protocolo aqui apresentado pretende resolver consenso tolerante a faltas bizantinas, categoria na qual se incluem as intrusões. O *early-consensus* de Schiper só tolera paragem de processos.

tipo de sistemas, sendo fácil perceber que se a maior parte dos processos forem maliciosos, estes poderiam controlar o protocolo. A proporção $N = 3f + 1$ foi provada ser a que dá o número mínimo de processos necessários para tolerar f faltas bizantinas (intrusões) num sistema assíncrono [33, 22].

5.1 Protocolo

Cada processo que participa no protocolo executa o algoritmo 3. Os argumentos da função são a lista com os identificadores de processos (*elist*), um identificador da execução do protocolo (*cid*) e o valor proposto ($value_i$). O valor proposto pode ter um número de *bytes* arbitrário, por exemplo, um bit, centenas de bytes ou um ficheiro com vários MBytes.

A ideia de concretizar um consenso tolerante a intrusões sobre um serviço (*ChongDong_ TBA*) realizado através de um consenso tolerante a faltas por paragem pode parecer estranha. Por isso, convém recordar que o objectivo aqui é apenas ilustrar como o ChongDong pode ser usado para realizar protocolos tolerantes a intrusões; o consenso é apenas um exemplo. É importante também ter em conta que o ChongDong pode ser concretizado em hardware tendo disponíveis recursos mais limitados. Por isso, o *ChongDong_ TBA* faz acordo sobre valores de 160 bits, enquanto o protocolo de consenso tolerante a intrusões pode usar valores de tamanho arbitrário (por exemplo os referidos MBytes).

O protocolo é organizado em duas fases. Na primeira fase, os processos começam por utilizar a rede pública de comunicação para transmitir o valor para outros processos (linha 4). Como a comunicação é feita por um canal seguro, mesmo que um intruso controlasse a rede seria incapaz de corromper o conteúdo das mensagens e o valor seria correctamente recebido. O processo bloqueia-se nesta fase até receber $2f + 1$ valores de diferentes processos, o que garante que a maioria dos valores recebidos pertencem a processos correctos (assume-se que no máximo f são maliciosos; linhas 5-8).

Na segunda fase, os processos tentam chegar ao valor do consenso recorrendo ao serviço *ChongDong_ TBA* (linhas 10-25). Os processos começam por seleccionar o valor mais comum entre os valores propostos existentes em *bag* (linha 11). Esta condição simples garante que todos os processos correctos escolhem o mesmo valor se todos tiverem proposto valores idênticos (propriedade de Validade). Por outro lado, se os processos correctos tiverem diferentes valores propostos, então podem escolher valores distintos.

A seguir, os processos invocam *ChongDong_ TBA* com o *hash* criptográfico do valor escolhido, *Hash(v)*, e esperam pela decisão (linha 12). A maioria das chamadas a *ChongDong_ TBA* vêm de processos correctos porque o parâmetro *quorum* é definido com $2f + 1$. Quando *ChongDong_ TBA* retorna, todos os processos obterão o mesmo resultado, utilizando depois um pequeno teste para determinar o valor decidido. Basicamente, há dois resultados possíveis, dependendo dos valores propostos originais (linhas 13-17). Se todos os processos correctos (ou pelo menos $2f + 1$) tiverem o mesmo valor original, então esse é o valor decidido. Caso contrário, pode ser escolhido um valor por omissão.

Algoritmo 3 Consenso tolerante a intrusões (executado por cada p_i)

1: **function** consensus(elist, cid, $value_i$)

2: $hash\_v \leftarrow \perp$ {hash do valor decidido}
3: $bag \leftarrow \perp$ {bag para guardar os valores recebidos}

4: **multicast**(B-value, i, $value_i$)
5: **repeat**
6: **receive**(B-value, k, $value_k$)
7: $bag \leftarrow bag \cup value_k$
8: **until** (bag has $2f + 1$ values from diferente processes)
9: **activate task**($T1$, $T2$)

10: **task** $T1$:
11: $v \leftarrow$ most_Common_Value(bag)
12: out $\leftarrow$ *ChongDong_TBA*(eid, elist, cid, $2f + 1$, Hash(v))
13: **if** (at least $f + 1$ proposed the same value) **then**
14: *hash-v* = *out.value*
15: **else**
16: **return**(*default-value*)
17: **end if**

18: **task** $T2$:
19: **when** receive(Decide, k, $value_k$) **do**
20: $bag \leftarrow bag \cup value_k$
21: **end when**
22: **when** (*hash-v* $\neq \perp$) and ($\exists value_k \in bag$: Hash($value_k$ = *hash-v*)) **do**
23: **multicast**(Decide, i, $value_k$)
24: **return**($value_k$)
25: **end when**

5.2 Concretização

A comunicação entre os processo que executam consenso tolerante a intrusões é feita através de canais SSL. Foi utilizada a biblioteca OpenSSL para a sua concretização[8]. O protocolo faz *multicast* das mensagens e o SSL só suporta comunicação ponto-a-ponto. O mecanismo de *multicast* foi concretizado através de diversos envios de mensagens ponto-a-ponto.

Os processos obtêm o IP e o porto uns dos outros através de um servidor designado por *SSLCoordinationServer*. O seu endereço IP e porto são bem conhecidos. Quando um processo começa a sua execução, em primeiro lugar comunica com o *SSLCoordinationServer* através de um canal SSL para obter os endereços IP e portos dos outros processos que já estão em execução. Se for o primeiro a ligar-se fica à espera que os restantes processos se liguem. Caso contrário, obtém os endereços IP dos processos já em execução e tenta estabelecer uma conexão SSL com cada um. Depois, fica também à espera das ligações dos restantes processos e estabelece conexões com cada um. Quando todos os processos estão ligados o servidor *SSLCoordinationServer* deixa de estar envolvido no protocolo (v. figura 2).

A concretização actual parte do pressuposto de que o servidor *SSLCoordi-*

[8] http://www.openssl.org

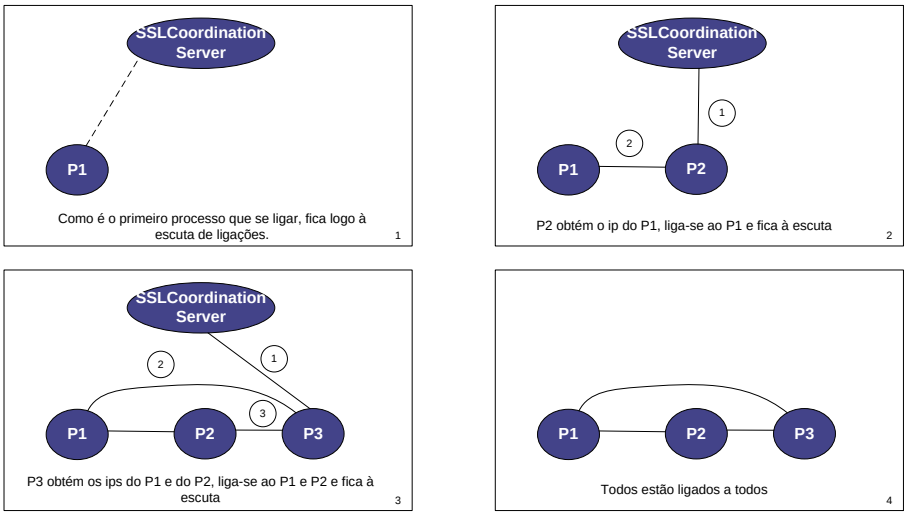

Figura 2: Estabelecimento de conexões SSL entre processos.

nationServer é seguro. Essa hipótese poderia ser relaxada tornando o servidor tolerante a intrusões usando um esquema simples baseado em sistemas de quorums [34].

5.3 Medição do desempenho

Esta secção apresenta medições do desempenho do protocolo de consenso tolerante a intrusões. A experiência envolveu quatro PCs com processadores Intel Pentium III a 500 Mhz, e 256MB SDRam PC133. Cada um continha dois adaptadores de rede 3Com 10/100. A rede pública e privada do ChongDong eram duas redes Fast-Ethernet *switched* a 100 Mbps. As versões de *software* usadas foram: Red Hat 9.0 com o núcleo Linux 2.4.25, LIDS 1.2.0, e g++ 3.2.2. A função de *hash* usada foi a concretização do OpenSSL do SHA-1 [35]. Em cada máquina foi executado um ChongDong e uma aplicação que corre o protocolo de consenso tolerante a intrusões. Assim, o número total de processos foi $N = 4$ e o número máximo de processos que podiam falhar $f = 1$.

Na experiência foi medida a variação da latência do protocolo com a dimensão do valor proposto por todos os processos (v. figura 3). O protocolo foi executado 100 vezes para cada valor no gráfico. A medição foi feita de seguinte maneira: inicialmente um processo difunde a mensagem de iniciação do protocolo para todos os processos envolvidos; quando todos os processos recebem a mensagem de iniciação, executam o protocolo e, quando terminam, calculam o tempo que demorou a sua execução; por fim é efectuada a média dos tempos obtidos.

O gráfico mostra que a latência aumenta com o tamanho do pedido, o que

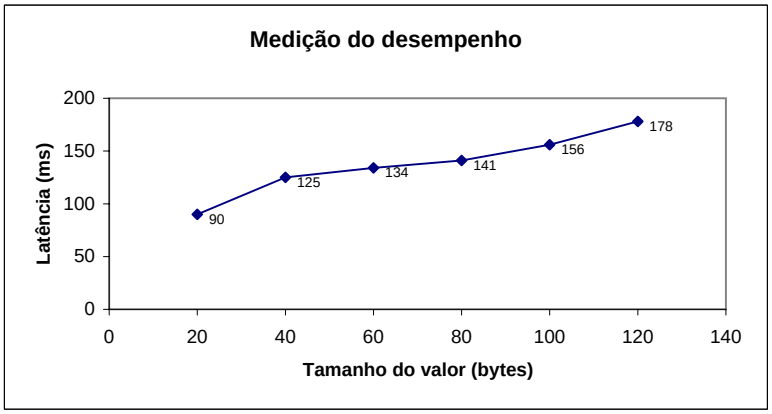

Figura 3: Medições do desempenho do protocolo de consenso tolerante a intrusões.

seria esperado dado que um maior comprimento do valor implica o consumo de tempo de processamento, por exemplo, o tempo gasto a efectuar o cálculo do *hash* dos valores.

Na concretização do serviço *ChongDong_ TBA*, foram utilizadas duas *threads* correspondentes às duas tarefas do protocolo apresentado na secção 4.2. O mecanismo de controlo de acesso concorrente às variáveis partilhadas pelas duas *threads* acabou por prejudicar fortemente o desempenho do protocolo. Por isso, as medidas apresentadas acima são ainda medidas preliminares, uma vez que o serviço está a ser novamente concretizado com apenas uma *thread*. No entanto, estes valores são da ordem de grandeza ou até inferiores aos obtidos com outros protocolos tolerantes a intrusões [36, 37].

6 Conclusão

O presente artigo aborda o problema da segurança em sistemas distribuídos através de uma perspectiva diferente da da segurança clássica: a *tolerância a intrusões*. O modelo usado considera que algumas componentes do sistema podem falhar arbitrariamente vítimas de intrusões, mas que existe uma componente da classe dos *wormholes* com propriedades mais fortes do que o resto do sistema, e que fornece um número reduzido de serviços de uma maneira segura e correcta.

Este artigo apresenta a concretização de um *wormhole* chamado ChongDong que fornece um serviço de acordo destinado a suportar a execução dos protocolos tolerantes a intrusões. A própria componente tem um canal de comunicação concretizado através de uma rede Ethernet privada separada do canal de comunicação "normal" (outra Ethernet). A parte do ChongDong existente localmente em cada máquina foi tornada segura usando os mecanismos de controlo de acesso

fornecidos pelo LIDS.

Foi ainda apresentada a concretização de um protocolo de consenso tolerante a intrusões que utiliza os serviços do ChongDong. O protocolo comporta-se de acordo com a especificação mesmo que ocorram intrusões em até um terço dos processos menos um.

Referências

[1] Fraga, J.S., Powell, D.: A fault- and intrusion-tolerant file system. In: Proceedings of the 3rd International Conference on Computer Security. (1985) 203–218

[2] Lala, J.H., ed.: Foundations of Intrusion Tolerant Systems. IEEE Computer Society Press (2003)

[3] Veríssimo, P., Neves, N.F., Correia, M.: Intrusion-tolerant architectures: Concepts and design. In Lemos, R., Gacek, C., Romanovsky, A., eds.: Architecting Dependable Systems. Volume 2677 of Lecture Notes in Computer Science. Springer-Verlag (2003) 3–36

[4] Adelsbach, A., Alessandri, D., Cachin, C., Creese, S., Deswarte, Y., Kursawe, K., Laprie, J.C., Powell, D., Randell, B., Riordan, J., Ryan, P., Simmonds, W., Stroud, R., Veríssimo, P., Waidner, M., Wespi, A.: Conceptual Model and Architecture of MAFTIA. Project MAFTIA deliverable D21. (2002)

[5] Lamport, L., Shostak, R., Pease, M.: The Byzantine generals problem. ACM Transactions on Programming Languages and Systems **4** (1982) 382–401

[6] Veríssimo, P.: Uncertainty and predictability: Can they be reconciled? In: Future Directions in Distributed Computing. Volume 2584 of Lecture Notes in Computer Science. Springer-Verlag (2003) 108–113

[7] Huagang, X.: Build a secure system with LIDS. http://www.lids.org/document/build_lids-0.2.html (2000)

[8] Correia, M., Neves, N.F., Veríssimo, P.: From consensus to atomic broadcast: Time-free byzantine-resistant protocols without signatures. DI/FCUL TR 05–14, Department of Informatics, University of Lisbon (2005)

[9] Correia, M.: Serviços distribuídos tolerantes a intrusões: resultados recentes e problemas abertos. In Gaspary, L.P., Siqueira, F., eds.: SBSeg 2005, V Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais, Livro Texto dos Minicursos. Sociedade Brasileira de Computação (2005) 113–162

[10] Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of distributed consensus with one faulty process. Journal of the ACM **32** (1985) 374–382

[11] IEEE: Draft Standard for Information Technology - Portable Operating System Interface (POSIX) - Amendment: Protection, Audit and Control Interfaces. P1003.1e. (1999) (withdrawn).

[12] Tobotras, B.: Linux Capabilities FAQ 0.2. http://www.kernel.org/pub/linux/libs/security/linux-privs/kernel-2.2/capfaq-0.2.txt (1999)

[13] Jaeger, T., Edwards, A., Zhang, X.: Consistency analysis of authorization hook placement in the Linux security modules framework. ACM Transactions on Information and System Security **7** (2004) 175–205

[14] Loscocco, P., Smalley, S.: Integrating flexible support for security policies into the Linux operating system. http://www.nsa.gov/selinux/slinux-abs.html (2001)

[15] Correia, M., Lung, L.C., Neves, N.F., Veríssimo, P.: Efficient Byzantine-resilient reliable multicast on a hybrid failure model. In: Proceedings of the 21st IEEE Symposium on Reliable Distributed Systems. (2002) 2–11

[16] Correia, M., Neves, N.F., Lung, L.C., Veríssimo, P.: Low complexity Byzantine-resilient consensus. Distributed Computing **17** (2005) 237–249

[17] Correia, M., Veríssimo, P., Neves, N.F.: The design of a COTS real-time distributed security kernel. In: Proceedings of the Fourth European Dependable Computing Conference. (2002) 234–252

[18] Correia, M., Neves, N.F., Veríssimo, P.: How to tolerate half less one Byzantine nodes in practical distributed systems. In: Proceedings of the 23rd IEEE Symposium on Reliable Distributed Systems. (2004) 174–183

[19] Sousa, P., Neves, N.F., Veríssimo, P.: Proactive resilience through architectural hybridization. DI/FCUL TR 05–8, Department of Informatics, University of Lisbon (2005)

[20] Veríssimo, P., Casimiro, A.: The Timely Computing Base model and architecture. IEEE Transactions on Computers **51** (2002) 916–930

[21] Ben-Or, M.: Another advantage of free choice: Completely asynchronous agreement protocols. In: Proceedings of the 2nd ACM Symposium on Principles of Distributed Computing. (1983) 27–30

[22] Dwork, C., Lynch, N., Stockmeyer, L.: Consensus in the presence of partial synchrony. Journal of the ACM **35** (1988) 288–323

[23] Malkhi, D., Reiter, M.: Unreliable intrusion detection in distributed computations. In: Proceedings of the 10th Computer Security Foundations Workshop. (1997) 116–124

[24] Koziol, J., Litchfield, D., Aitel, D., Anley, C., Eren, S., Mehta, N., Hassell, R.: The Shellcoder's Handbook. John Wiley & Sons (2004)

[25] Hatch, B.: An overview of LIDS. Infocus (2001) http://www.securityfocus.com/infocus/1496.

[26] Gasser, M.: Building a secure computer system. Van Nostrand Reinhold Co., New York, NY, USA (1988)

[27] Schiper, A.: Early consensus in an asynchronous system with a weak failure detector. Distributed Computing **10** (1997) 149–157

[28] Chandra, T., Toueg, S.: Unreliable failure detectors for reliable distributed systems. Journal of the ACM **43** (1996) 225–267

[29] Menezes, A.J., Oorschot, P.C.V., Vanstone, S.A.: Handbook of Applied Cryptography. CRC Press (1997)

[30] Viega, J., McGraw, G.: Building Secure Software. Addison Wesley (2002)

[31] Neves, N.F., Correia, M., Veríssimo, P.: Wormhole-aware Byzantine protocols. In: 2nd Bertinoro Workshop on Future Directions in Distributed Computing: Survivability - Obstacles and Solutions. (2004)

[32] Hickman, K.: The SSL protocol. Netscape Communications Corp. (1995)

[33] Bracha, G., Toueg, S.: Asynchronous consensus and broadcast protocols. Journal of the ACM **32** (1985) 824–840

[34] Malkhi, D., Reiter, M.: Secure and scalable replication in Phalanx. In: Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems. (1998)

[35] NIST: Announcement of weakness in the secure hash standard (1994)

[36] Reiter, M.: Secure agreement protocols: Reliable and atomic group multicast in Rampart. In: Proceedings of the 2nd ACM Conference on Computer and Communications Security. (1994) 68–80

[37] Kihlstrom, K.P., Moser, L.E., Melliar-Smith, P.M.: The SecureRing group communication system. ACM Transactions on Information and System Security **4** (2001) 371–406