FTP Tolerante a Intrusões[1]

José Pascoal[1], Tiago Jorge[1]
Miguel Correia[2], Nuno Ferreira Neves[2], Paulo Veríssimo[2]

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

{jose.pascoal.ext,tiago.jorge.ext}@siemens.com

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

{mpc,nuno,pjv}@di.fc.ul.pt

Resumo

A segurança de serviços distribuídos na Internet é uma preocupação constante dos administradores de sistemas. Uma abordagem recente denominada *tolerância a intrusões* pretende aplicar o paradigma da *tolerância a faltas* no domínio da *segurança*. O objectivo é o de procurar não apenas prevenir a ocorrência de intrusões, mas criar mecanismos que mantenham o sistema operacional mesmo que estas sucedam.

Este artigo apresenta o projecto de um serviço de FTP tolerante a intrusões. Este serviço utiliza um componente distribuído da classe dos *wormholes* para tolerar intrusões em alguns servidores. O número de servidores necessários é inferior ao de outros sistemas semelhantes na literatura, o que tem um importante impacto no custo da solução. O desempenho do serviço é medido.

1 Introdução

A segurança de *serviços distribuídos* da Internet como a Web, o correio electrónico ou o FTP é uma preocupação constante dos administradores de sistemas informáticos. Uma abordagem recente denominada *tolerância a intrusões* pretende aplicar mecanismos do domínio da *tolerância a faltas* em *segurança*, com o objectivo de construir sistemas mais confiáveis e seguros [1, 2, 3]. A ideia geral é a de que as intrusões, ataques e vulnerabilidades podem ser consideradas como sendo *faltas*, logo podem ser *toleradas* usando mecanismos de tolerância a faltas. Assim, pretende-se que o sistema permaneça operacional mesmo que ocorram algumas intrusões. O que a área tem de inovador é que estas faltas são particularmente perniciosas, tendo de ser englobadas na categoria de faltas mais geral: as faltas arbitrárias ou bizantinas.

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

No contexto específico dos serviços distribuídos, têm sido estudadas técnicas de tolerância a intrusões para garantir a integridade, disponibilidade e confidencialidade de dados e dos próprios serviços mesmo que alguns dos servidores sejam atacados e corrompidos por piratas informáticos ou código malicioso [4]. A ideia consiste em concretizar o serviço através de um conjunto de servidores, sendo a tarefa do atacante dificultada desde que esses servidores sejam diferentes ou, mais precisamente, que o sucesso em atacar um deles não reduza a dificuldade em atacar o seguinte [5, 6]. Isto exige, por exemplo, sistemas operativos diferentes e código diferente em cada servidor [7], além da necessidade de usar as técnicas de segurança clássicas como a colocação de remendos de segurança *(patches)*.

As principais técnicas que visam garantir a integridade e a disponibilidade dos serviços são a replicação de máquinas de estados [8, 9, 10] e os sistemas de quorums [11, 12]. Para garantir não só a integridade e a disponibilidade mas também a confidencialidade dos dados podem ser usadas técnicas como a partilha de segredos [13] ou *erasure codes* [14]. Todas estas técnicas têm um problema evidente: cada servidor adicional tem um custo elevado, já que exige o desenvolvimento de uma nova versão do código, provavelmente por uma nova equipa de programadores, para além dos custos de hardware e licenças de software.

Este artigo apresenta a primeira concretização de um serviço tolerante a intrusões com replicação de máquinas de estados baseado numa solução recentemente introduzida por alguns dos autores que permite *reduzir o número de réplicas* [15]. A solução precisa apenas de $n = 2f + 1$ réplicas/servidores para tolerar f intrusões, ou seja, 3 réplicas para tolerar uma intrusão, 5 para tolerar duas, etc. Os outros sistemas do mesmo tipo, o Rampart [9] e o BFT [10], precisam de um número de réplicas consideravelmente superior: $n = 3f + 1$ (4 réplicas para tolerar uma intrusão, 7 para tolerar duas, etc.).

A redução do número de réplicas é conseguida através de uma espécie de "oráculo" distribuído denominado *wormhole* [16]. O conceito de *wormhole* é muito abrangente e pretende incluir diversos tipos de componentes, locais ou distribuídos, que permitam lidar com algum tipo de *incerteza* num sistema distribuído. Neste caso a incerteza com que se pretende lidar é em termos de segurança. O *wormhole* apresentado no artigo tem como principal finalidade estabelecer uma ordenação para um conjunto de mensagens que, como veremos, é um dos problemas fundamentais da replicação de máquinas de estados. Este *wormhole* surge na sequência de outro que tinha o mesmo objectivo de suportar protocolos tolerantes a intrusões, como difusão fiável [17] e consenso [18], mas que assumia hipóteses temporais muito fortes (era tempo-real) e não suportava replicação de máquinas de estados com $2f + 1$ réplicas [19].

O artigo relata os desafios e dificuldades encontrados no projecto de um *serviço de FTP* [20] *tolerante a intrusões* e do novo *wormhole* que o suporta. Este componente distribuído, denominado WOO (*Wormhole Ordering Oracle*), é o primeiro *wormhole* baseado num núcleo de segurança, o Fiasco [21].

As contribuições do artigo são:

- apresenta o projecto de um *wormhole* seguro (WOO) baseado num núcleo de segurança (Fiasco);
- explica a concretização de um serviço de FTP tolerante a intrusões com um número de réplicas inferior ao usado por sistemas análogos na literatura;
- faz uma avaliação do desempenho do serviço desenvolvido.

2 Modelo e arquitectura do sistema

O sistema é composto por um conjunto de máquinas interligadas por uma rede. O modelo assumido para este sistema é o modelo assíncrono, ou seja, não são feitas quaisquer hipóteses sobre tempos de processamento ou de comunicação.

Esse sistema básico é extendido com o WOO. Este componente é distribuído, ou seja, tem código local executado num subconjunto das máquinas (WOOs locais) e um canal de comunicação privado. A arquitectura de um sistema com um WOO é apresentada na figura 1. Sobre o WOO fazemos duas hipóteses fundamentais:

- o WOO é seguro, não poder ser atacado com sucesso, pode apenas falhar por paragem;
- o WOO tem sincronia suficiente para concretizar um detector de falhas.

Logicamente que substanciar estas hipóteses, sobretudo a primeira, é fundamental para que o componente seja confiável *(trusted)* e o serviço nele baseado de confiança *(trustworthy)*. A hipótese só faz sentido devido à simplicidade do WOO que fornece um único serviço: o serviço de ordenação já mencionado acima. Logo, a sua interface é simples e passível de ser tornada segura. Este assunto é discutido na secção 3.

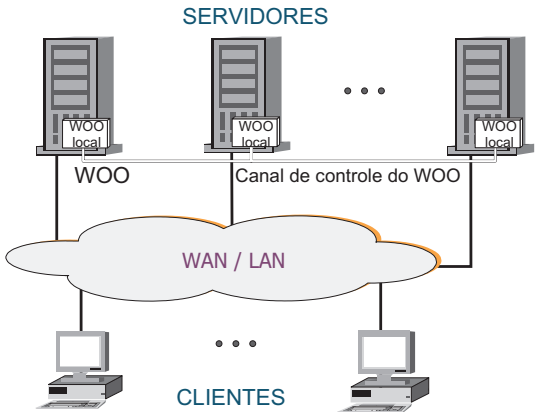

Figura 1: Arquitectura do sistema.

O serviço de FTP é concretizado por um conjunto de *servidores* $S = \{s_1, ..., s_n\}$ e acedido por um conjunto de clientes $C = \{c_1, ..., c_m\}$. O termo *processos* é usado para denominar globalmente os servidores e os clientes. Toda a máquina com um servidor inclui um WOO local, não se passando o mesmo com as dos clientes. Esta arquitectura é apresentada na figura 1.

Um processo diz-se *correcto* se executa o seu protocolo, caso contrário diz-se *falhado*. Um processo pode falhar por diversas razões, tanto acidentais como derivadas de uma intrusão. Alguns exemplos são paragem *(crash)*, não enviar uma mensagem que era suposto enviar, enviar essa mensagem mas com conteúdo modificado, ou enviar diversas mensagens com o mesmo identificador. Um processo falhado pode até fazer conluio com outros processos falhadas com o objectivo de corromper o funcionamento do serviço. No caso de um servidor, este é também considerado falhado se o seu WOO local parar. O artigo toma como hipótese que a maioria dos servidores não falham (ou seja, falham no máximo $f = \lfloor \frac{n-1}{2} \rfloor$), mas não assume qualquer limite para o número de clientes que falham.

2.1 Replicação de Máquinas de Estados

Uma das maneiras de construir serviços distribuídos tolerantes a faltas, e também *tolerantes a intrusões*, é utilizar o paradigma da *replicação de máquinas de estados* [8]. Este baseia-se em utilizar servidores replicados permitindo que mesmo que algumas réplicas falhem o serviço continue operacional.

Uma *máquina de estados* é caracterizada por um conjunto de *variáveis de estado*, que definem o estado da máquina, e um conjunto de *comandos*, que modificam as variáveis de estado. Os comandos têm de ser atómicos no sentido em que não podem interferir com outros comandos. O *paradigma da replicação de máquinas de estados* consiste na replicação de uma máquina de estados em n servidores $s_i \in S$. O conjunto S dos servidores concretiza o *serviço* que é utilizado pelos clientes.

O sistema funciona basicamente da seguinte maneira : (1) um cliente envia um comando para um dos servidores; (2) o servidor difunde o comando para todos os outros servidores usando um protocolo de difusão atómica que entrega todos os comandos pela mesma ordem aos servidores; (3) cada servidor executa o comando e envia a resposta para o cliente; (4) o cliente espera por $f+1$ respostas iguais de servidores diferentes; o resultado destas respostas é o resultado do comando enviado. É fácil perceber que se todos os servidores começarem com o mesmo estado inicial e os comandos forem deterministas (em todos os servidores actuarem sobre o estado da mesma forma), o que se assume ser verdade, todos vão seguir a mesma sequência de estados. Assim, mesmo que uma minoria deles seja maliciosa, vai ser sempre possível fazer uma votação e obter resultados correctos.

3 Concretização do WOO

O modelo de sistema usualmente considerado em trabalhos de tolerância a intrusões é homogéneo, tanto sob o ponto de vista das faltas como do tempo. Esses trabalhos consideram que todos os processos podem ser atacados e em todos podem ocorrer intrusões (mas não em mais do que f) e que não existem limites temporais para o processamento e comunicação (embora haja sempre algum tipo de hipóteses temporais). Neste artigo, como já referido, esse modelo é extendido com um componente distribuído e seguro, o WOO (*Wormhole Ordering Oracle*). Não são feitas quaisquer hipóteses temporais sobre o funcionamento do sistema "normal", apenas sobre o WOO é feita uma hipótese fraca. O WOO pode ser tornado seguro por ser relativamente simples, oferecendo apenas um serviço que será descrito mais à frente: o *Trusted Multicast Ordering service* (TMO).

A arquitectura de um sistema com um WOO já foi apresentada atrás na figura 1. Cada servidor contém um sistema operativo e um WOO local instalado e isolado do resto do sistema. A comunicação dos processos é realizada da forma normal, através de uma Ethernet, da Internet, etc. Quando um processo pretende utilizar o serviços do *wormhole*, inicia uma ligação com o WOO local e chama os serviços através de uma biblioteca. O WOO local de cada máquina comunica com todos os outros através de um canal isolado e protegido (canal de controle na figura).

3.1 Fiasco e L4Linux

O primeiro aspecto a considerar quando se pensa em concretizar um *wormhole* seguro é como vai ser protegido o módulo local, neste caso o WOO local. Neste artigo é utilizado um *micro-kernel* gratuito denominado Fiasco[2], que concretiza uma interface para *micro-kernels* chamada L4 [21][3]. Numa aplicação comercial seria necessário que o WOO local fosse protegido com um elevado grau de confiança, sendo portanto desejável uma solução em hardware, como a sua colocação num coprocessador seguro ou numa placa PC-104. No entanto, sob o ponto de vista de investigação é mais interessante concretizá-lo em software, já que assim é mais simples distribuir o código e permitir que outros grupos o testem. Essa é a principal motivação para a escolha de um *micro-kernel*. A motivação para a escolha específica do Fiasco foi este ter sido usado como núcleo de um outro núcleo de segurança denominado Perseus [22]. Inicialmente pensou-se em usar o próprio Perseus mas o código estava demasiado instável e as funcionalidades mais importantes já estavam disponíveis no Fiasco de forma que se optou por usar este último.

O WOO local é concretizado como um módulo do Fiasco. No contexto do trabalho em *micro-kernels* estes módulos são denominados "servidores", mas não passam de um processo que corre sobre o núcleo.

[2] O nome é alemão mas o significado idêntico ao português...
[3] http://os.inf.tu-dresden.de/fiasco/

O Fiasco retira ao sistema operativo o controle do hardware, incluindo a gestão de memória e interrupções. O sistema operativo usado foi o L4Linux[4], uma concretização do Linux sobre a interface L4. O L4Linux é executado em modo utilizador do Fiasco, logo o *superuser* não tem controle total do sistema e, concretamente, não pode interferir com o WOO local.

Pode-se considerar que o Fiasco observa as propriedades de um *núcleo de segurança* e que concretiza um *monitor de referência* [23]. É *completo* porque nenhum processo consegue contactar os servidores em execução sem passar pelo núcleo. É razoável assumir que é *isolado* e *verificável* dada a sua relativa simplicidade (cerca de 5000 linhas de código C++).

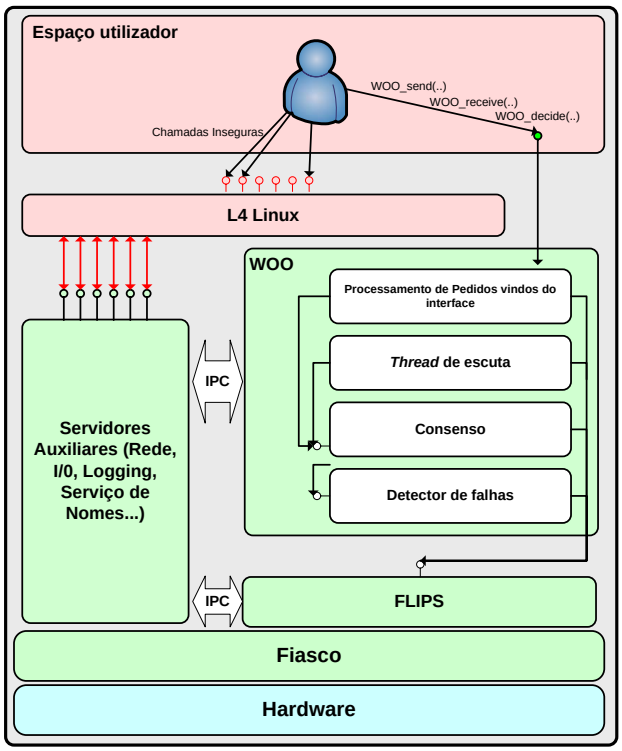

Figura 2: Concretização do *wormhole*

A figura 2 apresenta os componentes usados para concretizar o sistema. No topo está o processo executado em modo utilizador do L4Linux, que se encontra logo abaixo. Em baixo encontra-se o hardware e o núcleo Fiasco. Entre o Fiasco e o L4Linux estão diversos módulos ("servidores") que acrescentam funcionalidade ao Fiasco e que também assumimos serem seguros. A meio, do lado direito, encontra-se o WOO local.

A comunicação entre WOOs locais é sempre feita através do canal de co-

[4]http://os.inf.tu-dresden.de/L4/LinuxOnL4/

municação privado. O acesso a este canal através das máquinas onde existe um WOO local está reservado a esse componente, sendo inacessível ao L4Linux e, por maioria de razão, às aplicações que nele corram. Em relação à possível interacção directa de atacantes com a rede privada do WOO, simplesmente assume-se que esta está protegida fisicamente e que esse acesso é impossível.

3.2 Software de suporte

A concretização utiliza diversos "servidores" Fiasco e outro software de suporte que será descrito em seguida. A concretização exigiu a combinação de muitos componentes diferentes, alguns não documentados e outros ainda em fase de desenvolvimento. Essa foi claramente uma limitação encontrada na utilização de um *micro-kernel* gratuito.

O L4Env é um ambiente de programação para desenvolvimento de aplicações sobre *micro-kernels* da família L4 [24]. O objectivo do L4Env é definir um conjunto mínimo de funções que concretizem uma base comum disponível a todas as aplicações L4. É disponibilizado sob a forma de um pacote de servidores L4 que gerem recursos básicos como memória, tarefas e recursos de I/O.

A concretização do WOO utilizou exaustivamente as facilidades oferecidas por este ambiente, principalmente o *threading* de L4, *locks*, e um servidor de nomes, de modo a traduzir em nomes os identificadores únicos de *threads* de L4. A infraestrutura de rede foi garantida pelo FLIPS que consiste numa adaptação parcial do stack TCP/IP do Linux 2.4 para L4. Usa igualmente o l4vfs que concretiza um sistema virtual de ficheiros para que servidores L4 possam escrever e ler ficheiros em disco. O FLIPS e o l4vfs são projectos internos da Universidade de Dresden, que está a desenvolver o Fiasco, e para os quais não existe documentação disponível.

O *bootloader* é o primeiro software que é executado quando do arranque de um computador. A sua função é carregar e transferir o controle para o núcleo do sistema operativo, que por sua vez irá iniciar o resto do sistema. Neste trabalho, foi utilizado o GRUB[5], que estando em conformidade com a especificação de *multiboot*, permite carregar módulos adicionais para além do núcleo principal. Foi necessário usar o GRUB pois existia a necessidade de carregar não apenas o Fiasco mas também os servidores auxiliares.

A comunicação entre processos (IPC) é um dos mecanismos basilares oferecido pelo L4 e pelo Fiasco. No entanto, para concretizar um esquema de chamada a um método num servidor usando o IPC do L4/Fiasco, é preciso realizar diversos passos: o cliente tem de construir uma mensagem com o pedido; o servidor tem de esperar pelo pedido; quando recebe o pedido tem de invocar a respectiva função, construir a resposta e enviá-la. Para automatizar e facilitar esta tarefa, o Fiasco inclui um compilador de IDL (*Interface Description Language*). Este compilador, o DICE[6], gera *stubs* de comunicação IPC L4 a partir de uma descrição da interface dos métodos a chamar em DICE IDL.

[5] http://www.gnu.org/software/grub/
[6] http://os.inf.tu-dresden.de/dice/

3.3 Concretização do serviço TMO

O único serviço disponibilizado pelo WOO, o *Trusted Multicast Ordering service* (TMO), destina-se a atribuir números de ordem a mensagens. Quando um processo quer enviar uma mensagem com ordenação, entrega um resumo criptográfico da mesma ao WOO local que o vai disseminar pelos restantes WOOs locais. Entretanto, a mensagem em si é enviada pela rede normal. Quando a mensagem é recebida noutra máquina, o processo que a recebeu informa o seu WOO local de que a recebeu. Quando uma maioria dos processos confirmar a recepção da mensagem, é-lhe atribuído um número de ordem.

Esta secção explica como esse serviço é concretizado. Os processos interagem com o serviço através de uma biblioteca cujas principais funções se encontram na tabela 1. A biblioteca encapsula a comunicação dos processos com o WOO local, que é feita através de IPC L4. Os *stubs* que são a base das funções da biblioteca foram gerados usando o DICE.

Função	**Assinatura**
WOO Send	WOO_Send_retval **WOO_send**(eid, elist, threshold, msg_id, msg_hash)
WOO Receive	WOO_Receive_retval **WOO_receive**(eid, elist, threshold, msg_id, msg_hash, sender_eid)
WOO Decide	WOO_Decide_retval **WOO_decide**(tag)

Tabela 1: API do sistema

Em geral cada mensagem que se pretende ordenar causa uma execução do serviço TMO. Chamemos a essa execução "um TMO". Um TMO é representado internamente em cada WOO local através de um conjunto de informação: o *eid* do processo que o iniciou; um vector *elist* com os *eids* dos processos a quem a mensagem será entregue; o *threshold* de processos a partir do qual o número de ordem pode ser decidido; um identificador da mensagem; um resumo criptográfico (*hash*) da mensagem [25]; o número de ordem (se já existir); uma lista dos processo que já receberam a mensagem correspondente ao TMO. Cada WOO local contém três estruturas de dados principais: tabela de todos os TMO sem número de ordem atribuído; tabela de todos os TMO já terminados, ou seja, com número de ordem atribuído; e tabela dos *grupos de processos* conhecidos, ou seja, das variantes de *elist* que já chamaram o TMO.

A primeira função da interface é a *WOO_send* (tabela 1). É chamada pelo processo que *envia* a mensagem para iniciar um TMO. Quando a função é chamada, o WOO local realiza várias verificações dos parâmetros. O novo TMO é adicionado a uma tabela e é enviada internamente uma mensagem aos outros WOOs locais sinalizando o seu início. É devolvida uma *tag* que permite depois ao processo identificar esse TMO quando chamar *WOO_decide*.

A segunda função de interface é a *WOO_receive*. É chamada pelos membros do grupo que *recebem* a mensagem pela rede normal. Tal como no *WOO_send*, são verificadas várias opções de integridade e também se o *hash* é igual ao fornecido na chamada a *WOO_send*. Se estiver incorrecto, é devolvido um erro

e entregue a *tag* correcta desse TMO a quem invocou o método para posteriormente poder aceder ao resultado. Caso contrário, o WOO local regista a recepção da mensagem relativa ao TMO e difunde-se para todos os WOOs locais a ocorrência. Se esse TMO tiver atingido *threshold*−1 recepções (perfazendo *threshold* processos que têm a mensagem), é executado um protocolo de consenso entre todos os WOOs locais para acordar o seu número de ordem.

A terceira função é a *WOO_decide*. Esta é invocada quando qualquer dos processos envolvidos num TMO pretendem obter o número de ordem atribuído a uma mensagem. Tem como parâmetro de entrada a *tag* da TMO. É devolvido ao processo o número de ordem e o *hash* da TMO.

A funcionalidade que referimos é executada essencialmente por duas *threads*. A primeira realiza a interface descrita acima. A segunda faz a recepção de mensagens vindas de outros WOOs locais.

Existem ainda mais duas *threads*. Como referido acima, a decisão sobre o número da mensagem é realizada através de um protocolo de consenso. O consenso é um problema importante em sistemas distribuídos [26, 27, 18]. O objectivo consiste em escolher um dos valores propostos por um de vários processos distribuídos. Apesar de ser simples de formular, não existe solução determinista para o problema em sistemas puramente assíncronos [28]. Este resultado tem sido contornado de diversas formas, incluindo os detectores de falhas não fiáveis [26, 27].

Neste artigo toda a sincronia necessária para resolver o consenso é encapsulada num detector de falhas executado dentro do WOO. Cada WOO local tem uma *thread* que periodicamente envia um *heartbeat* aos outros locais indicando que o WOO está operacional. Quando um WOO local não recebe durante algum tempo um *heartbeat* vindo de outro WOO local, coloca-o na lista dos suspeitos de terem falhado. Se depois disso recebe um *heartbeat* vindo desse WOO local, retira-o dessa lista. Assim, o detector de falhas pode enganar-se – não é fiável – mas o protocolo de consenso não é afectado por esses erros.

O protocolo de consenso usado é relativamente simples pois tem apenas de tolerar paragens de WOOs, não intrusões, já que é executado pelos WOOs locais comunicando através da sua rede privada. O protocolo usado foi o *early-consensus* de A. Schiper [27], que é uma versão melhorada do protocolo original com detectores de falhas de Chandra e Toueg [26]. O protocolo é executado por uma *thread* criada quando o WOO local arranca. Esta *thread* executa apenas um consenso de cada vez. O consenso serve para decidir quais são as mensagens a ordenar. Como são executados de forma ordenada, um de cada vez, a atribuição de números de ordem às mensagens é feita sequencialmente, da mesma forma por todos os WOOs locais.

3.4 Segurança do WOO

A segurança do WOO é baseada fundamentalmente na realização do WOO local como um "servidor" do núcleo Fiasco e da protecção física da rede privada do WOO (v. secção 3.1). No entanto, o WOO pode ser também atacado através da sua interface, o que exige algumas medidas adicionais.

A primeira dessas medidas é a concretização de um mecanismo de *controle de admissão*. A função deste mecanismo consiste em evitar ataques de negação de serviço.

Cada WOO Local é configurado com um parâmetro que define a sua capacidade de executar TMOs por hora (RT). Cada vez que um processo se liga ao WOO para começar a usar o serviço TMO, indica qual o intervalo de tempo mínimo entre TMOs que pretende realizar (TS, em segundos). Então, o WOO local calcula o número máximo de TMOs por hora que esse elemento vai ocupar, $3600 * TS$. Se este valor exceder a capacidade do *wormhole*, isto é se $RT - 3600 * RT - UT < 0$, sendo UT os TMOs/hora já reservados por outro(s) processo(s), o pedido é rejeitado. Caso contrário esses recursos são reservados para esse processo. Cada vez que chamar o WOO_send, será analisado o tempo que decorreu desde a última chamada. Se for inferior a RT, será retornado um erro indicando que o processo está a exceder os recursos reservados.

Muitos ataques perpetuados através da interface de um componente de software procuram explorar defeitos no código para inserir código malicioso ou de alguma forma interferir com o seu funcionamento. Um tipo de ferramenta adequado para resolver este tipo de problemas durante a fase de desenvolvimento é a análise estática de código [29].

No caso do WOO, após a concretização do código-fonte e do teste da sua funcionalidade, foi realizada uma análise estática do código utilizando duas ferramentas: Flawfinder e RATS. O Flawfinder[7] é um programa em Python, desenvolvido por David Wheeler, utilizado para realizar análise estática de código desenvolvido em C/C++. Funciona através do uso de uma base de dados de funções da biblioteca C/C++ com problemas bem conhecidos, tais como funções passíveis de criar vulnerabilidades de *buffer overflow*. O RATS (*Rough Auditing Tool for Security*)[8] é igualmente uma ferramenta *open-source* para analisar código. Pode ser utilizada sobre código C, C++, Perl, PHP e Python e tem um funcionamento semelhante ao Flawfinder. Analisa possíveis pontos de falha, gerando um relatório final com recomendações de como corrigir os erros.

A análise do código do WOO usando o Flawfinder gerou cerca de 400 linhas de output (v. excerto na figura 3). A cada potencial vulnerabilidade foi atribuído um nível de risco entre 1 (mínimo) e 5 (máximo). A zona do código que gerou cada alarme foi analisada manualmente para verificar se continha ou não uma vulnerabilidade. Os alarmes gerados podem ser agrupados e resumidos nas seguintes categorias:

- *sscanf* - a primeira vulnerabilidade de nível 4 detectada; neste contexto não apresenta risco, devido a não existir input externo na construção da string de envio;

- *strcat* - vulnerabilidade de nível 4; apesar de não existirem inputs externos, esta função foi modificada para *strncat*;

[7] http://www.dwheeler.com/flawfinder/
[8] http:/www.securesoftware.com/resources/download_rats.html

```
Flawfinder version 1.24, (C) 2001-2003 David A. Wheeler.
Number of dangerous functions in C/C++ ruleset: 128
...
./WOO_Packet_TMO_Receive.cc:96:  [4] (buffer) sscanf:
  The scanf() family's %s operation, without a limit specification,
  permits buffer overflows. Specify a limit to \%s, or use a different input
  function.
./WOO_TMO.cc:97:  [4] (buffer) strcat:
  Does not check for buffer overflows when concatenating to destination.
  Consider using strncat or strlcat (warning, strncat is easily misused).
./WOO_TMO.cc:101:  [4] (buffer) strcat:
  Does not check for buffer overflows when concatenating to destination.
  Consider using strncat or strlcat (warning, strncat is easily misused).
...
```

Figura 3: Um excerto do output do Flawfinder.

- *memcpy* - vulnerabilidades de nível 2; todas as invocações deste método são realizadas sobre *buffers* criados pelo programador ou na serizalização e desserialização de parâmetros de pacotes de rede, logo estão controladas e todos os limites são verificados;
- vectores de tamanho fixo - vulnerabilidades de nível 2; este tipo de buffers consistem em memória alocada para enviar pela rede e apenas preenchida pelos métodos de *memcpy* do item anterior; podem também ser tabelas internas que não recebem inputs exteriores e que são controladas por gestores de vectores; não existe vulnerabilidade;
- funções de uso de strings - vulnerabilidades de nível 1; são avisos relativos a sistemas mais antigos em que o parâmetro de limite no *strncpy* não era observado, não sendo este o caso.

O RATS considera apenas 3 níveis de risco. Detectou as mesmas vulnerabilidades do Flawfinder, tendo considerado que os buffers de tamanho fixo, o *sscanf* e o *strcat* são os que representam maior risco. Classificou o *memcpy* como sendo uma vulnerabilidade de um nível inferior. Denota-se menor nível de pormenor no *output* comparativamente ao do Flawfinder.

4 O serviço de FTP

Como foi referido na introdução, o artigo apresenta a concretização de um serviço de FTP distribuído tolerante a intrusões. O serviço é baseado em replicação de máquinas de estados, já usada por exemplo para concretizar um serviço de NFS tolerante a intrusões, se bem que usando um modelo e hipóteses diferentes [10]. O componente cuja concretização se descreveu na secção anterior não é propriamente parte do serviço de FTP. O objectivo do WOO é o de

suportar genericamente a concretização de serviços desse tipo, quaisquer que eles sejam. Assim, o serviço de FTP usa-o como um suporte à execução, como usa também chamadas ao sistema operativo e bibliotecas de software.

O sistema é constituído por três tipos de componentes: clientes $c_i \in C$, servidores $s_i \in S$ e um serviço de localização. O serviço de FTP é realizado pelo conjunto dos servidores. Assume-se que não mais do que f servidores falham de um total de $|S| \geq 2f + 1$, ou seja, uma maioria dos servidores tem de permanecer correcta para o sistema se comportar como especificado. Não há limite para o número de clientes que podem ser maliciosos.

A arquitectura do sistema encontra-se esboçada na figura 1, excepto o serviço de localização que será explicado em seguida.

4.1 Serviço de localização

O serviço de localização tem como objectivo fornecer as localizações de todos os servidores que concretizam o serviço de FTP. Para tal tem uma lista de localizações dos servidores, que vai sendo construída adicionando a informação de cada servidor conforme estes se vão ligando. A localização é o endereço de nível transporte, ou seja, um par {*endereço IP, porto*}.

Quando um servidor começa a sua execução, estabelece uma conexão SSL [30] com o serviço de localização. Os dois autenticam-se mutuamente usando o próprio SSL (os certificados são distribuídos previamente). Depois, o servidor envia ao serviço de localização uma mensagem com o formato:

$$\langle \text{SERVIDOR, id} \rangle$$

sendo SERVIDOR uma constante que identifica o tipo do emissor e *id* o seu IP e porto. Ao receber o pedido, o serviço de localização envia-lhe a lista de servidores activos, acrescenta o *id* a essa lista e fecha a conexão SSL.

Um cliente pode usar o serviço de localização quase da mesma forma para obter a localização dos servidores. A principal diferença é que o serviço não autentica o cliente, só o cliente autentica o serviço. Além disso, a constante usada é CLIENTE e o serviço não acrescenta o *id* dos clientes à lista dos servidores activos.

Todas as conexões referidas acima são canais seguros SSL. Para obter estes canais foi usada a biblioteca OpenSSL. Ao estabelecer uma conexão SSL é necessário os dois processos negociarem qual o modo de operação, ou seja, o tipo de autenticação, o algoritmo de cifra e a síntese criptográfica a utilizar para proteger a comunicação. Neste trabalho não é necessário assegurar a confidencialidade da comunicação, logo esta não é cifrada. Assim sendo, foi utilizado o modo de operação NULL-SHA do OpenSSL. Este modo usa o algoritmo RSA [31] para fazer autenticação e como função de síntese no cálculo dos MACs (*Message Authentication Codes* [25]) usa o algoritmo SHA-1 [32].

Os certificados utilizados na autenticação foram emitidos por uma pseudo-Autoridade de Certificação (CA). Um certificado contém uma associação entre um nome (e possivelmente outra informação) e a chave pública do serviço ou

servidor, assinada com a chave privada da CA. A pseudo-CA não passa de um pequeno programa que assina estes certificados. Assume-se que todos os servidores e clientes têm certificado com as chaves públicas do serviço de localização (assinado pela CA) e da CA (auto-assinado), pois necessitam delas para verificar a assinatura dos certificados que receberem. É também assumido que apenas servidores legítimos podem obter certificados assinados pela CA. Estas hipóteses têm de ser garantidas de forma administrativa.

A concretização actual do serviço de localização é ainda preliminar: não é tolerante a intrusões, simplesmente assume-se que é seguro e que a lista de servidores não pode ser alterada por um atacante. No entanto, uma versão tolerante a intrusões poderia ser facilmente concretizada usando sistemas de quorums tolerantes a intrusões [33].

4.2 Servidores

Os servidores essencialmente recebem os pedidos dos clientes, processam-os e enviam-lhes respostas. Os pedidos contêm comandos FTP (v. secção 4). Antes de poder desempenhar estas funções, cada servidor tem de passar por uma fase de inicialização.

A fase de inicialização passa por estabelecer uma conexão SSL com o serviço de localização, enviando a este o seu endereço, de modo a obter a lista de servidores online e a ser posteriormente adicionado a esta. Após receber esta lista, se esta não estiver vazia, o servidor vai estabelecer uma conexão SSL com cada elemento da mesma. Para o efeito vai enviar a cada elemento da lista uma mensagem da forma:

$$\langle \text{SERVIDOR, id, eid} \rangle$$

sendo SERVIDOR uma constante que identifica o tipo do emissor, *id* o seu endereço IP e porto, e *eid* o seu identificador perante o WOO (secção 3.3). Quando outro servidor recebe esta mensagem, guarda-a conjuntamente com a estrutura do OpenSSL que define a conexão entre os dois. Seguidamente envia uma estrutura com os seus dados que recebe o mesmo tratamento.

O servidor é constituído por duas *threads*. As duas partilham algumas variáveis, sobretudo três "sacos" onde são guardadas as mensagens com pedidos dos clientes. Estes contêm mensagens em três estados diferentes de processamento relacionados com a execução do serviço TMO do WOO que realiza a sua ordenação: mensagens que ainda não foram passadas ao TMO; mensagens que estão a aguardar que o TMO termine; e mensagens prontas a serem entregues (TMO já terminou).

A *thread* principal fica à escuta de novas ligações de clientes. As mensagens com pedidos provenientes dos clientes são da seguinte forma:

$$\langle \text{PEDIDO, id, num, cmd, vec} \rangle$$

onde PEDIDO é o tipo de pedido, *id* representa a identificação do emissor (endereço IP e porto), *num* é o número do pedido, *cmd* é o comando (e seus ar-

gumentos) que o servidor deve executar, e *vec* é um vector de MACs (o objectivo é explicado na próxima secção).

Quando um servidor recebe um pedido de um cliente, encapsula-o numa mensagem com o seguinte formato:

⟨ACAST, id, mreq, msg_id, sender_eid, elist, threshold⟩

onde *ACAST* é o tipo da mensagem, *id* representa o IP/porto do emissor, *mreq* é o pedido em si, *msg_id* é um número de mensagem, *sender_eid* representa o *eid* do servidor, *elist* é a lista dos *eid* dos processos envolvidos no protocolo, e *threshold* é o valor $\lfloor \frac{n-1}{2} \rfloor + 1$ (para $n = 2f + 1$, é $f + 1$, ou seja metade dos servidores mais um). O f referido anteriormente representa o número de processos que podem falhar. Depois de criar a nova mensagem, guarda-a no primeiro saco referido atrás e difunde-a para os outros servidores.

A outra *thread* serve para processar as mensagens recebidas e determinar a ordem pela qual estas vão ser entregues. É esta *thread* que interage com o WOO de modo a obter a ordem das mensagens. Para isso, tem de propor cada uma das mensagens ao WOO, aguardar o resultado e processar os pedidos conforme a ordem que este determinar. As respostas aos clientes são da forma:

⟨RESPOSTA, id, num, res⟩

onde RESPOSTA é o tipo da mensagem, *id* representa a identificação do emissor (endereço IP e porto, no formato IP:porto), *num* é o número do pedido e *res* é o resultado da execução do comando.

Uma descrição mais pormenorizada do protocolo executado pelos servidores encontra-se em [15].

4.3 Clientes

O cliente é o componente de *software* que vai utilizar os serviços fornecidos pelos servidores. Os clientes enviam comandos aos servidores e aguardam as respostas com os resultados.

Antes de poder enviar mensagens aos servidores, o cliente tem de passar pela fase de inicialização descrita acima. Após esta fase, o envio dos comandos é feito da seguinte maneira:

- o cliente envia o comando a um dos servidores;
- espera as respostas de $f + 1$ servidores ($f + 1$ porque se o sistema tolera f faltas, recebendo $f + 1$, é garantido que pelo menos uma é correcta);
- se não receber as respostas após um período de tempo T_{max}, reenvia o comando para f servidores; assim garante-se que $f+1$ servidores recebem o pedido, logo pelo menos um não é malicioso e inicia a ordenação desse pedido.

Quando o cliente envia um comando ao servidor, envia-lhe também um vector de MACs do pedido. Um MAC é uma soma de controle criptográfica obtida através de uma função de síntese criptográfica (*hash*) e uma chave secreta. O vector contém tantos MACs quantos os servidores (n). Cada MAC é calculado usando uma chave secreta partilhada entre o cliente e o servidor correspondente. O vector serve para impedir que um servidor malicioso corrompa um pedido de um cliente sem que a modificação seja detectada pelos servidores correctos. Quando um servidor recebe uma mensagem, calcula um novo MAC desta usando a chave partilhada entre ele e o cliente, e compara-o com o MAC que lhe corresponde contido no vector. Se estes forem iguais, a mensagem é legítima e o servidor processa-a. Caso contrário descarta-a. As chaves simétricas partilhadas são distribuídas inicialmente quando o cliente estabelece uma conexão SSL com cada servidor, e são periodicamente refrescadas. A função utilizada para calcular os MACs é a concretização do algoritmo HMAC do OpenSSL[9].

A concretização actual do cliente é constituída por várias *threads*. A *thread* principal tem como função processar os comandos pedidos pelo utilizador e enviá-los ao servidor. As restantes *threads*, uma por servidor, recebem e processam as respostas devolvidas pelos servidores. Estas solução não é necessariamente a mais eficiente, mas o desempenho do cliente de FTP não é geralmente muito crítico.

4.4 Serviço de FTP

O serviço de FTP simplificado que foi concretizado é baseado no sistema cliente servidor que se acabou de descrever. O serviço permite fornece apenas três comandos: listagem dos ficheiros contidos no servidor; carregamento de um ficheiro no servidor; e descarregamento de ficheiros do servidor. Estes comandos são colocados nas mensagens PEDIDO enviadas pelo cliente.

Para fazer o carregamento de um ficheiro para o servidor foi definido o comando *stor* (análogo ao comando com o mesmo nome na definição original do FTP [20]). Este comando tem como argumento o nome do ficheiro a enviar para o servidor. Quando este comando é executado, o cliente verifica se o ficheiro existe e, caso exista, envia-o em blocos de 16000 bytes para o servidor. A razão para esta fragmentação é que o OpenSSL apenas consegue processar blocos de 16384 bytes de cada vez. Cada bloco é enviado ao servidor dentro de um pacote com a seguinte forma:

$$\langle \text{PEDIDO, id, num, cmd, vec, file\_part, tam\_part, size\_fich} \rangle$$

onde *file_part* representa um bloco do ficheiro, *tam_part* é o tamanho do bloco enviado e *size_fich* é o tamanho total do ficheiro. O significado dos cinco primeiros campos já foi descrito anteriormente. Caso o servidor envie um erro num dos blocos, o envio do ficheiro é cancelado. Como resposta a este pedido o servidor envia um pacote RESPOSTA que será explicado mais adiante.

[9] Este esquema baseado em vectores de MACs podia ser substituído por assinaturas baseadas em criptografia de chave pública. No entanto, o desempenho obtido usando MACs é geralmente muito superior [10].

O comando a usar para fazer descarregamento de um ficheiro contido no servidor é o *retr*. Tal como o *stor*, também este tem como argumento o nome do ficheiro pretendido. O pacote enviado ao servidor é o mesmo mas não tem os últimos três campos preenchidos e o campo *cmd* contém o comando RETR em vez de conter STOR.

O servidor responde com um pacote com o seguinte formato:

⟨RESPOSTA, id, num, rpl, tam_dados, file_name, res_part, tam_part, n_part, erro⟩

onde *rpl* representa o tipo da resposta (neste caso RETR), *tam_dados* é o tamanho total do ficheiro, *file_name* representa o nome do ficheiro em questão, *res_part* é um bloco do ficheiro a enviar, *tam_part* é o tamanho do bloco enviado, *n_part* é o número do bloco enviado e, finalmente, *erro* representa um código de erro na execução do comando. Conforme as respostas vão chegando, o cliente vai construindo o ficheiro bloco a bloco, escrevendo cada um no sítio certo com base no campo *n_part*.

O comando criado para obter a listagem de ficheiros contidos no servidor é *list*. Este comando não tem qualquer argumento. É igualmente enviado para o servidor num pacote semelhante aos anteriores. Como se pode depreender, o protótipo não concretiza o protocolo FTP completo mas apenas um subconjunto. A funcionalidade não realizada mais importante é a gestão de directorias.

5 Desempenho do sistema

Esta secção apresenta as medidas de desempenho do sistema. As experiências realizadas envolveram três servidores ($|S| = 3$) instalados em PCs com processadores Intel Pentium III a 500 Mhz, e 256MB SDRam PC133. Cada PC tinha dois adaptadores de rede 3Com 10/100. A rede normal e o canal de controle do WOO eram duas redes Fast-Ethernet *switched* a 100Mbps. O cliente foi executado num PC com as mesmas características.

Foram realizadas duas experiências a fim de obter os valores médios da latência (tempo entre o envio do pedido e a obtenção da resposta) e do débito de comandos (número de comandos processados por unidade de tempo). A rede normal não tinha qualquer tráfego para além do gerado pelo sistema.

Uma das experiências teve como objectivo medir a latência e o débito do descarregamento de ficheiros. Para tal, foram feitos pedidos de vários ficheiros com tamanhos entre 1 e 10000 bytes. Os resultados desta experiência podem ser vistos na figura 4. Analisando estes resultados, pode-se constatar que a latência aumenta quase linearmente com o tamanho do ficheiro. Como seria de esperar, o débito é inversamente proporcional ao tamanho do ficheiro, diminuindo com o aumento do mesmo.

A outra experiência serviu para obter valores de latência e débito do carregamento de ficheiros. Como na experiência anterior, foram também enviados ficheiros com tamanhos compreendidos entre 1 e 10000 bytes. Mais uma vez, a

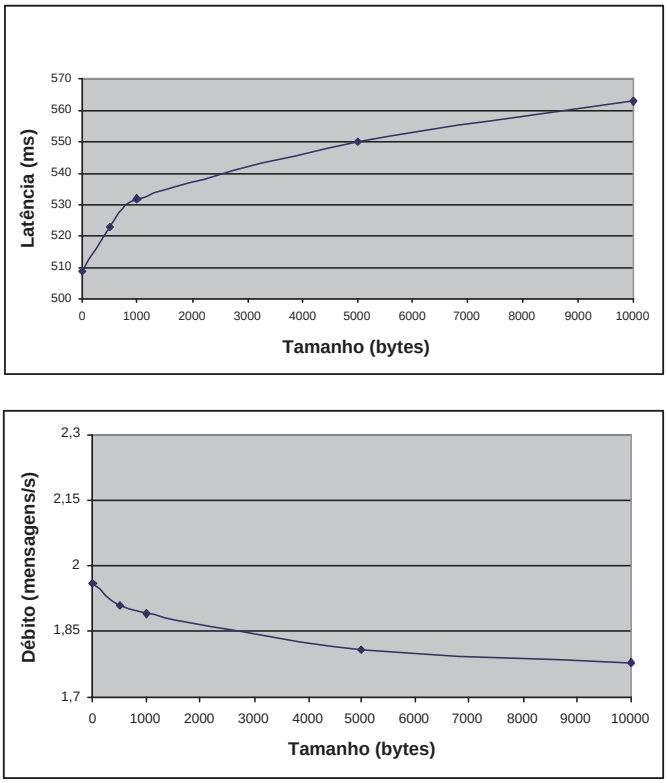

Figura 4: Latência e débito do descarregamento

latência aumentou com o tamanho dos ficheiros enquanto que o débito diminuiu. Isto pode ser visto na figura 5.

O uso de três servidores permite tolerar um servidor malicioso. Foram feitas algumas experiências com um servidor malicioso que simplesmente não respondia a nenhum pedido. O desempenho do sistema não foi afectado de forma sensível, excepto no caso em que o cliente enviava o pedido em primeiro lugar para esse servidor. Nesse caso a latência era afectada pois o pedido só era processado quando o cliente reenviava o pedido para outro servidor, ao fim de um *timeout*. Foram também feitas experiências com um servidor malicioso que corrompia todas as mensagens que enviava. O desempenho do sistema era afectado de forma semelhante ao caso do servidor silencioso. Em nenhum dos casos a correcção dos resultados do serviço foi afectada.

Foi ainda feita uma experiência baseada numa versão do sistema na qual as chamadas ao WOO foram substituídas por *dummies*, funções que não realizam a funcionalidade prevista. Esta versão do sistema logicamente não concretizava a semântica do sistema. Foi usada para comparar o desempenho do sistema quando executado sobre L4Linux/Fiasco e sobre Linux. O carregamento de um

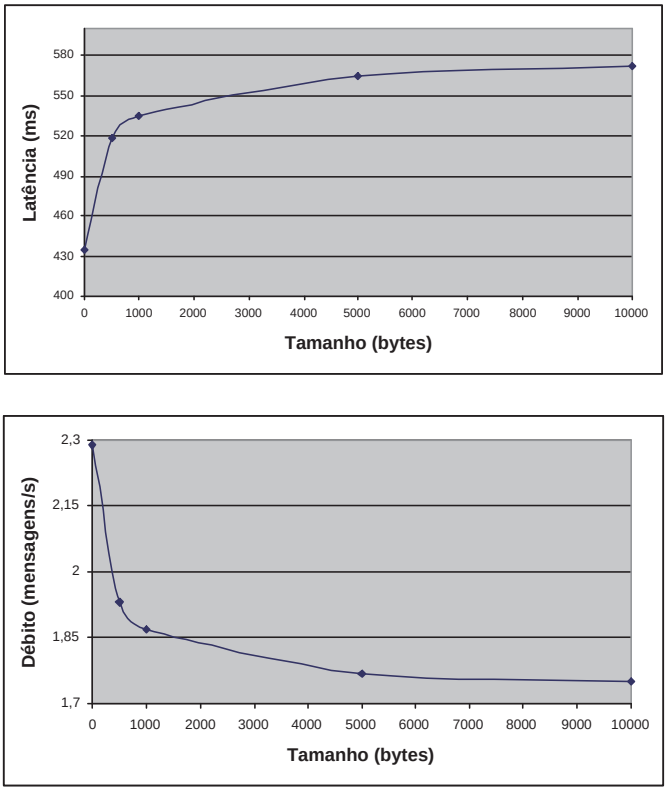

Figura 5: Latência e débito do carregamento

ficheiro pequeno que demorava cerca de 450 ms sobre L4Linux/Fiasco, demorava apenas 45 ms sobre Linux. A razão para essa quebra de desempenho de 10 vezes deve-se ao L4Linux. Este, tal como já referido anteriormente, é uma adaptação do núcleo do Linux de modo a funcionar em modo-utilizador sobre um *micro-kernel* da família L4, como o Fiasco. Isto é conseguido substituindo as operações que o Linux realiza directamente sobre o hardware pela invocação de funções da interface do *micro-kernel* que emulam o hardware. Esta emulação é que causa a forte degradação do desempenho do sistema.

6 Conclusão

O artigo descreveu a concretização de um *serviço de FTP tolerante a intrusões* baseado em replicação de máquina de estados. Este serviço precisa apenas de $2f + 1$ réplicas para tolerar f faltas ao contrário das $3f + 1$ necessárias em sistemas anteriores. Esta redução foi possível recorrendo a um *wormhole* denominado WOO que oferece um serviço de ordenação de mensagens, o TMO.

Foram descritos os componentes deste serviço bem como as suas funções. A concretização do WOO usando um *micro-kernel* chamado Fiasco foi apresentada detalhadamente e algumas concretizações alternativas brevemente discutidas. Foram apresentadas e discutidas algumas medidas de desempenho.

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] 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

[3] Avizienis, A., Laprie, J.C., Randell, B., Landwehr, C.: Basic concepts and taxonomy of dependable and secure computing. IEEE Transactions on Dependable and Secure Computing **1** (2004) 11–33

[4] 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

[5] Deswarte, Y., Kanoun, K., Laprie, J.C.: Diversity against accidental and deliberate faults. In: Computer Security, Dependability & Assurance: From Needs to Solutions. IEEE Press (1998)

[6] Obelheiro, R.R., Bessani, A.N., Fraga, J.S., Lung, L.C.: Analisando a viabilidade da implementação prática de sistemas tolerantes a intrusões. In: Anais do 5o Simpósio Brasileiro de Segurança. (2005)

[7] Avizienis, A.: The N-version approach to fault tolerant software. IEEE Transactions on Software Engineering **11** (1985) 1491–1501

[8] Schneider, F.B.: Implementing faul-tolerant services using the state machine approach: A tutorial. ACM Computing Surveys **22** (1990) 299–319

[9] Reiter, M.K.: The Rampart toolkit for building high-integrity services. In: Theory and Practice in Distributed Systems. Volume 938 of Lecture Notes in Computer Science. Springer-Verlag (1995) 99–110

[10] Castro, M., Liskov, B.: Practical Byzantine fault tolerance and proactive recovery. ACM Transactions on Computer Systems **20** (2002) 398–461

[11] Malkhi, D., Reiter, M.: Byzantine quorum systems. Distributed Computing **11** (1998) 203–213

[12] Martin, J.P., Alvisi, L., Dahlin, M.: Minimal Byzantine storage. In: Proceedings of the 16th International Conference on Distributed Computing. Volume 2508 of LNCS., Springer-Verlag (2002) 311–325

[13] Lakshmanan, S., Ahamad, M., Venkateswaran, H.: Responsive security for stored data. IEEE Transactions on Parallel and Distributed Systems **14** (2003) 818–828

[14] Cachin, C., Tessaro, S.: Asynchronous verifiable information dispersal. RZ 3569, IBM Research (2004)

[15] 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

[16] 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

[17] 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

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

[19] 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

[20] Postel, J., Reynolds, J.: File transport protocol (FTP). IETF Request for Comments: RFC 959 (1985)

[21] Hohmuth, M.: The fiasco kernel: Requirements definition. Technical report, Dresden University of Technology (1998)

[22] Stueble, C.: Development of a prototype for a security platform for mobile devices. Master's thesis, Universitat des Saarlandes (2000)

[23] Gasser, M.: Building a Secure Computer System. Van Nostrand Reinhold (1988)

[24] Group, O.S.R.: L4env, an environment for l4 applications. Technical report, University of Dresden (2003) http://os.inf.tu-dresden.de/l4env/doc/l4env-concept/l4env.pdf.

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

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

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

[28] 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

[29] Evans, D., Larochelle, D.: Improving security using extensible lightweight static analysis. IEEE Software **19** (2002) 42–51

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

[31] Rivest, R.L., Shamir, A., Adleman, L.M.: A method for obtaining digital signatures and public-key cryptosystems. Communications of the ACM **21** (1978) 120–126

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

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