UNIVERSIDADE FEDERAL DE SANTA CATARINA
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA

Fábio Resner

EXECUÇÃO SEGURA DE CÓDIGO EM MÓDULO DE SEGURANÇA CRIPTOGRÁFICO

Florianópolis

2012

Fábio Resner

EXECUÇÃO SEGURA DE CÓDIGO EM MÓDULO DE SEGURANÇA CRIPTOGRÁFICO

Trabalho de conclusão de curso submetido ao Curso de Bacharelado em Ciências da Computação para a obtenção do Grau de Bacharel em Ciências da Computação.
Orientador: Olinto José Varella Furtado, Dr.
Coorientador: Ricardo Felipe Custódio, Dr.

Florianópolis

2012

Catalogação na fonte elaborada pela biblioteca da
Universidade Federal de Santa Catarina

A ficha catalográfica é confeccionada pela Biblioteca Central.

Tamanho: 7cm x 12 cm

Fonte: Times New Roman 9,5

Maiores informações em:
http://www.bu.ufsc.br/design/Catalogacao.html

Fábio Resner

EXECUÇÃO SEGURA DE CÓDIGO EM MÓDULO DE SEGURANÇA CRIPTOGRÁFICO

Este Trabalho de conclusão de curso foi julgado aprovado para a obtenção do Título de "Bacharel em Ciências da Computação", e aprovado em sua forma final pelo Curso de Bacharelado em Ciências da Computação.

Florianópolis, 20 de novembro 2012.

Vitório Bruno Mazolla
Coordenador

Olinto José Varella Furtado, Dr.
Orientador

Banca Examinadora:

Presidente da banca
Presidente

Ricardo Felipe Custódio, Dr.
Coorientador

Jean Martina, Dr.

André Bereza Junior

A minha mãe, meu pai e meu irmão, que são a razão de eu ter chegado até aqui.

AGRADECIMENTOS

A Deus, que sem eu saber porque, me cercou de tanto suporte e pessoas boas que me fizeram superar as dificuldades, a minha família por sempre ter proporcionado todo o suporte, apoio, educação e exemplo necessários, aos amigos do PNEQP, que me ajudaram em inúmeras e longas madrugadas de esforço, ao LabSEC e todos os meus companheiros, aos professores que fizeram parte da minha formação e principalmente aqueles que fizeram parte de forma mais próxima desta etapa final, a todos os meus verdadeiros amigos de coração, que me acompanharam durante toda essa longa caminhada, o meu muito obrigado!

$$0x2B \mid \sim 0x2B.$$

Hammlet

RESUMO

Este trabalho se propõe a pesquisar na literatura e em trabalhos correlatos, meios para garantir a execução segura de código e aplicá-los no contexto de um Módulo de segurança criptográfica. A execução segura de código é um problema em aberto, e é crucial para o não vazamento de informações sigilosas que comprometem sistemas de todo o tipo, e por consequência, seus usuários, sejam eles pessoas ou entidades. O grande problema em garantir a execução segura, se dá pela enorme complexidade que os sistemas computacionais atingiram, tornando muito difícil provar ou prevêr todos os possíveis comportamentos e falhas nos mesmos. A escolha de aplicar estas técnicas em um módulo de segurança criptográfica é que este não é um sistema de propósito geral e possui requisitos de performance muito menores do que estes, tornando possível abrir mão do desempenho para atingir um nível de segurança maior. Como objetivo, temos por delimitar os problemas enfrentados nas tentativas de implementar um mecanismo que garanta a execução segura, estudar as técnicas existentes e propôr soluções para que se alcance o resultado esperado. O problema de executar código de forma segura pode ser visto sobre vários aspectos, (KLEIN et al., 2009), (SEKAR et al., 2003), (DHURJATI; KOWSHIK; ADVE, 2006), são alguns trabalhos que propõe soluções. Este estudo pretende assim apresentar diferentes técnicas que vem sendo aplicadas na tentativa de executar código de forma segura, mostrando seus pontos forte e fracos, e propôr um plano de trabalho para o desenvolvimento de um mecanismo passível de ser implementado no módulo de segurança criptográfico alvo, o ASI-HSM.

Palavras-chave: Execução Segura de Código, Módulo de Segurança Criptográfico, Model Carrying Code, seL4.

ABSTRACT

This final year project intend to search for correlated projects in the literature, searching for solutions to guarantee safe code execution and their application in the context of a Hardware Security Module. The safe execution of code is an open problem , and it is crucial against the leak of secret information that compromise all kinds of systems, and as consequence, their users, as individuals or entities. The big problem in guarateeing safe code execution is given by the great complexity that computational systems have reached, becoming too hard to proof or to antecipate all the possible behaviours and flaws on them. The choice of applying these techniques in a hardware security module is that it is not a general purpose system and it has less performance requisites, enabling us to think more about security over performance. The objective is to find the problems that exist while trying to implement a safe code execution mechanism, study the existent techniques and propose solutions to reach our goal. The safe code execution problem can be seen by various aspects, (KLEIN et al., 2009), (SEKAR et al., 2003), (DHURJATI; KOWSHIK; ADVE, 2006), are some works that propose solutions. This study intends to show different techniques that are being applied while trying to execute code in a safe way, showing their strong and weak points and in the end, propose a work plan for the development of a mechanism that will be implemented in the target hardware security module, the ASI-HSM.

Keywords: Safe code execution, Hardware Security Module, Model Carrying Code, seL4.

LISTA DE FIGURAS

SUMÁRIO

1 INTRODUÇÃO

1.1 CONTEXTUALIZAÇÃO

A segurança, em qualquer nicho, tem se tornado um fator primordial. Na computação não é diferente. Todos os dias acompanhamos notícias sobre sistemas computacionais que foram invadidos e informações sigilosas roubadas e divulgadas. Até mesmo empresas renomadas que desenvolvem soluções para segurança computacional tem sido vítimas constantes de ataques.

Grande parte dos problemas de segurança, deve-se a evolução das redes de computadores e dos serviços oferecidos pelas mesmas, principalmente através da internet. Estes serviços já estão tão fortemente ligados ao nosso dia-a-dia, proporcionando uma praticidade tão grande, que é praticamente impossível simplesmente passar a ignorá-los tendo em vista a segurança de nossos dados. Por este motivo a segurança computacional deixou de ser um simples fator adicional e tornou-se um quesito de forte impacto.

A maior parte dos problemas de segurança computacional, do ponto de vista técnico, deve-se a código mal escrito, resultando em programas vulneráveis a ataques e programas desenvolvidos para explorar estes ataques. Estes códigos mal escritos podem estar inseridos em qualquer programa, desde a aplicação final que está sendo desenvolvida a tecnologia que está sendo utilizada assim como no sistema operacional.

Outro problema ao se falar em executar código de forma segura, além de garantir que o código não possua falhas, é especificar ao usuário (aquele que vai executar o código), qual o comportamento do programa, e conseguir provar a este usuário que o programa executa de acordo com esta especificação. Podemos notar dois conceitos diferentes: provar ao usuário o comportamento do programa e garantir que o programa sempre opere da maneira esperada, ou seja, provar que a especificação do programa está correta.

É importante destacar estes dois conceitos pois eles devem ser considerados de forma conjunta. É possível provar com exatidão que um dado código sempre vai executar de acordo com a especificação, mas provar que a especificação está correta e abrange todos os, e apenas os corretos, fluxos de execução de um programa, é algo complicado.

1.2 TRABALHOS EXISTENTES

Vários trabalhos e pesquisas tem sido desenvolvidos no sentido de solucionar o problema, das mais variadas formas. Vamos discorrer aqui sobre alguns deles.

A certificação digital utiliza-se da assinatura de código por parte do desenvolvedor, para garantir que um código qualquer possa ser verificado no momento da execução. Isso nos garante a autenticidade, ou seja, sabemos quem é o autor do código, e a integridade do mesmo (o código que saiu do fabricante é o código que está sendo executado na máquina do usuário).

Outros trabalhos como (SEKAR et al., 2003) e (NECULA, 1997) utilizam-se de análise e geração de provas e modelos formais em um nível mais alto. Os modelos podem ser utilizados para especificar ao usuário o comportamento esperado do código, já a prova mostra que o código será executado segundo tal especificação. Estes modelos e provas são gerados através de duas técnicas principalmente, a análise estática e a análise dinâmica do código. A análise estática por si só, dificilmente consegue prevêr todos os fluxos de execução do programa. Por outro lado a análise dinâmica gera uma carga grande afetando muito o desempenho. Dessa forma estas técnicas são comumente utilizadas em conjunto.

(KLEIN et al., 2009) trabalha na prova formal de um micro-kernel da família L4. o seL4 define uma especificação e prova que a implementação segue essa especificação. Em (DHURJATI; KOWSHIK; ADVE, 2006) os autores desenvolveram uma técnica para detectar erros de memória, evitando diversas formas de ataque. É utilizada sempre que possível analise estática de código para evitar verificações em tempo de execução, deixando o código com um overhead não proibitivo comparado com outras técnicas.

1.3 PROPOSTAS DE SOLUÇÕES EXISTENTES

Os certificados digitais resolvem o problema da autenticação do produtor do código, porém não garante nada sobre o código em si, desta forma os certificados podem ser utilizados conjuntamente com uma solução para o problema de execução segura, pois fornece um fator de segurança adicional (a autenticação).

A análise estática de código é capaz de detectar problemas no código fonte, resolvendo os casos passíveis de detecção em tempo de compilação, entretanto não é capaz de prever o comportamento dinâmico dos programas. A análise dinâmica de código vem justamente no sentido de detectar vulnerabilidades em tempo de execução, o problema fica por parte de como detectar

todos os possíveis cenários de ataque.

O Proof Carrying Code é capaz de provar formalmente o comportamento do programa em cima da prova fornecida pelo produtor, desta forma o consumidor é capaz de verificar aspectos do código fornecido pelo produtor, mas não é capaz de detectar aspectos do comportamento da aplicação que não são provadas pelo produtor, desta forma o produtor é responsável por apontar o que é e o que não é relevante do ponto de vista de segurança.

O Model Carrying Code define o sistema de políticas para que o consumidor possa definir e por em prática os seus próprios conceitos do que é ou não é seguro e o que pode ou não pode ser feito. O monitoramento é feito em tempo de execução e o comportamento pode ser checado antes de ser efetivamente executado.

1.4 PROBLEMAS EM ABERTO

Além de técnicas que sejam capazes de detectar problemas de código mal escrito em tempo de execução e de técnicas para correção de problemas em tempo de compilação, precisamos de uma abordagem que torne possível a execução de programas concorrentes em uma mesma plataforma sem que uma ferramenta interfira no funcionamento da outra, no nosso caso esta interferência se foca principalmente quanto ao acesso a certos recursos restritos de um e outro programa.

No caso do HSM (foco deste trabalho), não conseguimos embarcar novos softwares na plataforma garantindo o mesmo nível de segurança (ou um nível de segurança razoável), tampouco podemos garantir que duas aplicações executem concorrentemente sem interferências.

Outro problema é que mesmo que alguma das técnicas existentes pudesse solucionar o problema de forma completa, ou com um nível de garantia aceitável, caímos no problema de utilização de código de terceiros e problemas de licença. Estes foram motivos que levaram também a não utilização de Módulos de Segurança Criptográficos de tecnologia importada.

Outro problema é que atualmente, mesmo desenvolvendo o software de gerência de chaves, o sistema operacional que gerencia o hardware em ultima instância é um FreeBSD, sistema operacional também desenvolvido por terceiros.

1.5 FOCO DO TRABALHO

Este trabalho visa identificar e propor soluções nacionais para o problema de execução segura de código dentro de um Módulo de Segurança Criptográfico (HSM), esperando garantir que duas aplicações concorrentes possam ser executadas simultaneamente sem que uma interfira no funcionamento da outra e sem que nenhuma das duas aplicações sejam fonte de vulnerabilidades que enfraqueçam o perímetro de segurança do HSM.

O perímetro de segurança do HSM é cheio de restrições, justamente para que sejam inviabilizados ataques ao mesmo. Porém esse perímetro seria enfraquecido caso novas aplicações fossem embarcadas na plataforma.

1.6 PROPOSTA PRELIMINAR

1.6.1 Objetivos Gerais

O objetivo geral deste trabalho é propor uma solução para o problema de execução segura de código de modo que o HSM não perca seu perímetro de segurança diferenciado ao embarcar novos aplicativos ou permitir o acesso através de softwares pela rede.

1.6.2 Objetivos Específicos

1. Estudar e propôr um mecanismo de detecção (e resolução) de erros de memória em tempo de execução e compilação.

2. Estudar e propôr um mecanismo que não permita que duas aplicações concorrentes no HSM acessem uma o material sensível da outra.

3. Permitir que qualquer aplicação, independente da linguagem, possa executar no HSM com o mecanismo de execução segura.

1.6.3 Riscos

Como o problema de execução segura de código vem sendo tratado a tempos em diversos lugares ao redor do globo sem que se tenha uma solução definitiva para o problema. O objetivo deste trabalho é propôr uma solução para a execução segura de código no ambito do HSM, e o risco é de que esta

solução não possa ser detalhada completamente no escopo deste trabalho.

1.6.4 Resultados Esperados

Esperamos com este trabalho realizar um estudo sobre a área de execução segura de código, soluções, trabalhos e ferramentas existentes, identificação de problemas nas ferramentas existentes e problemas ao se utilizar especificamente estas ferramentas e técnicas no contexto do ASI-HSM e propor uma solução que aumente a confiança no perímetro de segurança do HSM de forma a viabilizar um aumento nas aplicações do HSM.

2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo visamos dar uma introdução sobre o tipo de problema, ou área específica da segurança em computação, que será abordado durante o trabalho, contextualização sobre o Módulo de Segurança Criptográfico (HSM) e sobre a infraestrutura de compilação LLVM.

2.1 SEGURANÇA

2.1.1 Introdução

A segurança computacional é uma sub-área da computação que por sua vez envolve outras diversas sub-áreas específicas que tratam de vários aspectos. Nesta seção serão abordados alguns conceitos básicos sobre os aspectos explorados no escopo deste trabalho.

2.1.2 Memory Corruption

Erros de corrupção de memória são responsáveis por inúmeras falhas em sistemas computacionais. Ocorrem geralmente quando por falta de conhecimento ou descuido, um programador faz mal gerenciamento da memória permitindo que entradas maliciosas por parte do usuário ocasionem acesso a endereços de memória não permitidos.

Um subtipo de erro de corrupção de memória muito antigo e muito comumente encontrado ainda hoje são os erros de buffer-overflow, ou estouro de arranjo. Este erro ocorre quando por exemplo o programador não checa os limites do array e área de memória após o fim do array são escritas com a entrada do usuário. Este caso pode resultar em um Segmentation Fault e o programa é abortado por operar de uma forma inesperada.

Para exemplificar, vamos tomar como exemplo o seguinte trecho de código escrito na linguagem C, retirado do livro The ShellCoder's Handbook (ANLEY et al., 2011):

```
// serial.c

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```
int valid_serial(char *psz) {

        size_t len = strlen(psz);
        unsigned total = 0;
        size_t i;

        if(len < 10)
                return 0;

        for(i = 0; i < len; i++) {
                if((psz[i] < '0') || (psz[i] > 'z'))
                        return 0;

                total += psz[i];
        }

        if(total % 853 == 83)
                return 1;

        return 0;
}

int validate_serial() {

        char serial[24];
        fscanf(stdin, "%s", serial);

        if( valid_serial(serial))
                return 1;
        else
                return 0;
}

int do_valid_stuff() {

        printf("The serial number is valid:\n");
        exit(0);
}

int do_invalid_stuff() {
```

```
        printf("Invalid serial number!\nExiting\n");
        exit(1);
}

int main (int argc, char *argv[]) {

        if(validate_serial())
                do_valid_stuff();
        else
                do_invalid_stuff();

        return 0;
}
```

Este trecho de código representa um programa que faz a validação de um número serial digitado pelo usuário. Podemos observar que a função validate_serial() chamada a partir do método main, lê o número serial do usuário. Este serial é salvo no array serial que é do tipo char e possui tamanho 24.

Note que nenhuma verificação sobre o tamanho da entrada é realizada, dessa forma se escrevermos um número serial com tamanho maior que 24, os endereços adjacentes da memória serão sobrescritos. Nosso intuito é explorar essa vulnerabilidade de forma que possamos fazer o programa executar a função do_valid_stuff() sem que tenhamos conhecimento de um serial valido.

Para fazer isso, precisamos sobrescrever o endereço de retorno da função, com o endereço da função do_valid_stuff(). Para descobrir essas informações vamos utilizar o gdb para debugar o programa, lembrando que vamos utilizar algumas flags de compilação para que fique mais simples de entender o processo.

O comando utilizado foi gcc -ggdb -mpreferred-stack-boundary=2 para que a pilha seja incrementada sempre com o tamanho de uma palavra, caso não utilizassemos essa flag, o compilador poderia otimizar a pilha e dificultar o processo.

Nas versões mais atuais, o gcc possui mecanismos de proteção contra esse tipo de ataque, então pode ser necessário utilizar a flag -fno-stack-protector ao compilar o código.

```
(gdb) disas main
Dump of assembler code for function main:
0x080485e6 <+0>:  push %ebp
0x080485e7 <+1>:  mov  %esp,%ebp
0x080485e9 <+3>:  call 0x804856a <validate_serial>
```

```
0x080485ee <+8>:  test %eax,%eax
0x080485f0 <+10>: je   0x80485f9 <main+19>
0x080485f2 <+12>: call 0x80485aa <do_valid_stuff>
0x080485f7 <+17>: jmp  0x80485fe <main+24>
0x080485f9 <+19>: call 0x80485c8 <do_invalid_stuff>
0x080485fe <+24>: mov  $0x0,%eax
0x08048603 <+29>: pop  %ebp
0x08048604 <+30>: ret
End of assembler dump.
```

Quando a função validate_serial() for chamada, o prolog será executado e os valores salvos na pilha serão o %ebp (4 bytes) as variáveis locais, neste caso o array serial (24 bytes) e o endereço de retorno da função, que equivale a próxima função a ser executada antes da chamada da função.

Sabendo disso, para sobrescrever o endereço de retorno com o endereço da função do_valid_stuff() (0x080485f2) precisamos sobrescrever 28 bytes (%ebp + array) aleatóriamente e os próximos 4 bytes com o endereço. Para fazer isso vamos utilizar o printf do bash.

```
$printf "AAAAAAAAAABBBBBBBBBB
        CCCCAAAA\xf2\x85\x04\x08" | ./serial

The serial number is valid:
```

Observamos que o endereço de retorno é sobrescrito com sucesso e nossa função é invocada corretamente. Um observação a ser feita é que o endereço dos bytes foi escrito de forma invertida por estarmos executando este código em uma arquitetura IA32 que é little-endian.

2.1.3 SQL Injection

Falhas de SQL Injection estão ligadas a erros deixados pelo programador ao escrever código que faz acesso ao banco de dados. Existem diversos cenários de erro onde o código SQL pode ser injetado, mas uma falha dessas pode possibilitar um atacante a realizar praticamente qualquer tipo de operação sobre o banco.

Existem tecnologias e boas práticas ao escrever código para acesso a banco de dados que invibializam ou diminuem a chance de ocorrência deste ataque. Geralmente os erros de SQL Injection estão associados a aplicações web (é considerado umas das 10 maiores vulnerabilidades encontradas em aplicações web em 2007 e 2010, segundo a Open Web Application Security Project - OWASP), entretanto podem ocorrer em qualquer tipo de aplicação.

A forma mais simples de explicar como se dá um ataque de SQL Injection é através de exemplos (retirado de http://en.wikipedia.org/wiki/SQL_injection (WIKIPEDIA, 2012)):

```
statement = "SELECT * FROM users
            WHERE name = '" + userName + "';"
```

Este código simples foi escrito para obter todas as entradas onde o nome do usuário = username. Porém observe que userName é uma entrada do usuário. Caso não haja nenhum filtro sobre a pesquisa, um usuário malicioso tem algumas opções. Suponha que este código seja utilizado para autenticação, o atacante pode utilizar a seguinte entrada:

```
' OR '1'='1
```

Se juntarmos a entrada maliciosa a consulta inicial, teremos no final:

```
SELECT * FROM users WHERE name = '' OR '1'='1';
```

O que o usuário malicioso fez foi passar um nome de usuário nulo, apenas fechando a aspa simples do nome do usuário e continuou inserindo uma cláusula OU seguido de um código que é sempre avaliado como verdadeiro ('1' = '1'), forçando a seleção de um nome de usuário válido. Outro exemplo de inserção maliciosa poderia ser:

```
a';DROP TABLE users;
```

Juntando novamente a entrada maliciosa a consulta inicial, teremos:

```
SELECT * FROM users WHERE name = 'a';
         DROP TABLE users;
```

Que apaga toda a tabela de usuários.

2.1.4 Conclusão

Nesta seção observamos alguns conceitos básicos de segurança, a maioria deles conceitos de segurança de código que é um problema grave de segurança enfrentado atualmente. Estes conceitos simples servem de base e exemplo para entender as melhorias de segurança propostas neste trabalho e sua importância.

2.2 MÓDULO DE SEGURANÇA CRIPTOGRÁFICO

2.2.1 Introdução

Um HSM (Hardware Security Module) é um hardware que tem como objetivo principal gerenciar material sensível, como chaves criptográficas.

O SANS institute em seu whitepaper An Overview of Hardware Security Modules (ATTRIDGE, 2002), define HSM como um hardware e seu respectivo software/firmware que geralmente se conecta dentro de um computador ou servidor, provendo um mínimo de funções criptográficas como por exemplo (mas não limitado à) cifragem, decifragem, geração de chaves e funções de hash. O instituto ainda conceitua que esses dispositivos oferecem algum tipo de proteção física e possui uma interface de usuário e uma interface programável.

Os HSMs são empregados em situações onde o gerenciamento da chave é vital, como por exemplo em infraestrutura de chaves públicas ou sistemas bancários online. A gestão das chaves deve ser feita pelo HSM de forma que a mesma nunca seja visível para o mundo externo, deve ser criada, utilizada e caso necessário destruída dentro do HSM.

2.2.2 ASI-HSM

O ASI-HSM é a plataforma alvo deste trabalho. É um HSM produzido no Brasil, possui seus componentes importados, porém a fabricação e os softwares são desenvolvidos em parceria pela empresa Kryptus e o LabSEC (Laboratório de Segurança em Computação) da UFSC.

O ASI-HSM é um equipamento homologado pelo Instituto Nacional de Tecnologia de Informação (ITI) que é o orgão que realiza este processo no Brasil de acordo com as normas e requisitos contidos no documento MCT (Manual de Condutas Técnicas) 7, (BEREZA JÚNIOR, A.,) contém informações mais detalhadas sobre o processo de homologação. O ASI-HSM consiste de um sistema operacional (FreeBSD) que contém apenas as aplicações e bibliotecas estritamente necessárias ao seu funcionamento.

Uma visão geral da arquitetura do ASI-HSM pode ser visualizada na figura abaixo retirada de (SOUZA, 2008). Nas próximas sessões iremos apresentar as partes mais importantes desta arquitetura, essenciais para a compreensão do perímetro criptográfico do HSM. Informações detalhadas sobre o funcionamento completo do HSM e suas aplicações podem ser obtidas em (MARTINA, 2005), (SOUZA, 2008), (Rick Lopes de Souza,).

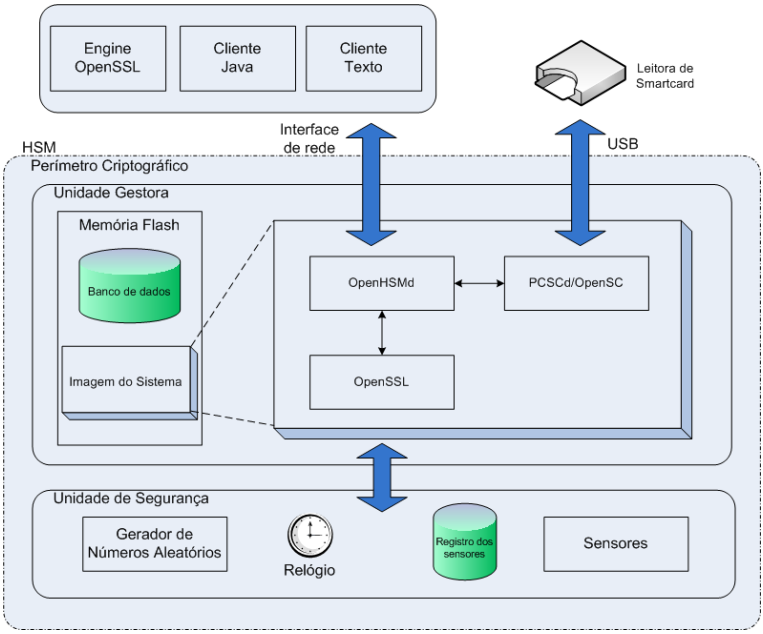

Figura 1: Arquitetura do ASI-HSM.

2.2.2.1 Segurança Física

Fisicamente o ASI-HSM é separado em duas unidades, a unidade de gerência (UG) e a unidade de segurança (US). É importante citarmos estas unidades físicas pois nos ajuda a entender o perímetro de segurança diferenciado do HSM. A UG possui os componentes de hardware essenciais a qualquer computador, os principais são (BEREZA JÚNIOR, A.,):

1. Placa mãe ALIX, modelo alix3d2
2. Processador AMD Geode 500MHz
3. Memória 256MB DDR DRAM
4. Compact Flash com capacidade de 1GB

Todos estes componentes estão protegidos por uma série de sensores que são controlados pela US, agregando uma grande segurança física aos componentes onde são salvos os softwares e o material sensível protegido

pelo HSM. Os mais importantes desses sensores são (BEREZA JÚNIOR, A.,):

1. Luminosidade: disparado caso o perímetro protetor da UG seja violado e um feixe de luz seja detectado.

2. Temperatura: disparado ao esquentar ou resfriar o HSM de forma excessiva.

3. Tensão: disparado caso ocorra a elevação da tensão de entrada.

Além de controlar os sensores que detectam alterações na UG, a US guarda uma chave criptográfica simétrica que é utilizada para cifrar os dados da Compact Flash (onde se encontra o material sensível protegido pelo HSM), garantindo o sigilo dos dados. Caso os sensores físicos detectem uma tentativa de ataque, a US apaga essa chave, mantendo os dados cifrados e prevenindo o acesso aos mesmos. A US também é responsável pela geração (em hardware) da semente utilizada no algoritmo de geração de número aleatórios. E por último, ela possui um relógio de alta estabilidade para realizar o controle de relógio.

2.2.2.2 Segurança Lógica

Na parte lógica, o ASI-HSM utiliza um sistema de grupos de gerenciamento. Cada grupo é responsável por certas funções pré-definidas e para realizar a execução destas, o grupo precisa passar por um processo de autenticação.

O ASI-HSM possui três grupos de gerenciamento: grupo de administradores, grupo de auditores e grupo de operadores. O grupo de administradores é único e é responsável por criar todos os outros grupos. Suas principais funções são (BEREZA JÚNIOR, A.,):

1. Apagar configurações

2. Atualizar firmware

3. Desligar

4. Alterar data/hora

5. Gerar/Recuperar backup

6. Gerar par de chaves assimétricas

Os auditores podem possuir mais de um grupo, sendo necessária a existência de no mínimo um grupo. Suas principais funções são: (BEREZA JÚNIOR, A.,):

1. Exportar logs
2. Bloqueio do dispositivo
3. Recuperar backup

Os operadores são responsáveis pelo gerenciamento do uso de chaves criptográficas, eles não as criam (esta função é do grupo de administradores) porém gerenciam quando as chaves podem ser utilizadas e quantas vezes. Suas principais funções são (BEREZA JÚNIOR, A.,):

1. Carregar chave para uso
2. Definir políticas de utilização da chave

   Tempo de uso

   Número de usos

Através destes grupos, o HSM define protocolos de utilização e restrições lógicas sobre a execução de funções, aumentando seu perímetro de segurança. Esta seção não tem o objetivo de explicar exaustivamente os processos de autenticação e gerenciamento do HSM, o objetivo como já citado no inicio deste capítulo é apenas dar uma explanação básica sobre o funcionamento do ASI-HSM, para que se possa entender o perímetro de segurança que o protege.

2.2.3 OpenHSMd

O OpenHSM é o software de gerenciamento do ASI-HSM desenvolvido pelo LabSEC. É responsável pela implementação das funções criptográficas que serão disponibilizadas bem como funções de backup, gerenciamento do material sensível e todas as funcionalidades mencionadas nos grupos de gerenciamento. Este software possui basicamente três interfaces de comunicação com o mundo externo, uma interface gráfica, um cliente texto (interfaces de administração) e uma Engine OpenSSL.

Estas interfaces definem os possíveis acessos ao material sensível guardado pelo HSM bem como as funções disponibilizadas pelo mesmo. As interfaces também possibilitam acesso a realização de rotinas de configuração e administração, estas duas ultimas sendo realizadas através do cliente texto e

da interface gráfica. A utilização das funcionalidades é realizada via Engine OpenSSL. Toda a comunicação das interfaces de administração é realizada com protocolo seguro SSL/TLS.

2.2.3.1 Interface Gráfica

A interface gráfica é implementada na linguagem Java por questões de portabilidade. Java é considerada uma linguagem segura por ter entre outras características, um sistema fortemente tipado e por realizar o gerenciamento de memória independentemente do usuário.

Apesar de a interface ser implementada em Java e esta ser considerada uma linguagem segura, as entradas do usuário são enviadas diretamente ao servidor OpenHSMd (via protocolo SSL/TLS), que é implementado em C. Caso o servidor não faça as verificações necessárias nas entradas, estas entradas originadas via interface gráfica podem gerar erros. Contudo, o software que implementa a interface gráfica é bem conhecido e poucas são as entradas por parte do usuário. A interface gráfica do ASI-HSM pode ser visualizada na Figura 2.

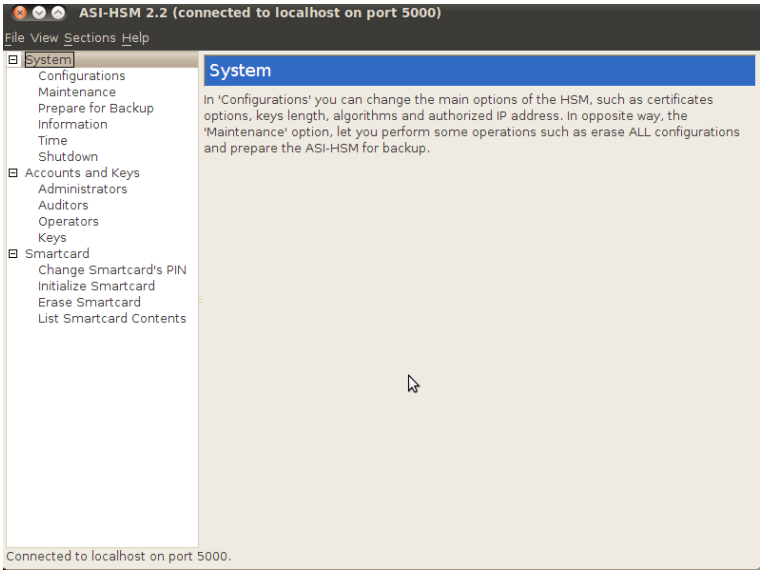

Figura 2: Interface Gráfica.

2.2.3.2 Interface Texto

A interface texto é uma ferramenta de linha de comando específica para sistemas UNIX-like escrita em C. Em questões de funcionalidades, ela é equivalente a interface gráfica. O servidor OpenHSMd interpreta as mensagens vindas de ambas as interfaces da mesma forma, porém nesta interface texto o usuário tem mais controle sobre as entradas, abrindo uma gama maior de possibilidades de ataques. Uma imagem da interface texto é mostrada na Figura 3.

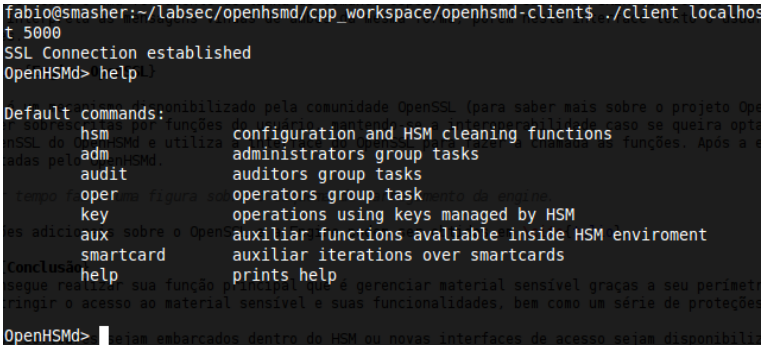

Figura 3: Interface Texto.

2.2.3.3 Engine OpenSSL

A engine é um mecanismo disponibilizado pela comunidade OpenSSL (para saber mais sobre o projeto OpenSSL acesse: http://www.openssl.org/ (OPENSSL, 2012)) para que funções padrões desta biblioteca possam ser sobrescritas por funções do usuário, mantendo-se a interoperabilidade caso se queira optar por qualquer uma das duas. Neste caso, uma aplicação cliente carrega a Engine OpenSSL do OpenHSMd e utiliza a interface do OpenSSL para fazer a chamada as funções. Após a engine ter sido carregada as chamadas serão redirecionadas para as funções implementadas pelo OpenHSMd.

Informações adicionais sobre o OpenSSL e a Engine podem ser obtidas em (SOUZA, 2008).

2.2.4 Conclusão

O HSM consegue realizar sua função principal que é gerenciar material sensível graças a seu perímetro de segurança diferenciado. Como citado, o HSM possui funções bem definidas para restringir o acesso ao material sensível e suas funcionalidades, bem como um série de proteções de hardware que dificultam a realização de ataques.

Caso novas aplicações sejam embarcadas dentro do HSM ou novas interfaces de acesso sejam disponibilizadas (principalmente com conexão através de redes), este perímetro de segurança se enfraquece, dessa forma precisamos de um mecanismo adicional que aumente a segurança do perímetro, para tornar estas novas possibilidades viáveis.

Maiores informações sobre o HSM podem ser encontrados em (MARTINA, 2005), (SOUZA, 2008), (BEREZA JÚNIOR, A.,), (Rick Lopes de Souza,).

2.3 MÁQUINAS VIRTUAIS

2.3.1 Introdução

Máquinas virtuais, como o nome já expressa, são máquinas em software que executam programas como se fossem um computador físico. O conceito de máquinas virtuais foi levantado no âambito deste estudo por fornecer interoperabilidade de arquiteturas e a possibilidade de reutilização de trabalhos já existentes. Apesar de a reutilização de trabalhos não parecer tão óbvia quando falamos em máquinas virtuais convencionais, a opção que tomamos vai tornar claro este conceito e será explicado ao longo do texto.

Outra vantagem que parece interessante é a segmentação de memória que as máquinas virtuais criam quando rodadas em cima de uma única máquina real, que se encaixa perfeitamente nos objetivos que queremos atingir com este estudo.

2.3.2 Low Level Virtual Machine

Apesar de ter o nome de máquina virtual, a LLVM tem pouco a ver com máquinas virtuais convencionais, porém pode ser utilizada para construir as mesmas. Em LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation (LATTNER; ADVE, 2004) os autores descrevem

que a LLVM é complementar e não uma alternativa a máquinas virtuais de alto nível, como por exemplo a Java Virtual Machine (JVM).

A LLVM, segundo LATTNER em descrição do site do próprio projeto é uma coleção de ferramentas e compiladores modulares e reusáveis. Ainda segundo o mesmo autor, a intenção inicial do projeto era de prover uma técnica de compilação, que suportasse ambas as técnicas de compilação: estática e dinâmica, para linguagens de programação arbitrárias.

O autor explica que a LLVM define uma representação comum de código de baixo nível na forma Static Single Assignment (SSA) que inclui uma série de funcionalidades: um sistema de tipagem simples que provê as primitivas comumente utilizadas para implementar funcionalidades de linguagens de programação em alto nível; uma instrução para aritmética de endereço tipado; e um mecanismo simples, capaz de implementar funcionalidades de tratamento de exceções em linguagens de alto nível (e setjmp/-longjmp em C) de forma uniforme e eficiente.

A LLVM foi escolhida para o contexto deste trabalho por 3 aspectos:

1. Heterogeneidade de linguagens - Uma vez que o mecanismo de análise estática de código for implementado na linguagem intermediária da LLVM, um programa escrito em uma linguagem qualquer é passível da análise, caso haja um front-end para a linguagem desenvolvido no contexto da LLVM.

2. Reutilização de trabalhos da comunidade - A LLVM conta com uma comunidade ativa de colaboradores e diversos projetos já desenvolvidos que podem vir a ser reutilizados neste contexto, como por exemplo o SafeCODE (Para detecção de erros de memória).

3. Suporte

2.3.2.1 Arquitetura

A LLVM possui uma característica muito importante que é uma implementação desacoplada de um arquitetura de compiladores muito conhecida, que é a arquitetura em 3 passos, mas que poucos compiladores existentes implementam, a representação desta arquitetura é exemplificada na Figura 4, retirada de http://www.aosabook.org/en/llvm.html (LATTNER, 2012a).

Os principais componentes desta arquitetura são o front-end, o otimizador e o back-end. O front-end é responsável por fazer o parsing do código fonte checando erros e construindo a árvore de sintaxe abstrata que representa o código de entrada. O otimizador é responsável por fazer uma

série de transformações no código para melhorar o desempenho em tempo de execução.

O back-end é responsável por gerar código para a arquitetura correspondente. Além de gerar o código corretamente, o back-end é responsável por gerar um código de máquina que se beneficie das características singulares da arquitetura alvo.

Figura 4: Os três componentes principais de uma arquitetura de três passos.

Essa característica torna o compilador versátil quanto ao suporte de multiplas linguagens de programação e multiplas arquiteturas alvo. No caso de o compilador utilizar-se de uma representação de código comum no otimizador, pode-se adicionar uma nova linguagem escrevendo um frontend para ela, e de forma similar, caso queira-se adicionar uma nova arquitetura alvo, basta que se escreva um backend para a mesma. Este cenário é ilustrado na Figura 5, retirada de http://www.aosabook.org/en/llvm.html (LATTNER, 2012a).

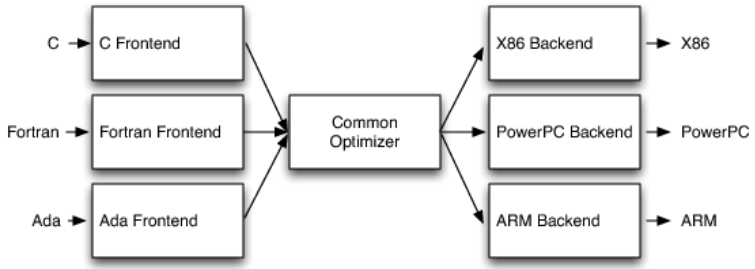

Figura 5: Versatilidade de um compilador de três passos.

Podemos concluir que a versatilidade e poder da LLVM, são originados de sua implementação da arquitetura de 3 passos e de sua representação intermediária bem definida. Para que se escreva um front-end para a LLVM, basta que o usuário conheça o LLVM-IR e suas invariantes.

O autor destaca que após o aspecto da arquitetura, o fato de a LLVM ser projetada para ser um conjunto de bibliotecas ao invés de um compilador monolítico de linha de comando como o GCC, ou um máquina virtual opaca

como a JVM, é o que a torna uma ferramenta tão poderosa. Ele diz que a LLVM é uma infra-estrutura, uma coleção de tecnologias de compiladores ou bibliotecas que podem ser empregadas em problema específicos, ressaltando que as bibliotecas da LLVM não fazem nada por si só, mas provêem uma série de funcionalidades que o desenvolvedor deve juntar da forma que melhor lhe convenha.

2.3.2.2 Representação Intermediária

O autor cita que a representação de código da LLVM (LLVM-IR) é uma de suas características arquiteturais mais importantes. Esta é a forma como o código é representado no compilador e foi feito de forma a abrigar análises intermediárias e transformações geralmente encontradas no otimizador do compilador. Abaixo temos um exemplo de código C,

```
unsigned add1(unsigned a, unsigned b) {
  return a+b;
}

unsigned add2(unsigned a, unsigned b) {
  if (a == 0) return b;
  return add2(a-1, b+1);
}
```

e sua representação em LLVM-IR, retirado de http://www.aosabook.org/en/llvm.html (LATTNER, 2012a).

```
define i32 @add1(i32 %a, i32 %b) {
entry:
  %tmp1 = add i32 %a, %b
  ret i32 %tmp1
}

define i32 @add2(i32 %a, i32 %b) {
entry:
  %tmp1 = icmp eq i32 %a, 0
  br i1 %tmp1, label %done, label %recurse

recurse:
  %tmp2 = sub i32 %a, 1
  %tmp3 = add i32 %b, 1
  %tmp4 = call i32 @add2(i32 %tmp2, i32 %tmp3)
```

```
  ret i32 %tmp4

done:
  ret i32 %b
}
```

O autor informa que a LLVM-IR é um instruction set virtual de baixo nível de arquitetura RISC-like e é uma representação completa de código. Ele cita que a LLVM-IR diferentemente de instruction sets RISC, é fortemente tipada, com um sistema de tipagem bastante simples e que não usa um conjunto fixo de registradores e sim um conjunto infinito de temporários nomeados com um caractere %, como observado no exemplo acima.

Os arquivos gerados pela LLVM citados por suas extensões, são os seguintes:

1. .ll - arquivos que contém a representação intermediária textual
2. .bc - llvm bitCode

A LLVM IR pode ser serializada/deserializada de forma eficiente para a forma binária chamada LLVM bit code. Como a LLVM IR é uma representação auto-contida e a serialização/deserialização é um processo sem perdas, parte da compilação pode ser realizada, salva em disco e resumida num momento futuro.

A primeira utilidade que o autor cita para essa propriedade são as otimizações em tempo de ligação. O compilador pode se utilizar desta funcionalidade, diz o autor, para realizar otimizações que não seriam possíveis pelo fato de o compilador enxergar apenas uma unidade de tradução (por exemplo um arquivo fonte .c com seus headers) não sendo possível realizar otimizações além das fronteiras do arquivo. A figura abaixo, retirada de http://www.aosabook.org/en/llvm.html (LATTNER, 2012a), ilustra o proicesso:

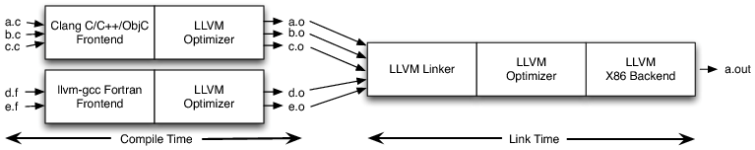

Figura 6: Otimizações em tempo de ligação.

O autor descreve o processo dizendo que o ligador detecta que há LLVM bitcode nos arquivos .o ao invés de codigo de máquina nativo, deste ponto em diante ele lê todos os bitcodes para a memória, junta todos e roda

o otimizador novamente sobre todo o agregado. Dessa forma o otimizador pode enxergar muito mais do código e realizar mais otimizações em cima do mesmo. Por ultima ele destaca o fato de por a LLVM IR ser neutra em relação ao código fonte, este processo de LTO (otimização em tempo de ligação) é algo natural, diferentemente de outros compiladores, que por não possuírem uma arquitetura com uma representação intermediária neutra, possuem um processo de serialização/deserialização lento e caro.

A segunda utilidade citada por ele são as otimizações em tempo de instalação, que ultrapassam o tempo de ligação. As vantagens de otimizações em tempo de instalação devem-se a ciência das específicidades da plataforma alvo, adquirindo dessa forma a melhor forma de otimizar o código para as características do hardware. A Figura abaixo, retirada de http://www.aosabook.org/en/llvm.html (LATTNER, 2012a), ilustra o processo:

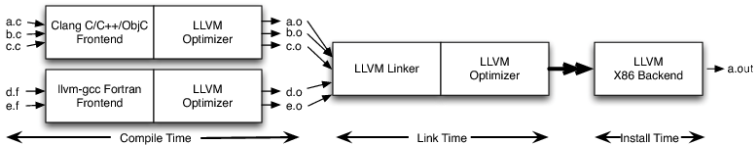

Figura 7: Otimizações em tempo de instalação.

2.3.3 Conclusão

A LLVM é uma escolha bastante útil e inteligente quando falamos de interoperabilidade e reuso. Sua infraestrutura e trabalhos existentes são alternativas interessantes ao se aplicar no propósito desta pesquisa. Posteriormente serão elucidadas as propostas que foram desenvolvidas utilizando a LLVM e seus projetos correlatos.

3 TRABALHOS CORRELATOS

Este capítulo irá destacar os principais trabalhos estudados durante a pesquisa, que serviram como embasamento para a formulação das soluções propostas, que serão indicadas posteriormente.

3.1 MODEL-CARRYING CODE

3.1.1 Introdução

O Model Carrying Code (MCC) foi um dos primeiros trabalhos no qual procuramos referência quando a idéia de uma mecanismo de execução segura de código foi pensado. A idéia do MCC é voltada mais para a execução segura de aplicações em dispositivos móveis, porém faz a utilização de políticas, o que era uma das idéias para o HSM, principalmente aplicadas sobre as chaves. Este capítulo detalha um pouco mais sobre o funcionamento do projeto seguindo (SEKAR et al., 2003).

3.1.2 A Abordagem

O framework do Model Carrying Code possui um lado produtor e outro lado consumidor. O produtor, é responsável por extrair um modelo, que cobre todos os aspectos de segurança relevantes do código. Uma ferramenta pode ser construída para que esse processo seja realizado automaticamente. Esse modelo é então enviado ao consumidor juntamente com o código da aplicação. A figura 8 ilustra o framework:

Antes de executar, o usuário pode verificar o modelo e confrontá-lo com as políticas locais de segurança. Essas políticas são definidas pelo usuário e o framework define uma linguagem para que estas políticas sejam especificadas. O comportamento do programa é avaliado através do que foram chamados de eventos. Em ultima instância, as ações relacionadas a segurança em um programa, são efetivadas na camada de system calls. Por esta razão as system calls constituem o alfabeto básico de eventos do Model Carrying Code.

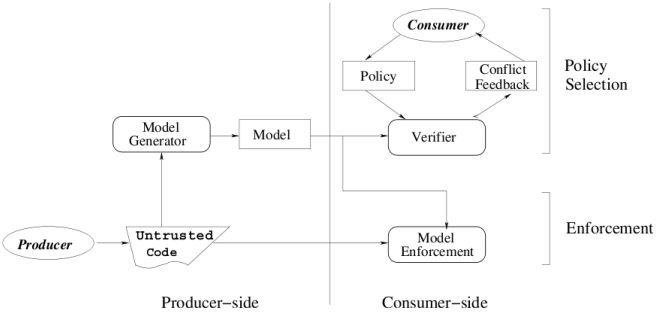

Figura 8: O Framework do Model Carrying Code.

3.1.3 Políticas de Segurança

Grande parte das políticas são escritas na forma de Expressões Regulares sobre Eventos, que são uma extensão das expressões regulares. Um compilador transforma estas expressões regulares em autômatos que são utilizados pelo verificador.

A figura ilustra um exemplo de política. Esta política permite qualquer comportamento, porém faz uma transição para um estado final caso o programa tente se conectar a um recurso fora da rede local ou tente abrir um arquivo com permissão de escrita.

Um produtor malicioso pode tentar gerar um modelo que não representa o correto comportamento do programa. Este problema é resolvido utilizando-se o que é chamado de Runtime Model Enforcement, que monitora a atividade do programa em tempo de execução e o verifica de acordo com o modelo.

Seguindo este modelo, o consumidor e o produtor ficam desacoplados no sentido de que o produtor não precisa saber nada sobre as restrições do consumidor e este pode checar a segurança do código antes da execução. Adicionalmente, o modelo extraído ajuda a encurtar o buraco semântico que há entre o código binário de baixo nível e as políticas de segurança de alto nível.

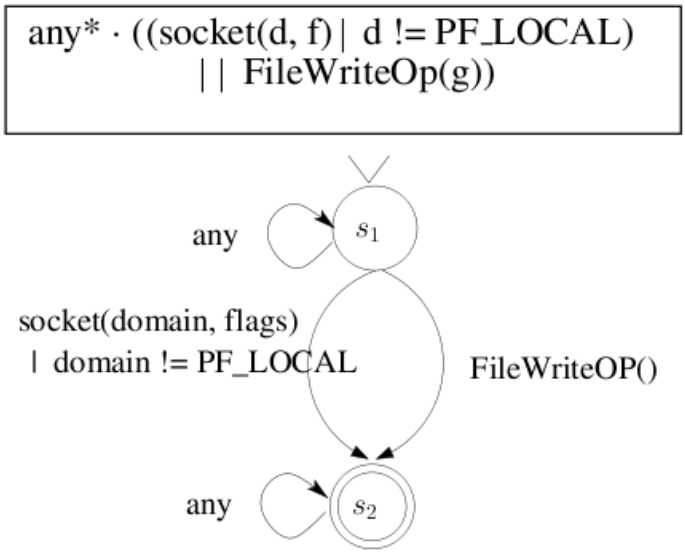

Figura 9: Exemplo de política para leitura de um arquivo sensível.

3.1.4 Extração do modelo

Os modelos são extraídos pelo consumidor utilizando o monitoramento da execução do programa. O produtor precisa ter casos de testes suficientes para cobrir todos os possíveis comportamentos do programa em tempo de execução. Caso os testes não sejam suficientes, o modelo pode não cobrir todos os comportamentos possíveis do programa e sua execução pode ser abortada durante a execução no lado do consumidor.

O extrator de modelo é constituído de dois componentes. O componente online, que intercepta as system calls e as loga juntamente com os argumentos úteis. O componente offline, constrói o autômato utilizando o log gerado pelo componente online.

3.1.5 Verificação

A verificação é realizada antes da execução, no lado do consumidor. A verificação é realizada através das políticas definidas pelo usuário e o comportamento descrito pelo modelo. As políticas definem apenas os comportamentos proibidos, e não os comportamentos permitidos. Para permitir que o usuário readapte suas políticas, uma mensagem é apresentada ao usuário no caso de violação:

```
open operation on file /tmp/logfile in write mode
socket operation involving the domain PF_INTER
```

Dessa forma o usuário pode ajustar suas políticas e partir ou não para a execução do código.

3.1.6 Model Enforcement

O correto fluxo do programa é checado com o modelo em tempo de execução. Esse trabalho é realizado por um módulo do kernel que intercepta as system calls e verifica se os estados de execução do programa estão corretamente espelhados no modelo. Este é a principal fonte de overhead no Model Carrying Code. Caso o código viole o modelo, a execução é abortada. As únicas razões para que isso ocorra são o caso de o modelo estar errado ou não cobrir todos os fluxos de execução possíveis.

3.2 SAFECODE

3.2.1 Introdução

O SAFECode é uma técnica de compilação desenvolvida e mais detalhadamente explicada em (DHURJATI; KOWSHIK; ADVE, 2006), para proteger os programas de erros de memória. A implementação concreta das técnicas foi realizada em um compilador, também chamado de compilador SAFECode, utilizando a estrutura da LLVM.

O intuito do SAFECode é a utilização de técnicas que permitam que o máximo de análises seja realizada em tempo de compilação (análise estática), e inserindo código para realizar análise em tempo de execução (análise dinâmica) apenas quando necessário. Estes objetivos são alcançados através do desenvolvimento de uma representação de código que permita que a maior parte

das análises sejam realizadas estaticamente.

Alguns exemplos citados no guia do usuário no site do projeto (ADVE, 2012) são buffer overflows, frees inválidos e dangling pointers (ponteiros que não apontam para objetos válidos do tipo apropriado).

O SAFECode também pode ser utilizado para diagnosticar erros de memória em programas, similar ao Valgrind (http://valgrind.org/).

Como o SAFECode é um projeto bastante complexo, não entraremos em detalhes sobre as técnicas aplicadas para garantir a segurança da memória. Focaremos na parte de utilização.

3.2.2 Compilando um Programa com o SAFECode

Os passos aqui citados podem ser encontrados no guia do usuário (http://safecode.cs.illinois.edu/docs/UsersGuide.html). Assumimos também que o SAFECode já se encontra devidamente instalado. Algumas dependências são necessárias, como a LLVM. Todos os passos para a instalação são encontrados em http://safecode.cs.illinois.edu/docs/Install.html.

O compilador Clang é um compilador que também é subprojeto da infraestrutura da LLVM. A maneira mais fácil de compilar um programa utilizando o SAFECode, é utilizar a versão modificada do Clang disponível a partir da instalação do SAFECode. Quando utilizado desta maneira, as transformações do SAFECode são realizadas de forma transparente. O comando:

```
clang -g -fmemsafety -c -o file.o file.c
```

utiliza o compilador Clang para compilar o programa. A flag -fmemsafety é utilizada para habilitar as transformações do SAFECode. A flag -g gera informações para debugar o programa e é utilizada pelo SAFECode para melhorar suas verificações em tempo de execução. Para exemplificar o seu funcionamento, utilizaremos o mesmo código do programa apresentado na sessão 2.1, o serial.c. Relembrando, o código continha um array (serial) que atribuia a entrada do usuário sem checar o seu tamanho, ocasionando em um possível buffer overflow.

```
int validate_serial() {

        char serial[24];
        fscanf(stdin, "%s", serial);

        if( valid_serial(serial))
                return 1;
```

```
        else
                return 0;
}
```

O programa é compilado normalmente com o SAFECode:

```
clang -g -fmemsafety -o cserial serial.c
```

3.2.3 Executando o programa

Ao executar o programa sem estourar o limite do array, nada acontece. Entramos com um string de tamanho 24 (o tamanho exato do array):

```
fabio@smasher:~/code$ ./serial
AAAAAAAAAABBBBBBBBBBCCCC
Invalid serial number!
Exiting
fabio@smasher:~/code$
```

Ao acrescentarmos mais um caracter, sobrescrevendo uma area de memória fora dos limites, a execução é abortada e um relatório apresentado o erro de tentativa de estourar os limites da pilha é apresentado:

```
fabio@smasher:~/code$ ./serial
AAAAAAAAAABBBBBBBBBBCCCC
Invalid serial number!
Exiting

fabio@smasher:~/code$ ./serial
AAAAAAAAAABBBBBBBBBBCCCCC

*** stack smashing detected ***: ./serial terminated
======= Backtrace: =========
/lib/tls/i686/cmov/libc.so.6(__fortify_fail+0x50)[0xb7696d10]
/lib/tls/i686/cmov/libc.so.6(+0xe7cba)[0xb7696cba]
./serial[0x8048624]
./serial[0x804866d]
/lib/tls/i686/cmov/libc.so.6(__libc_start_main+0xe6)[0xb75c5bd6
./serial[0x8048491]
======= Memory map: ========
08048000-08049000 r-xp 00000000 08:02 5114653 /home/fabio/code/
08049000-0804a000 r--p 00000000 08:02 5114653 /home/fabio/code/
0804a000-0804b000 rw-p 00001000 08:02 5114653 /home/fabio/code/
```

```
0863a000-0865b000 rw-p 00000000 00:00 0          [heap]
b7578000-b7595000 r-xp 00000000 08:04 6422555    /lib/libgcc_s
b7595000-b7596000 r--p 0001c000 08:04 6422555    /lib/libgcc_s
b7596000-b7597000 rw-p 0001d000 08:04 6422555    /lib/libgcc_s
b75ae000-b75af000 rw-p 00000000 00:00 0
b75af000-b7708000 r-xp 00000000 08:04 6569616    /lib/tls/i686
b7708000-b770a000 r--p 00159000 08:04 6569616    /lib/tls/i686
b770a000-b770b000 rw-p 0015b000 08:04 6569616    /lib/tls/i686
b770b000-b770e000 rw-p 00000000 00:00 0
b7724000-b7727000 rw-p 00000000 00:00 0
b7727000-b7728000 r-xp 00000000 00:00 0          [vdso]
b7728000-b7743000 r-xp 00000000 08:04 6432566    /lib/ld-2.11
b7743000-b7744000 r--p 0001a000 08:04 6432566    /lib/ld-2.11
b7744000-b7745000 rw-p 0001b000 08:04 6432566    /lib/ld-2.11
bfbb1000-bfbc6000 rw-p 00000000 00:00 0          [stack]
Aborted
fabio@smasher:~/code$
```

3.2.4 Conclusão

Observamos de forma breve e prática mais um trabalho que pode ser útil na tentativa de se encontrar uma solução para o problema de execução segura de código, uma vez que o SAFECode é uma ferramenta não verificada e provada formalmente. Ela aumenta a segurança de softwares porém não há garantias de que todos os erros serão corrigidos e a integridade do sistema mantida em todos os casos. Ela pode ser parte integrante de uma solução completa, ou para sistemas menos críticos, utilizada como fator adicional de segurança.

3.3 SEL4 MICROKERNEL

3.3.1 Introdução

O seL4 é um microkernel desenvolvido dentro do projeto Trustworthy Systems (ERTOS), do Software Systems Research Group (SSRG), integrante do NICTA. Ele é um software base crítico para o desenvolvimento de todos os demais sistemas, e será descrito nas próximas subseções.

3.3.1.1 NICTA

O NICTA (http://www.nicta.com.au/) é um centro de pesquisas criado pelo governo Australiano com o objetivo de realizar (segundo descrição do próprio grupo) pesquisas de grande impacto de forma excelente. Não somente limitados a pesquisa, a entidade se dedica a colocar em prática os resultados de forma a trazer benefícios e riquezas para a Austrália.

O grupo ainda ressalva que seu foco é estar entre um dos maiores centros de pesquisa em tecnologia de informação e comunicação no mundo. A entidade atua em 4 grandes áreas, sendo uma delas Segurança e Ambiente.

3.3.1.2 Software Systems Research Group

O Software Systems Research Group, tem como objetivo, mudar a forma com que sistemas de software são projetados, implementados e verificados. As principais áreas de atuação do grupo são: Sistemas operacionais, Métodos formais, Engenharia de Software e gestão de processos de negócio.

3.3.1.3 Trustworthy Systems (ERTOS)

O Trustworthy Systems (ERTOS), é um projeto que tem como visão, mudar o rumo de projeto (design) e implementação de software, focando aspectos como segurança, confiabilidade e eficiência. Eles visam atingir estes objetivos com teoria e prática no projeto de sistemas juntamente com a utilização de métodos formais. Para atingir o desenvolvimento de grandes sistemas computacionais com todas as características desejadas, foi necessário desenvolver um núcleo de software mínimo e eficiente. Daí surgiu o seL4.

3.3.1.4 O MicroKernel

Dentro do ERTOS, é desenvolvido o seL4, Secure Microkernel Project (Projeto de Microkernel Seguro). O seL4 é fundamental para o desenvolvimento de todos os demais projetos, visto que é a única parte de todo o software que roda no modo privilegiado do hardware.

Ele é desenvolvido baseado na arquitetura da família de microkernels L4, que tem como características o tamanho compacto, alta performance e independente de política, onde independência de política significa que as políticas do Sistema Operacional não são implementadas no kernel em si

mas delegadas a servidores ou outras estratégias de user-level. O seL4 ainda extende estas características com um modelo de capabilities, que provê um mecanismo para forçar as garantias de segurança nos níveis de sistema operacional e aplicação. A seguir o projeto e suas características serão explicados com uma maior profundidade, baseado em (KLEIN et al., 2009)

A segurança e confiabilidade de um sistema computacional só pode ser tão grande quanto as do kernel do sistema operacional a qual está submetido. O kernel é definido como a parte do sistema executando no modo mais privilegiado do processador e possui acesso ilimitado ao hardware. Dessa forma qualquer problema na implementação do kernel tem o pontecial de abalar o correto funcionamento do resto do sistema.

É de senso comum que bugs sempre serão encontrados em softwares de todos os tamanhos. Dessa forma a abordagem mais comum é reduzir a quantidade de código do kernel para diminuir a exposição a bugs.

Com um código pequeno o suficiente é possível garantir a não existência de bugs, através de verificação formal realizada por máquina, provendo uma prova matemática de que a implementação do kernel está condizente com a sua especificação e livre de defeitos de implementação induzidos pelo programador.

O seL4 foi projetado exatamente para oferecer esse último degrau de garantia de funcionamento, entretanto ele assume o correto funcionamento do compilador, código assembly, código de boot, gerenciamento de caches e do hardware. Todo o resto é provado.

Segundo o paper, o seL4 atinge os seguintes objetivos:

1. É adaptado para o uso prático, atingindo uma performance comparável com os melhores microkernels. (Neste ponto o autor se refere ao fato de que o desempenho não é degradado pelo fato de o kernel ser provado formalmente).

2. O seu comportamento é precisamente formalizado em um nível abstrato.

3. O seu projeto formal é utilizado para provar propriedades desejáveis, incluindo a segurança na finalização e execução.

4. Sua implementação é provada formalmente para satisfazer a especificação.

5. Seus mecanismos de controle de acesso são provados formalmente para prover fortes garantias de segurança.

Os autores afirmam que a propriedade de funcionamento correto que é provada no seL4 é muito mais robusta e precisa do que técnicas automáticas

como model checking, análise estática ou implementações de kernel com linguagens type-safety conseguiriam ser. Além de analisar aspectos específicos do kernel como execução segura, o seL4 fornece uma especificação completa e prova do comportamento preciso do kernel.

3.3.2 O modelo de programação do seL4

Nesta seção será apresentada uma visão das características principais do seL4.

O seL4 é um microkernel de terceira geração, quer dizer, um microkernel caracterizado por uma API orientada a segurança, controle de acesso a recursos controlados por capabilities (capabilities serão explicadas em capítulos posteriores), virtualização como uma preocupação principal, técnicas inovadoras no que diz respeito a gerenciamento de recursos do kernel e um projeto de arquitetura adequado para análise formal. Isso difere bastante do desenvolvimento usual de kernels que se preocupa primariamente com performance.

O microkernel possui abstrações características para espaços de endereçamento virtual, threads, comunicação inter-processo e, diferentemente de outros kernels da família L4, utiliza o mecanismo de capabilities para autorização.

Os drivers de dispositivos no seL4, como comumente em outros kernels da família L4, rodam como aplicações normais de usuário. Essa característica possibilita a retirada de código de dentro da área protegida do kernel. Os drivers de dispositivos possuem acesso a registradores de dispositivos e memória, mapeando o dispositivo no espaço de endereçamento virtual, ou via acesso controlado para portas de dispositivos no hardware Intel x86.

O gerenciamento de memória no seL4 é explícito: Objetos inerentes ao kernel e espaços de endereçamento virtual são protegidos e gerenciados através de capabilities. Através do uso de capabilities, o modelo garante que toda a alocação de memória no kernel é explícita e autorizada.

3.3.3 O processo de desenvolvimento do kernel

O processo de desenvolvimento adotado pela equipe, adapta as necessidades tanto de desenvolvedores de sistemas operacionais, que utilizam uma metodologia bottom-up para obtenção de performance, uma vez que o hardware deve ser gerenciado de forma eficiente, quanto aos responsáveis pelas provas formais, que utilizam uma metodologia top-down, pois a rastreabili-

dade das provas é determinada pela complexidade do sistema.

Para garantir esse compromisso com ambas as visões, uma abordagem intermediária foi adotada, utilizando a linguagem funcional Haskell para prover uma linguagem de programação para os desenvolvedores de Sistemas Operacionais, e ao mesmo tempo provendo um artefato que pode ser automaticamente traduzido para a ferramenta de prova de teorema.

A figura 8, extraída do paper, ilustra as iterações do processo:

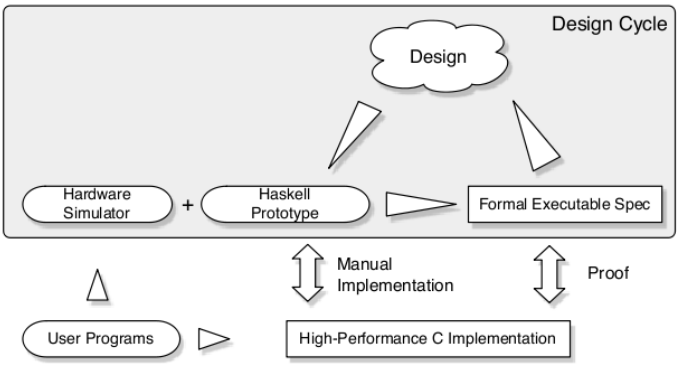

Figura 10: O processo de projeto do seL4.

Os artefatos em quadrados, são artefatos formais que contém um papel direto na prova. As setas bidirecionais representam esforço de prova ou implementação, as unidirecionais representam a influência de projeto/implementação de artefatos em outros artefatos. O artefato central é o protótipo em Haskell. Sua execução é realizada via plataforma de software que simula a plataforma em hardware. Essa estratégia foi adotada, ao invés de produzir a especificação executável diretamente no provador de teoremas, pois isto significaria uma curva de aprendizado bastante ingrime para a equipe de projeto e uma cadeia muito menos sofisticada de execução e simulação.

Foi utilizado na implementação apenas um subgrupo da linguagem Haskell, que pode ser traduzida automaticamente na liguagem utilizada pelo provador de teoremas. Os detalhes deste subgrupo podem ser encontrados em (KLEIN; DERRIN; ELPHINSTONE, 2009) e (DERRIN et al., 2006).

O protótipo em Haskell é um modelo executável e é a implementação do projeto final, porém o código é todo reescrito manualmente em C por algumas razões. A primeira delas, é que a máquina runtime de Haskell possui uma quantidade bastante significativa de código, muito maior do que o próprio microkernel, que pode ser bastante difícil de verificar. Outra é que

o runtime Haskell utiliza-se de garbage collector, o que não é adequado para ambientes de tempo real.

Além do mais, a utilização de C possibilita da implementação de baixo nível visando melhorar a performance. Seria possível uma tradução automática de Haskell para C, porém muitas micro-otimizações seriam perdidas no processo.

3.3.4 Verificação Formal

Na verificação formal é utilizado o provador de teoremas Isabelle/HOL (NIPKOW; PAULSON; WENZEL, 2002). A técnica utilizada é iterativa, auxiliada por máquina e com prova checada por máquina. Técnicas de prova iterativas necessitam de intervenção humana e criatividade para construir e guiar a prova. Formalmente o que está sendo mostrado é Refinamento (DEROEVER; ENGELHARDT, 1999): Uma prova por refinamento estabelece uma correspondência entre a representação em alto nível, com a representação em baixo nível (concreta, ou refinada) de um sistema.

A prova por refinamento garante que todas as propriedades lógicas de Hoare (sistema formal para provas rigorosas de corretude de programas computacionais) no modelo abstrato são mantidas no modelo refinado. Dessa forma, se um propriedade de segurança é provada em lógica Hoare sobre o modelo abstrato, o refinamento consegue garantir que a mesma propriedade se aplica para o código fonte do kernel.

A figura 9 mostra as camadas usadas na verificação do seL4. Elas estão relacionadas através de provas formais.

A especificação abstrata é um modelo operacional, ou seja, é a especificação completa do comportamento do sistema. Ela especifica basicamente a interface com o microkernel (systemcalls), seus argumento e os efeitos na chamada de cada uma delas, ou quando uma interrupção ou falha ocorrem. Não descreve em detalhes como é implementado no kernel.

A especificação executável é gerada a partir da implementação em Haskell traduzida para o provador de teorema (processo automatico). Essa especificação contém todas as estruturas de dados e detalhes de implementação que são esperadas na implementação final em C.

Ao final, encontra-se a implementação em C de alta performance do seL4. A verificação termina no código fonte, dessa forma o processo assume que pelo menos o compilador e o hardware funcionam de forma correta.

Resumidamente o processo se dá da seguinte forma:

1. É gerada uma especificação abstrata sobre o que o sistema faz, sem detalhar muito o como é feito.

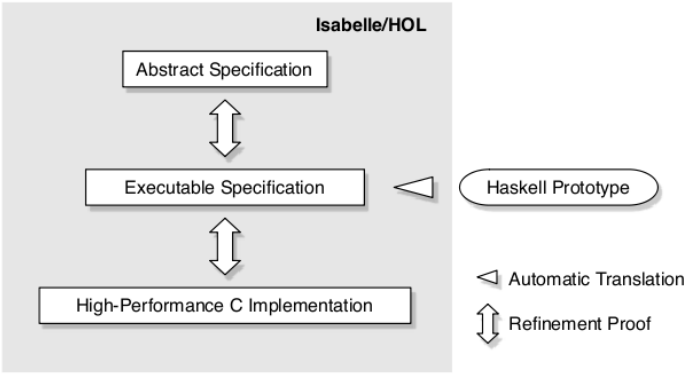

Figura 11: As camadas de refinamento na verificação do seL4.

2. Um modelo em haskell é implementado manualmente, a partir da especificação abstrata.
3. A implementação em Haskell é traduzida automaticamente para a linguagem do provador.
4. O refinamento é feito entre a especificação abstrata e a especificação gerada no passo 3, provando a equivalência entre as duas.
5. A partir desta especificação mais detalhada, a implementação de alta performance em C é realizada.
6. Da implementação em C, é gerada a especificação formal correspondente.
7. Novo refinamento é provado entre a especificação intermediária e a especificação em C.
8. Dessa forma a correlação entre a especificação abstrata e a especificação do código C é estabelecida.

3.3.5 O Projeto do Kernel para Verificação

Grande parte da prova de verificação pode ser pensado como mostrar Triplas de Hoare (principal caracteristica da lógica de Hoare, que descreve como a execução de um trecho de código altera o estado da computação) em declarações do sistema e em funções em cada nível da especificação.

Cada unidade da prova tem uma série de pré-condições que precisam ser satisfeitas antes da execução, uma sequencia de declarações em uma função que modificam o estado do sistema, e as pós condições que precisam ser mantidas após a execução. A dificuldade em mostrar que as pré e pós condições são mantidas está diretamente relacionada a complexidade que estas pré e pós condições expressam.

Para tornar a verificação do kernel viável, o projeto deve minimizar a complexidade deste componentes com foco em não perder a performance, para que o microkernel seja ideal para uso real.

Esta sessão é particulamente importante pelo fato de mostrar os componentes do sistema que tiveram sua complexidade reduzida para tornar a verificação viável. Este conhecimento pode ser reaproveitado durante a formulação da proposta de nosso mecanismo de execução segura aplicado ao HSM.

3.3.5.1 Váriaveis Globais e Efeito Colaterais

A utilização de váriaveis globais e seus efeitos colaterais é comum em kernels de sistemas operacionais. Um exemplo de efeito colateral causado com o uso de váriaveis globais pode ser observado na prática no uso de Threads concorrentes, uma influenciando no comportamento da outra. Apesar do problema dos efeitos colaterais derivado do uso de variáveis globais não ser um problema impeditivo para a verificação, seu uso pode tornar o processo mais difícil que o necessário.

O uso de variáves globais geralmente requer a declaração e prova de propriedades invariantes. Por exemplo se as filas de um scheduler são implementadas com listas duplamente encadeadas, a invariante pode dizer que todos os links para o nodo anterior apontam para o nodo apropriado e que todos os elementos apontam para o bloco de controle da Thread.

As invariantes são caras porque elas precisam ser provadas não apenas no escopo local das funções que manipulam a fila do scheduler, mas para todo o kernel. Deve ser provado que nenhuma outra manipulação de ponteiro no kernel destrói a lista ou suas propriedades.

As invariantes se tornam especialmente difíceis de provar quando são temporariamente violadas. Por exemplo no momento da adição de um nodo em uma lista duplamente encadeada, a invariante de que a lista é bem formada é violada.

3.3.5.2 Gerenciamento de Memória do Kernel

No seL4 o modelo de alocação de memória retira o controle de dentro do kernel e transfere para uma aplicação apropriadamente autorizada. Apesar da motivação inicial desta característica ser a necessidade de garantia de consumo de memória, ela também beneficia a verificação. Neste caso, apenas o mecanismo de exportação do gerenciamento de memória para aplicações autorizadas precisa ser provado, o mecanismo de gerenciamento de memória em si, deve ser provado, mas isso pode ser feito de maneira modular e pode utilizar-se de propriedades já provadas do kernel.

3.3.5.3 Concorrência e Não-determinísmo

Inicialmente o paper descreve que apesar de existirem técnicas para se provar sistemas com multiprocessadores, seu foco é em suporte a sistemas com apenas um processador, onde o problema pode ser melhor controlado. Muitos dos problemas de concorrência resultantes de Entrada e Saída (E/S), são resolvidos rodando os drivers de dispositivos a nível de usuário, porém ainda há o problema das interrupções. Três conceitos são exemplificados: Yielding, Interrupções e Exceções.

3.3.5.3.1 Yielding

É citado aqui que grande parte da complexidade da verificação do Yield é contornado utilizando-se uma execução de kernel baseada em eventos, com uma stack única de kernel e uma API quase atômica. Como o interesse primário deste trabalho é citar as técnicas utilizadas para que possam ser posteriormente adotadas, e como o assunto é razoavelmente complexo, não iremos nos aprofundar neste tópico. Deixaremos citada a referência encontrada no paper para explicações mais detalhadas (FORD et al., 1999).

3.3.5.3.2 Interrupção

A complexidade das interrupções pode ser evitada desabilitando todas elas durante a execução do kernel. Porém a latência das interrupções pode se tornar muito grande, o que foi considerado inaceitável. Ao invés de desabilitar as interrupções totalmente, elas foram desabilitadas em sua maioria, com a exceção de um pequeno e seleto grupo de pontos de interrupção. O problema

é simplificado habilitando estes pontos de interrupção via polling, ao invés de habilitados temporariamente. Este uso de pontos de interrupção cria um meio termo entre complexidade e latência.

A exceção para esta regra, é o momento da destruição de objetos. Estas operações podem ser ilimitadas no sentido da latência e são criticas para a integridade do kernel. Estas operações foram tornadas preemtivas armazenando o estado do progresso de destruição na ultima capability que referencia o objeto sendo destruído. Esta capability é chamada de zumbi. Isto garante que o correto reinício de um destroy não é dependente de registradores acessíveis pelo usuário.

Outra vantagem é que se outra Thread tentar destruir a zumbi, ela vai simplesmente continuar de onde a primeira Thread foi preemptada, ao invés de bloquear a nova Thread até que a primeira se complete.

3.3.5.3.3 Exceções

As exceções tem um efeito similar ao das interrupções. As exceções são evitadas completamente, com cuidados especiais apenas para faltas de memória (Memory Faults).

A parte de exceções no espaço de endereçamento virtual no kernel é evitado mapeando-se um região fixa do espaço de memória virtual para a memória física, independentemente se ela estiver sendo ativamente utilizada (pelo kernel) ou não. Essa região contém toda a memória que o kernel pode potencialmente vir a utilizar, garantidamente nunca produzindo uma falta. É provado que essa região aparece em cada espaço de endereçamento.

3.3.5.4 Entrada e Saída (E/S)

A maior parte da complexidade da E/S é evitada movendo os drivers de dispositivos para componentes protegidos em modo usuário. O único driver de dispositivo incluso no kernel é o driver de tempo, que gera tickets de tempo para o escalonador. Este driver é preparado na fase de inicialização do kernel e não é modificado ou acessado durante a execução. Não foi necessário modelar o timer explicitamente na prova, apenas é provado que o comportamento do sistema em cada ticket está correto.

3.3.6 Conclusão

Esta descrição breve porém um pouco mais detalhada sobre o trabalho desenvolvido no seL4, elucida as principais idéias que podem ser utilizadas como base na construção de nosso microkernel. A parte da prova foi deixada mais a segundo plano pois nossa proposta para prova difere da apresentada, enquanto o seL4 faz suas provas através de técnicas de refinamento em cima de modelos formais, pretendemos trabalhar com provas diretamente em cima da implementação.

4 PROPOSTAS PRELIMINARES

4.1 INTRODUÇÃO

Neste capítulo, pretendemos não somente expor a proposta final originada de toda a pesquisa realizada sobre a área, mas também todas as idéias intermediárias que surgiram durante o decorrer dos estudos.

Além de exemplificar e mostrar parte de toda a pesquisa desenvolvida, deixaremos registradas as propostas pois elas não são necessariamente antagônicas ou concorrentes com a proposta final.

A proposta final prevaleceu sobre as outras por se mostrar aparentemente mais robusta do ponto de vista da solução direcionada que buscávamos para o HSM, porém as outras podem ser aplicadas separadas ou em conjunto com a proposta final em trabalhos futuros, deixando o sistema ainda mais robusto.

4.2 PRIMEIRA PROPOSTA

O ponto inicial de nossas pesquisas foi: Softwares em todos os níveis não são confiáveis. Desde softwares aplicativos até softwares básicos, o usuário acaba por confiar na entidade que produz o software ou as vezes apenas o instala e utiliza por necessidade, sem fazer nenhum tipo de verificação. Várias notícias de que sistemas de código fechado enviam informações não autorizadas do usuário para o fabricante foram publicadas no decorrer dos últimos anos.

Alguns softwares possuem código aberto, o que dá uma sensação maior de segurança e confiabilidade, porém mesmo os sistemas de software de código aberto possuem falhas. Um exemplo disso, foi uma vulnerabilidade de 8 anos descoberta no kernel do linux apenas em 2009 que permitia ao atacante executar código arbitrário inserido na memória (deixo aqui a referência para um site que publicou a notícia, porém diversos podem ser encontrados http:br-linux.org2009vulnerabilidade-de-8-anos-no-kernel-linux).

O uso de código assinado utilizando certificados digitais, ajuda no sentido da autenticidade da autoridade que esta fornecendo o código, porém a segurança real do código e o que ele realiza continua sendo uma caixa preta.

Podemos começar a distinguir aqui dois aspectos importantes sobre a segurança do código. Uma é do ponto de vista que o código não contenha falhas, evitando que atacantes explorem vulnerabilidades e alterem o com-

portamento esperado do sistema. Outro aspecto é: mesmo que o código não contenha falhas, o usuário deve saber o que está sendo executado. Um código sem falhas pode perfeitamente ser malicioso no sentido de enviar informações sigilosas do usuário, como citado. Um terceiro ponto ainda seria a questão de confiabilidade no funcionamento do sistema: mesmo que o programa não contenha brechas de segurança e seu comportamento seja adequado, por problemas internos do software ele pode ser levado a um estado de inoperabilidade. Este problema também é comum.

Com esta idéia em mente, começamos as pesquisas para primariamente atingir os objetivos de especificar de uma maneira inteligível pelo usuário, o comportamento do software e garantir que o mesmo não possua falhas que levem a alteração deste comportamento. As primeiras fontes de inspiração foram os trabalhos de (NECULA, 1997) e (SEKAR et al., 2003), ambos envolvendo provas formais, porém com abordagens diferentes.

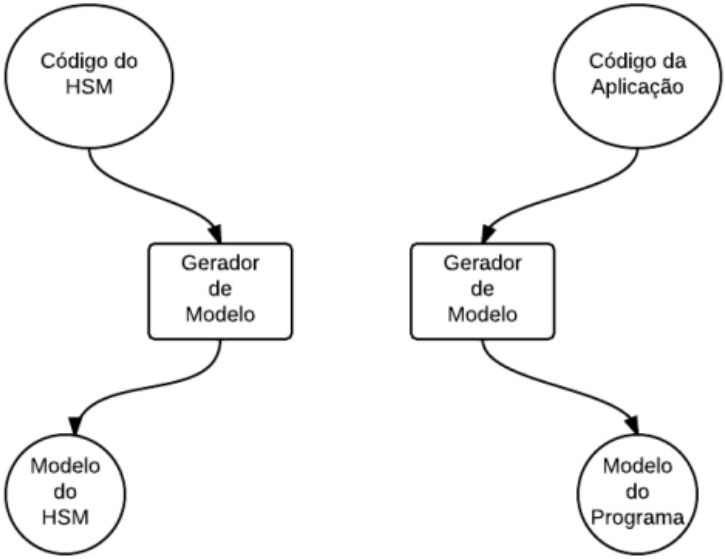

Figura 12: Processo de geração de modelo.

A idéia (bastante baseada e similar ao Model Carrying Code, apenas pensada para ser aplicada especificamente para o caso do HSM) era gerar modelos formais tanto para o código do OpenHSMd quanto para os demais programas que iriam rodar dentro do HSM (como por exemplo o SGCI, Sistema de Gerenciamento de Certificados ICP-EDU). Estes modelos formais seriam gerados a partir de casos de testes, que precisariam exaurir todos os

possíveis casos de execução do programa, gerando um grafo ou autômato do fluxo do programa. Este passo seria realizado pelo Gerador de Modelos da Figura 12, tendo como entrada o código e como saída o Modelo.

Do lado do produtor do código que rodaria dentro do HSM, o mesmo processo ocorreria, entrada do código fonte do programa no Gerador de Modelos e como saída o modelo gerado. Podemos começar a observar alguns problemas desta abordagem: O primeiro é que o produtor do código precisa gerar os modelos, ou seja, qualquer alteração do código, um novo modelo precisa ser gerado do lado do produtor e atualizado no lado do usuário.

Outro problema inerente é que o produtor precisa prevêr todos os fluxos de execução possíveis para o programa. Caso um fluxo seja deixado de fora, um erro pode ser indevidamente detectado.

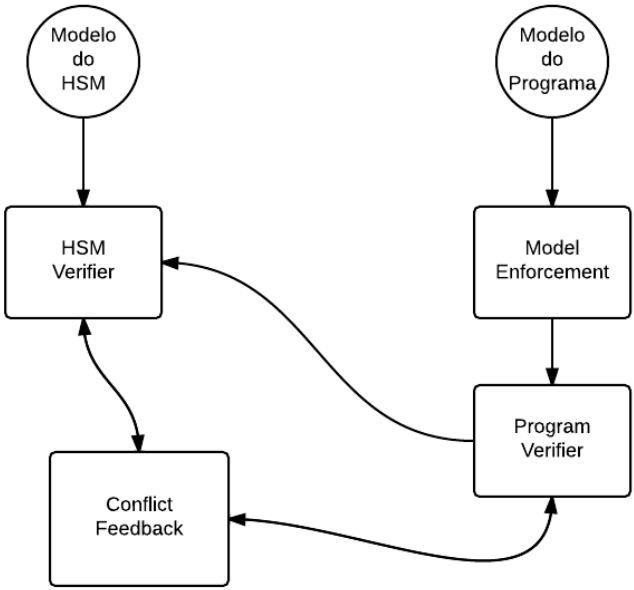

Figura 13: Fluxo de execução.

Na etapa de execução de código, teríamos o Model Enforcement. Este componente é um analisador que roda em tempo de execução juntamente com o programa. Sua função é pegar o modelo de entrada e garantir que a execução do programa não caia em um fluxo não permitido. Ele realiza as análises no nível de Systemcalls, que é uma das camadas mais críticas para

os problemas de segurança. Passando pelo Model Enforcement, teríamos o Program Verifier e o HSM Verifier, que seriam dois módulos adicionados para permitir a idéia de inserção de políticas.

A idéia principal das políticas, é por exemplo, possibilitar ou não que dois programas acessem uma chave ao mesmo tempo, e qualquer outro tipo de políticas relacionadas com o uso de chaves dentro do HSM. Foram pensandos em dois verificadores, pois a aplicação poderia ter suas próprias políticas definidas, porém o verificador do HSM teria como entrada a saída do verificador do programa, agindo como a ultima camada de filtros.

E finalmente o Conflict Feedback, seria um modulo para dar versatilidade. Caso haja um choque de políticas (claro que algumas políticas teriam que ser totalmente restritas) o usuário ainda teria a chance de inserir uma nova política no momento da execução, ou realmente abortar o programa.

Enforcement Overhead	
Interception only	**Total**
2%	30%
2%	21%
0%	2.4%

Figura 14: Enforcement overhead para o Model Carrying Code.

Esta idéia possui alguns problemas, além dos já citados durante o desenvolvimento da idéia, temos uma degradação de perfomance. Estes componentes que fazem o monitoramento da execução, fazendo a correlação com o modelo, geram uma degradação razoável, dependendo do programa. No caso do Model Carrying Code (SEKAR et al., 2003), é mostrado que este overhead pode chegar a 30%. A figura mostra os dados completos retirados do paper.

No caso do HSM, por incluir os Verificadores, este overhead poderia ser maior ainda. A idéia não é descartável e possui implementação prática no caso do Model Carrying Code, porém no decorrer dos estudos encontramos alternativas que pareceram melhores.

4.3 SEGUNDA PROPOSTA

Na continuidade dos estudos, começamos a classificar três aspectos sobre o problema global de execução segura de código:

1. Mostrar de forma inteligível ao usuário o comportamento do programa segundo sua especificação.

2. Garantir que o programa nunca execute fora dessa especificação.

3. Garantir o isolamento de memória, para que uma aplicação não interfira no comportamento da outra.

Com essa divisão, poderíamos focar melhor o problema atacado pelo trabalho, visto que o problema como um todo possuí uma complexidade muito grande para ser resolvida em um trabalho de conclusão de graduação. Destes aspectos, o que nos pareceu mais interessante de abordar, e que traria mais benefícios para o caso específico do HSM, foi o isolamento de memória.

A parte vital da segurança do HSM, é garantir que a chave privada jamais seja tocada. Por mais que a segurança do sistema como um todo seja importante, garantir a segurança da chave seria um passo importantíssimo. Resolvendo o problema de isolamento de memória, garantiríamos que uma aplicação não interferiria no funcionamento da outra, e assim, uma aplicação que fosse comprometida por alguma falha, não viria a comprometer todo o sistema.

Um problema ainda ficaria em aberto: A falha em uma única aplicação não comprometeria todas as chaves do sistema, porém uma falha em uma única aplicação ainda comprometeria sua própria chave.

Focando no primeiro problema, o isolamento de memória, uma alternativa natural foi a utilização de máquinas virtuais. Um teste bastante simples foi realizado para verificar o isolamento básico ao se utilizar máquinas virtuais:

1. Executar o OpenHSMd em uma máquina normal (Ubuntu 10.04 LTS - the Lucid Lynx).

2. Inserir dentro do OpenHSMd, um comando que imprimisse a chave, em hexadecimal, numa saída qualquer.

3. Carregar a chave para uso no OpenHSMd (imprimindo o hexa da chave).

4. Pegar privilégios de super usuário.

5. Utilizar o comando strings no arquivo /proc/kcore e direcionar a saída para um arquivo qualquer (strings /proc/kcore ¿ kcoreStrings).

6. Utilizar o comando cat no arquivo de saída procurando pela chave (cat kcoreStrings |grep chaveEmHexa).

Detalhando brevemente os passos acima: Em 1) executamos o OpenHSMd na máquina local, para realização do teste. Em 2) inserimos este print na chave em hexadecimal para facilitar o processo de captura da chave nos passos posteriores, o objetivo aqui é mostrar que o conteúdo da chave uma vez carregado, fica visível e passível de captura na memória (exitem artigos mostrando como encontrar a chave na memória em outros formatos).

Em 3) carregamos a chave no openHSMd para que ela vá para a memória, consequentemente colocando seu valor em hexa que mandamos imprimir também na memória. Em 4) pegamos privilégios de super-usuário para que seja possível ler o arquivo /proc/kcore. Em 5) direcionamos apenas os strings de /proc/kcore para um arquivo qualquer.

Os arquivos localizados em /proc não são realmente arquivos. O kernel disponibiliza na forma de arquivos para que o usuário possa acessar o conteúdo de diversos dispositivos, no caso de kcore, temos acesso a toda a memória do sistema. Em 5) confirmamos que o valor da chave se encontra nos conteúdos lidos da memória.

Ao executar o mesmo processo, porém rodando o OpenHSMd em uma máquina virtual de tipo 2 (VMWare), verificamos que a chave não se encontra no dump da memória da máquina externa.

Isso apenas nos mostra que sim, a simples utilização de uma máquina virtual já nos provê algum tipo de isolamento de memória. Alguns aspectos foram abstraídos neste exemplo para mostrar a idéia geral. As coisas não são tão simples quando realizamos o teste dentro do HSM, onde o sistema operacional é o FreeBSD e já provê um nível maior de segurança quanto ao acesso de memória interprocessos.

Outro aspecto deixado de lado, é a conta de super usuário, e que imprimimos o valor da chave em hexadecimal para facilitar nossa busca. Porém sabemos que para todos estes casos deixados de lado existem métodos mais complexos capazes de atingí-los.

Além disso não podemos confiar cegamente na simples utilização de máquinas virtuais, pois estaríamos apenas escondendo o problema em um nível mais baixo e não resolvendo-o.

Na continuação de busca para soluções dos problemas, veio a idéia de se utilizar a LLVM. Poderíamos utilizar a infraestrutura da LLVM para construir nosso mecanismo de isolamento de memória. A utilização da LLVM nos trás benefícios no sentido que a linguagem intermediária da mesma é

bem definida e independente de linguagem de programação e de arquitetura.

Para submetermos uma aplicação/plataforma a análise do nosso mecanismo, bastaria haver um front/back-end respectivamente que traduzisse/compilasse para a LLVM-IR/arquitetura alvo.

4.4 TERCEIRA PROPOSTA

Neste ponto da pesquisa, ao começar a entender mais sobre a LLVM, descobrimos a existência do projeto SafeCODE. Ao utilizar a LLVM temos a vantagem de poder, do ponto de vista técnico, reutilizar facilmente e nos beneficiarmos das melhorias de outros projetos desenvolvidos na infraestrutura. Projetos como o SafeCODE não nos dão 100% de garantia sobre a execução segura do código, apesar de proporcionar um ganho interessante de segurança, por isso pensamos em utilizar uma redundância:

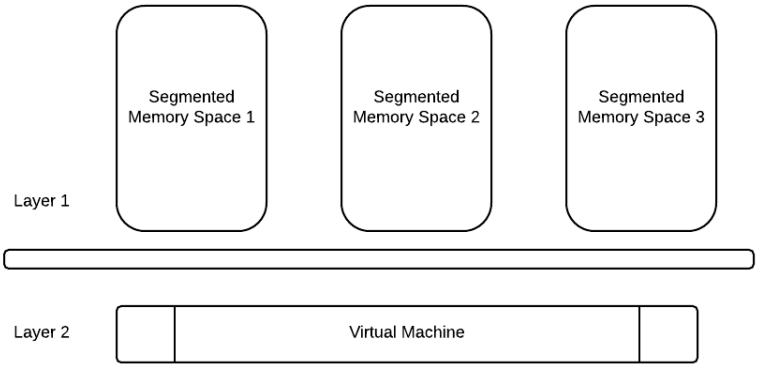

Figura 15: Arquitetura com LLVM e SafeCODE.

Na camada 1, onde teríamos as aplicações com seus sistemas operacionais virtualizados, compilaríamos o código das aplicações como o OpenHSMd com o SafeCODE, e na camada 2 a máquina virtual assegurando a restrição nos espaços de endereçamento. Caso o SafeCODE falhasse, a não interferência ficaria garantida. Ao consultar a comunidade LLVM/SafeCODE sobre a viabilidade desta alternativa, a resposta obtida foi que o próprio SafeCODE, em versões mais antigas, realizava esta função, porém não a nível de sistema operacional e que essa estrutura poderia resolver problemas que

o SafeCODE não resolve sozinho, como o fluxo de informações. O grande problema, novamente, seria o balanço entre performance e segurança.

Na busca de alternativas sobre como implementar de forma efetiva esta máquina virtual na camada 2. Chegamos a proposta final para nossa solução. Ainda nesta fase, o código do OpenHSMd foi compilado utilizando o SafeCODE e algumas necessidades de mudança foram detectadas imediatamente. Algumas das mudanças necessárias foram relatadas a equipe do OpenHSMd e foram registradas no projeto.

4.5 QUARTA PROPOSTA

Ao estudar as máquinas virtuais colocadas da forma da terceira proposta e procurando informações na literatura, chegamos aos conceitos de arquiteturas de kernel. Arquiteturas como microkernels ou exokernels podem assumir comportamentos similares a máquinas virtuais. Surgiu então a proposta de termos um microkernel na base do sistema, atuando como se fosse um hypervisor tipo 1, que garantiria o isolamento de memória.

Neste ponto focamos totalmente no problema do isolamento da memória, pois se aplica perfeitamente ao nosso contexto. O problema da execução segura de código vem sendo trabalhado amplamente ao redor do globo, até agora sem um caso de sucesso total. Nosso diferencial está no contexto em que queremos aplicar este mecanismo, o HSM. É irrefutável que a execução segura de código no HSM seria ideal, porém o propósito principal do HSM é o gerencimanto de material sensível, em nosso caso, chaves criptográficas.

Se conseguirmos garantir o isolamento de memória através do kernel, podemos colocar o OpenHSMd rodando em um espaço de memória próprio e abrir um único canal de comunicação, que é através do kernel. Qual a motivação? Verificar formalmente ou garantir que não existirão falhas na comunicação ou nos programas comunicantes é uma tarefa extremamente difícil que projetos como o SAFECode vem tentando resolver. Porém a proposta é desenvolver um microkernel com uma camada de systemcalls simples o suficiente de forma que seja passível de verificação formal.

Uma vez que a única comunicação com o OpenHSMd é via microkernel, e este microkernel é formalmente verificado, poderíamos garantir que a chave jamais seria extraída. Apenas as funções especificadas via microkernel poderiam ser realizadas sobre ela. Mesmo que todo o sistema que realiza a parte de comunicação externa fique comprometido, a chave ficaria garantidamente isolada na memória e não seria extraída.

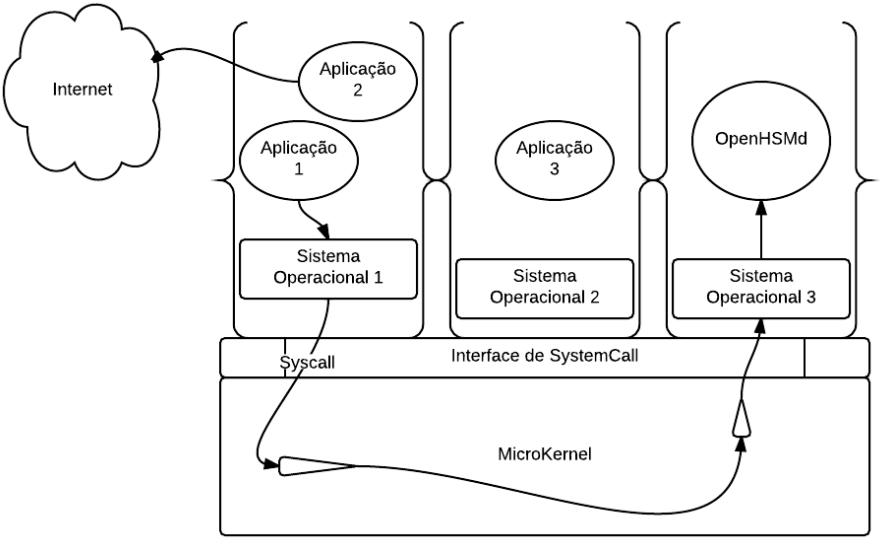

Figura 16: Visão geral da proposta com microkernel.

4.6 CONCLUSÃO

Este capítulo apresentou as propostas que surgiram durante os estudos/pesquisa sobre a concepção de um mecanismo de execução segura de código para o HSM. Como já citado, estas propostas não são necessariamente concorrentes e ficam registradas para que possam ser reavaliadas e talvez empregadas durante os trabalhos futuros do projeto.

5 SOLUÇÃO PROPOSTA

5.1 INTRODUÇÃO

Após pesquisar os fundamentos teóricos, trabalhos correlatos, propôr e estudar as possibilidades de um mecanismo de execução segura de código, a proposta 4 foi escolhida como um primeiro passo ao objetivo maior. A construção de um microkernel simples o suficiente para ser verificado formalmente é a base para construção de um sistema seguro, a exemplo do seL4 (KLEIN et al., 2009). Todas as outra técnicas e propostas intermediárias citadas neste trabalho podem ser complementares e agregar segurança ao sistema.

O foco principal desta proposta, é construir um microkernel que garanta o isolamento de memória entre as aplicações, com o objetivo de isolar o OpenHSMd e consequentemente as chaves gerenciadas pelo mesmo.

5.2 O MICROKERNEL

Inicialmente precisamos de um sistema operacional com arquitetura de microkernel. A reutilização do seL4 verificador não é possível pois o acesso a seu código fonte é fechado. Poderíamos utilizar o próprio L4 ou algum outro kernel desta família de microkernels. Porém vamos utilizar o sistema operacional EPOS, desenvolvido pelo Laboratório de Integração Software Hardware (LISHA) da UFSC.

A escolha por este sistema se dá por diversas razões, tanto técnicas quanto políticas. É um sistema desenvolvido dentro da Universidade Federal de Santa Catarina, igualmente ao OpenHSMd, e com arquitetura/código muito bem conhecidos pelo LISHA. Dessa forma ambos são soluções totalmente nacionais. Outro fator importante é que o LISHA desenvolveu uma técnica de prova formal diferenciada e já realizada sobre o escalonador do EPOS, possuindo o know-how para aplicar esta prova ao microkernel.

As provas do seL4 se tornam interessantes para serem tomadas como base para definir as propriedades que queremos provar. Igualmente importante no seL4 são os exemplos de simplificação de alguns mecanismos do Sistema Operacional de forma a manter o desempenho e facilitar o trabalho de verificação.

Os recursos básicos necessários para o microkernel, e que devem então ser mantidos no espaço de endereçamento privilegiado são: processos, me-

mória, sincronização, comunicação e encapsulamento de I/O. Possivelmente também as Capabilities devem ser mantidas dentro do espaço de endereçamento do kernel. As Capabilities mantidas dentro do espaço de endereçamento do kernel são abordadas logo abaixo.

Este microkernel fica na base do sistema como ilustrado na Figura 16 do capítulo 4, e faz o gerenciamento dos espaços de memória. Adicionalmente ele realizará a comunicação do OpenHSMd com as demais aplicações, no sentido de evitar a comunicação interprocessos e a complexidade de sua prova.

Atualmente o EPOS é entregue na forma de bibliotecas. Precisamos entregar estas mesmas funcionalidades numa arquitetura de microkernel. Após realizar esta tarefa, vamos ver um pouco sobre o suporte básico da arquitetura de hardware que integrados ao microkernel, vão nos dar base para garantir a não interferência entre processos. São eles, o duplo modo de endereçamento e a MMU.

5.2.1 Duplo Modo de Endereçamento

O duplo modo de endereçamento ou duplo modo de operação é o suporte de hardware que permite a distinção entre modo usuário e modo supervisor, ou modo super usuário. No modo usuário, instruções privilegiadas de acesso a recursos que podem vir a comprometer o sistema são limitados e realizados através de chamadas de sistema (system calls).

Esse suporte de hardware nos permite proteger a faixa de endereçamento do sistema, de um usuário normal. Dessa forma este recurso é básico e essencial para garantirmos a posterior segmentação dos espaços de endereçamento, pois dessa forma esse controle pode ser feito sem intervenção do usuário.

5.2.2 Memory Management Unit (MMU)

Seguiremos para explicação da necessidade de uma MMU, a linha descrita por (TANENBAUM, 2007). A abstração de espaços de endereçamento tornou-se necessária em função do crescimento dos programas (muito maiores do que a quantidade de memória disponível) e da necessidade do carregamento concorrente de programas na memória, sem que um interfira no funcionamento do outro.

Quando não possuímos abstração de espaços de endereçamento, os programas acessam a memória diretamente através de seus endereços físicos.

Sem nenhum mecanismo de proteção e com dois programas carregados na memória, caso o programa 1 escreva algum dado no endereço de memória 2000, ele pode estar sobrescrevendo algum dado do programa 2.

Técnicas para abstração de memória foram surgindo e evoluindo, para resolver o problema da não inteferência e da relocação (devido ao crescimento acelerado do tamanho dos programas em relação a memória disponível). Essas técnicas culminaram para o que conhecemos hoje como memória virtual.

A idéia por trás da memória virtual é que cada programa tenha o seu próprio espaço de endereçamento, que é dividido em pequenos pedaços chamados de páginas, quando é utilizada a técnica de paginação.

Cada página é um espaço contínuo de memória, e cada página é mapeada para a memória física. Quando em um sistema de memória virtual com paginação, um programa executa uma instrução como

```
MOV REG, 1000
```

uma cópia do conteúdo do endereço de memória virtual 1000 é copiado para REG. Em sistemas sem memória virtual, o endereço 1000 é colocado diretamente no barramento de memória para pegar o conteúdo da memória física 1000. Com memória virtual, esse endereço passa por uma MMU que mapeia o endereço virtual em endereços físicos. Uma visão lógica do posicionamento da MMU pode ser observado na figura abaixo:

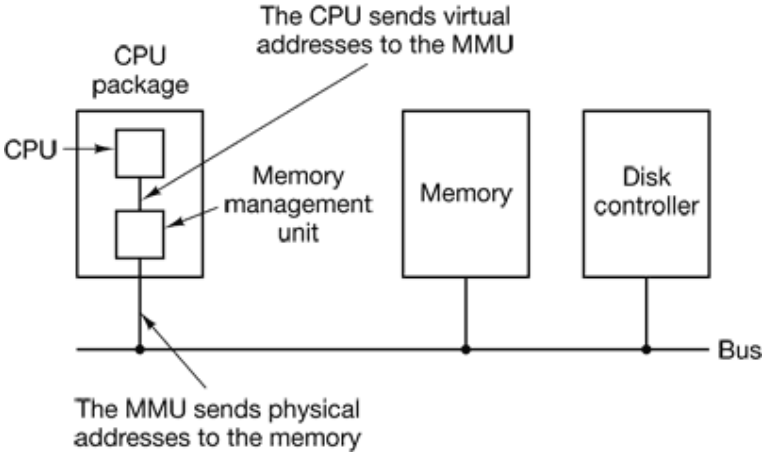

Figura 17: Esquema da MMU.

Dessa forma, a MMU é o mecanismo arquitetural comumente encontrado na maioria dos processadores atuais, tornando-se um dos fatores base para garantirmos a não interferência entre os processos.

5.3 CAPABILITIES

Uma vez tendo as memórias segmentas, capabilities são a chave para garantir o controle de acesso a todos os recursos disponibilizados para os processos isolados (I/O, alocações de novos recursos, comunicação com o kernel e com outros processos), com exceção de memória e tempo de CPU. O mecanismo de capabilities é também utilizado pelo projeto seL4, abordado anteriormente. Vamos discorrer um pouco mais sobre como funciona esse mecanismo, segundo a explicação de (TANENBAUM, 2007).

As capabilities são um modelo definido para proteção de objetos em sistemas computacionais. Objetos podem ser hardware (CPU, segmentos de memória, disco ou impressoras), ou podem ser software (processos, arquivos, bancos de dados ou semáforos).

Cada objeto possui um nome único, com o qual será referenciado e um conjunto finito de operações que processos ou usuários podem realizar sobre ele. Caracteriza-se como domínio, o conjunto de pares (objetos, permissões). Cada par especifíca um objeto e as operações que podem ser realizadas sobre ele.

O sistema precisa de alguma forma manter estas informações. Uma forma intuitiva e simples de se pensar, é através de uma matriz de proteção. Um exemplo de matriz de proteção envolvendo estes conceitos pode ser observado na figura abaixo:

	Object							
	File1	File2	File3	File4	File5	File6	Printer1	Plotter2
Domain 1	Read	Read Write						
2			Read	Read Write Execute	Read Write		Write	
3						Read Write Execute	Write	Write

Figura 18: Uma matriz de proteção.

Temos então os domínios representados como linhas e vários objetos representados como colunas. Cada posição da matriz contém as permissões que cada domínio possui sobre o objeto.

Na prática essa estrutura matricial é pouco interessante e pouco utilizada, pois muitas das posições das matrizes, onde permissões são armazenadas, são nulas, pois a grande maioria dos domínios não possuí acesso a maior parte dos objetos.

Dois métodos práticos utilizados são o armazenamento dessa matriz

por linhas ou por colunas. Nestas abordagens, apenas os elementos não nulos são armazenados. As Access Control Lists (ACL's) são um exemplo de armazenamento por coluna. No nosso caso vamos utilizar o armazenamento por linhas, que são as Capabilities.

As ACL's possuem uma política de gerenciamento ligada a usuários e grupos. As Capabilities utilizam as políticas ligadas a processos. A cada processo está ligada uma lista de objetos que podem ser acessados juntamente com uma indicação de quais operações são permitidas em cada um deles, em outras palavras, o seu domínio. Essa lista é chamada de Capabilities List (ou C-List) e os itens individuais da lista são as capabilities (Um objeto e suas permissões).

Cada capability garante ao processo certas permissões em um objeto. Um exemplo do sistema de capabilities pode ser observado na figura:

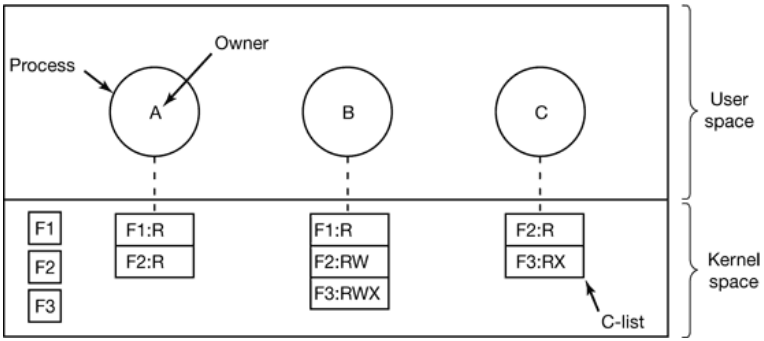

Figura 19: Cada processo possui sua lista de capabilities (C-List).

De alguma forma, as capabilities precisam ser protegidas contra adulteração por parte do usuário. Três métodos são conhecidos para este propósito:

1. Um hardware que disponibilize um bit extra em cada palavra de memória dizendo se este endereço contém uma capability ou não.

2. Manter a C-List dentro do sistema operacional (ou do espaço de endereçamento protegido).

3. Manter a C-List no espaço do usuário mas utilizando criptografia para evitar adulteração do usuário.

Abordaremos os métodos 2 e 3 que são independentes de arquitetura.

No método 2, mantendo a C-List dentro do sistema operacional, as capabilities são referenciadas pelos processos através de sua posição na Capabilities List. Um processo simplesmente solicita uma operação em um objeto, por exemplo, ”Leitura de 1KB do arquivo apontado pela capability com índice 2”. O sistema então checa na lista referente aquele processo se ele possui a permissão para realizar a operação solicitada.

O método 3 é mais direcionado para sistemas distribuídos e funciona da seguinte forma: Quando um processo cliente envia uma mensagem a um servidor de arquivos, por exemplo, para que ele crie um novo objeto para ele, o servidor cria o objeto e gera um número aleatório grande, o ”check field”. O check field é armazenado no servidor juntamente com o endereço dos blocos de disco do objeto. O check field não é enviado de volta ao usuário e nunca é colocado na rede. O servidor gera então uma capability no formato exemplificado na figura 19 e a envia de volta para o usuário.

Servidor	Objeto	Permissões	f(Objetos, Permissões, Check)

Figura 20: Uma capability protegida por criptografia

Esta capability contém o identificador do servidor, o número do objeto (o index na tabela do servidor onde o objeto foi armazenado) e as permissões. O último campo consiste da concatenação do objeto, permissões e o check field submetidos a uma função criptográfica.

Quando o usuário desejar acessar esse objeto, ele envia a caupability para o servidor como parte da requisição. O servidor extrai o index para encontrar o objeto e calcula o valor da função criptográfica utilizada tomando os dois primeiros parametros da própria capability e o terceiro de sua própria tabela (o check field que não foi retornado ao usuário e nem colocado na rede). Se o valor calculado corresponder ao valor do quarto campo da requisição enviada, a requisição é aceita, caso contrário rejeitada. Se algum outro usuário tentar acessar o objeto de outro, ele não vai ser capaz de reproduzir o quarto campo pois ele não conhece o valor do check field.

Com o uso deste método, é possível que um usuário crie uma capability somente leitura, e distribua essa capability a outro usuário para que o mesmo possa acessar o objeto. Se o outro usuário tentar alterar as permissões, o valor da função criptográfica não será verificado de forma correta e a solicitação será rejeitada.

Agora que abordamos os dois métodos, vamos fazer algumas comparações sobre ambos. Lidar com revogações de capabilities no método 2 (gerenciado pelo kernel) é complicado. É difícil para o sistema encontrar todas as capabilities enviadas sobre um dado objeto pois elas podem estar

armazenadas em C-Lists por todo o disco. Um meio de contornar isso, é que cada capability aponte para um objeto intermediário e este objeto intermediário aponte para o objeto real. Uma vez que necessite remover as capabilities sobre aquele objeto, apenas o link entre o objeto intermediário e o direto deve ser quebrado, invalidando todas as capabilities.

No método utilizando criptografia, simplesmente se altera o valor do check field armazenado junto com o objeto, invalidando o processo criptográfico e revogando todas as capabilities.

O problema é que nenhum dos dois métodos possibilita a revogação seletiva de capabilities. Este defeito é reconhecidamente um problema com todos os sistemas de capabilities.

Um problema muito relevante para a escolha no nosso caso, é que no método 3, um usuário pode simplesmente distribuir ilimitadamente uma capability válida, ou pior, pode ter essa capability válida roubada. Uma vez que alguém possua a capability válida, o acesso é liberado. O que não é nada interessante para o nosso cenário. Tendo as C-Lists gerenciadas dentro do kernel, esse problema não acontece, pois os processos apenas solicitam acesso e o próprio kernel verifica as capabilities.

Aqui foi explicado o sistema de capabilities que será utilizado no controle de acesso de aplicações via chamadas de microkernel. Este sistema pode ser utilizado inclusive para a criação de políticas de utilização de chaves, que é uma das necessidades para a evolução do projeto do HSM. O método que será adotado é o método de gerência das capabilities dentro do kernel, pois o método de criptografia pode gerar problemas com a manutenção da segurança da capability (roubo ou distribuição indevida).

5.4 VERIFICAÇÃO FORMAL

Mais importante do que a metodologia ou o ferramental que será utilizado para a verificação formal, são as definições de quais propriedades devem ser verificadas para provar a não-interferência entre processos. Porém algumas ressalvas e um comparativo com o seL4 quanto as técnicas empregadas, cabem neste contexto.

A verificação formal nos garante a correlação entre uma especificação e uma implementação, porém é difícil provar, ou verificar, a corretude da especificação. O seL4 utiliza o método de refinamento com HOL (lógica de primeira ordem) para realizar a verificação.

A técnica que iremos inicialmente adotar é o Model-Cheking, seguindo os trabalhos realizados em (LUDWICH; FROHLICH, 2012). Não serão abordados detalhes sobre o ModelChecking neste trabalho, ficam apenas registra-

dos os passos sugeridos para a realização da verificação formal. Mais detalhes sobre o método e exemplos de prova estão em (LUDWICH; FROHLICH, 2012). Seguindo esta metodologia, as provas são realizadas sem a necessidade de utilização de uma linguagem alternativa, elas são realizadas em C++, linguagem em que foi escrito o EPOS.

O estudo se encaminhará no sentido de como descrever as propriedades que desejamos verificar no formato esperado pelo Model-Checker. As propriedades que desejamos verificar, são abstratas por enquanto e bastante ligadas aos tópicos discutidos durante o trabalho:

1. Garantir que um processo seja incapaz de acessar o espaço de endereçamento de outro processo.

2. Garantir que um recurso só possa ser acessado caso seja verificada a posse de uma capability que autorize o acesso.

3. Garantir que processos do usuário não interfiram no espaço de endereçamento do kernel (garantia do duplo modo de endereçamento).

Além dessas propriedades, inicialmente iremos assumir, similarmente ao seL4, o correto funcionamento do compilador e do hardware.

5.5 CONCLUSÃO

Neste capítulo detalhamos um pouco mais alguns componentes e estratégias da solução proposta. Não há garantias de que a solução proposta vá alcançar seu objetivo, porém ela contém os próximos passos, e mostra a viabilidade para que eles sejam executados. O duplo modo de endereçamento nos serve como base para a construção do isolamento de memória, ao passo que as capabilities nos provém de uma maneira de controlar o acesso a rescursos, que pode ser pensado como um primeiro passo para a implantação de políticas de acesso dentro do HSM.

6 CONCLUSÕES E TRABALHOS FUTUROS

Ao começar este trabalho de pesquisa, tínhamos como objetivo propor uma solução para o problema de execução segura de código de modo que o HSM não perdesse seu perímetro de segurança diferenciado ao embarcar novos aplicativos, ou permitir o acesso através de softwares pela rede. Como objetivos específicos, foram listados:

1. Estudar e propôr um mecanismo de detecção (e resolução) de erros de memória em tempo de execução e compilação.

2. Estudar e propôr um mecanismo que não permita que duas aplicações concorrentes no HSM acessem uma o material sensível da outra.

3. Permitir que qualquer aplicação, independente da linguagem, possa rodar no HSM com o mecanismo de execução segura.

Buscando soluções para estes problemas, chegamos a diversas propostas intermediárias e uma solução proposta final, que cobre parte do problema para a execução segura de código no HSM. Nessa solução final foi escolhido como foco principal o problema da segmentação de memória, que no contexto do HSM é um primeiro passo valioso para garantir a não extração das chaves.

Utilizando o mecanismo de capabilities também há a possibilidade de explorarmos uma questão levantada anteriormente no HSM que é a idéia de políticas de utilização das chaves (de uma forma bastante primária). Listando especificamente os resultados obtidos para os objetivos específicos:

1. Pode ser utilizado o SAFECode para aumentar a confiabilidade na detecção e resolução de erros de memória, atentando ao fato de questões legais que não foram abordados neste trabalho. Entretando existem outras ferramentas com funcionalidade similar que não foram aqui abordadas, mas ficam registradas para pesquisas posteriores. São elas: SoftBound, BaggyBounds e Asan.

2. Foram apresentadas alternativas, porém a solução proposta (microkernel) é a que trata o problema da segmentação de memória, garantindo a não interferência entre os processos.

3. Este tópico ficou parcialmente resolvido com a utilização da infraestrutura da LLVM, com a necessidade de construção de front-ends/backends para linguagens/arquiteturas alvo. Não foi explorada uma possível

correlação com a construção do microkernel utilizando-se da infraestrutura da LLVM, porém fica também registrada aqui esta possibilidade.

Dessa forma conclui-se que este trabalho atingiu seu objetivo principal de propor uma solução inicial para o problema de execução segura de código no contexto do HSM. A segmentação de memória e construção de um microkernel verificado é com certeza a base para a construção de um sistema seguro. Na continuação deste trabalho, os próximos passos seriam:

1. Levantar os requisitos de comunicação do OpenHSMd para compatibilizar com o microkernel.
2. Verificação formal do microkernel.
3. Estudar a viabilidade de execução do OpenHSMd dentro do microkernel.

Como trabalhos futuros, ficam sugeridos:

1. Verificar a compatibilidade das propostas preliminares com a estrutura de microkernel proposta.
2. Trabalhar na verificação formal das camadas inferiores, assumidas como corretas (compilador, hardware).
3. Estudar as questões legais de se utilizar projetos como o SAFECode e afins.
4. Estudar a viabilidade de utilizar o SAFECode a partir da aplicação necessária das correções no OpenHSMd.

As limitações do trabalho são relacionadas aos riscos iniciais do projeto, de que a solução poderia ter um nível de detalhamento maior. Na parte de implementação, um protótipo do microkernel está em fase de desenvolvimento. As funcionalidades atuais do EPOS estão sendo portadas para uma arquitetura de microkernel.

7 APÊNDICE
Execução Segura de Código em Módulo de Segurança Criptográfico

Fábio Resner[1]

[1]Departamento de Informática e Estatística
Universidade Federal de Santa Catarina (UFSC) – Florianópolis, SC – Brazil
fabiusks@inf.ufsc.br

Abstract. *The safe execution of code is an open problem , and it is crucial against the leak of secret information that compromise all kinds of systems, and as consequence, their users, as individuals or entities. The big problem in guarateeing safe code execution is given by the great complexity that computational systems have reached, becoming too hard to proof or to antecipate all the possible behaviours and flaws on them. The choice of applying these techniques in a hardware security module is that it is not a general purpose system and it has less performance requisites, enabling us to think more about security over performance. The objective is to find the problems that exist while trying to implement a safe code execution mechanism, study the existent techniques and propose solutions to reach our goal.*

Resumo. *A execução segura de código é um problema em aberto, e é crucial para o não vazamento de informações sigilosas que comprometem sistemas de todo o tipo, e por consequência, seus usuários, sejam eles pessoas ou entidades. O grande problema em garantir a execução segura, se dá pela enorme complexidade que os sistemas computacionais atingiram, tornando muito difícil provar ou prevêr todos os possíveis comportamentos e falhas nos mesmos. A escolha de aplicar estas técnicas em um módulo de segurança criptográfica é que este não é um sistema de propósito geral e possui requisitos de performance muito menores do que estes, tornando possível abrir mão do desempenho para atingir um nível de segurança maior. Como objetivo, temos por delimitar os problemas enfrentados nas tentativas de implementar um mecanismo que garanta a execução segura, estudar as técnicas existentes e propôr soluções para que se alcance o resultado esperado.*

1. Introdução

A segurança, em qualquer nicho, tem se tornado um fator primordial. Na computação não é diferente. Todos os dias acompanhamos notícias sobre sistemas computacionais que foram invadidos e informações sigilosas roubadas e divulgadas. Até mesmo empresas renomadas que desenvolvem soluções para segurança computacional tem sido vítimas constantes de ataques.

Grande parte dos problemas de segurança, deve-se a evolução das redes de computadores e dos serviços oferecidos pelas mesmas, principalmente através da internet. Estes serviços já estão tão fortemente ligados ao nosso dia-a-dia, proporcionando uma praticidade tão grande, que é praticamente impossível simplesmente passar a ignorá-los tendo em vista a segurança de nossos dados. Por este motivo a segurança computacional deixou de ser um simples fator adicional e tornou-se um quesito de forte impacto.

A maior parte dos problemas de segurança computacional, do ponto de vista técnico, deve-se a código mal escrito, resultando em programas vulneráveis a ataques e programas desenvolvidos para explorar estes ataques. Estes códigos mal escritos podem estar inseridos em qualquer programa, desde a aplicação final que está sendo desenvolvida a tecnologia que está sendo utilizada assim como no sistema operacional.

Outro problema ao se falar em executar código de forma segura, além de garantir que o código não possua falhas, é especificar ao usuário (aquele que vai executar o código), qual o comportamento do programa, e conseguir provar a este usuário que o programa executa de acordo com esta especificação. Podemos notar dois conceitos diferentes: provar ao usuário o comportamento do programa e garantir que o programa sempre opere da maneira esperada, ou seja, provar que a especificação do programa está correta.

É importante destacar estes dois conceitos pois eles devem ser considerados de forma conjunta. É possível provar com exatidão que um dado código sempre vai executar de acordo com a especificação, mas provar que a especificação está correta e abrange todos os, e apenas os corretos, fluxos de execução de um programa, é algo complicado.

2. Módulo de Segurança Criptográfico

2.1. Introdução

Um HSM (Hardware Security Module) e um hardware que tem como objetivo principal gerenciar material sensível, como chaves criptográficas. O SANS institute em seu whitepaper An Overview of Hardware Se-curity Modules, define HSM como um hardware e seu respectivo software/firmware que geralmente se conecta dentro de um computador ou servidor, provendo um mínimo de funcoes criptográficas como por exemplo (mas não limitado a) cifragem, decifragem, geracao de chaves e funcoes de hash.

O instituto ainda conceitua que esses dispositivos oferecem algum tipo de proteção física e possui uma interface de usuário e uma interface programável. Os HSMs são empregados em situacoes onde o gerenciamento da a chave e vital, como por exemplo em infraestrutura de chaves públicas ou sistemas bancários online. A gestão das chaves deve ser feita pelo HSM de forma que a mesma nunca seja visível para o mundo externo, deve ser criada, utilizada e caso necessário destruída dentro do HSM.

2.2. ASI-HSM

O ASI-HSM e a plataforma alvo deste trabalho. E um HSM produzido no Brasil, possui seus componentes importados, porém a fabricação e os softwares são desenvolvidos em

parceria pela empresa Kryptus e o LabSEC (Laboratório de Seguranca em Computação) da UFSC.

O ASI-HSM e um equipamento homologado pelo Instituto Nacional de Tecnologia de Informacao (ITI) que e o orgão que realiza este processo no Brasil de acordo com as normas e requisitos contidos no documento MCT (Manual de Condutas Técnicas) 7. O ASI-HSM consiste de um sistema operacional (FreeBSD) que contém apenas as aplicacoes e bibliotecas estritamente necessárias ao seu funcionamento.

2.3. OpenHSMd

O OpenHSM e o software de gerenciamento do ASI-HSM desenvolvido pelo LabSEC. E responsável pela implementação das funcoes criptográficas que serão disponibilizadas bem como funcoes de backup, gerenciamento do material sensível e todas as funcionalidades mencionadas nos grupos de gerenciamento. Este software possui basicamente três interfaces de comunicação com o mundo externo, uma interface gráfica, um cliente texto (interfaces de administracao) e uma Engine OpenSSL.

Estas interfaces definem os possíveis acessos ao material sensível guardado pelo HSM bem como as funções disponibilizadas pelo mesmo. As interfaces também possibilitam acesso a realizacao de rotinas de configuração e administração, estas duas ultimas sendo realizadas através do cliente texto e da interface gráfica. A utilização das funcionalidades é realizada via Engine OpenSSL. Toda a comunicação das interfaces de administração é realizada com protocolo seguro SSL/TLS.

3. Model-Carrying Code

3.1. Introdução

O Model Carrying Code (MCC) foi um dos primeiros trabalhos no qual procuramos referência quando a idéia de uma mecanismo de execução segura de código foi pensado. A idéia do MCC é voltada mais para a execução segura de aplicações em dispositivos móveis, porém faz a utilização de políticas, o que era uma das idéias para o HSM, principalmente aplicadas sobre as chaves. Este capítulo detalha um pouco mais sobre o funcionamento do projeto seguindo (SEKAR et al., 2003).

3.1. A Abordagem

O framework do Model Carrying Code possuí um lado produtor e outro lado consumidor. O produtor, e responsável por extrair um modelo, que cobre todos os aspectos de segurança relevantes do código. Uma ferramenta pode ser construída para que esse processo seja realizado automaticamente. Esse modelo e então enviado ao consumidor juntamente com o código da aplicação. A figura 1 ilustra o framework:

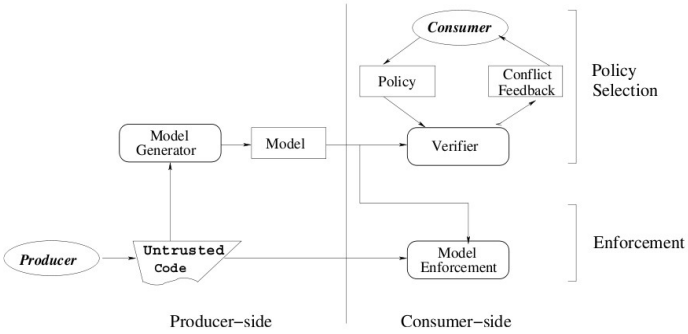

Figura 1. O Framework do Model Carrying Code

Antes de executar, o usuário pode verificar o modelo e confrontá-lo com as políticas locais de segurança. Essas políticas são definidas pelo usuário e o framework define uma linguagem para que estas políticas sejam especificadas.

O comportamento do programa é avaliado através do que foram chamados de eventos. Em ultima instância, as ações relacionadas a seguranca em um programa, são efetivadas na camada de system calls. Por esta razão as system calls constituem o alfabeto básico de eventos do Model Carrying Code.

3.2. Extração do Modelo

Os modelos são extraídos pelo consumidor utilizando o monitoramento da execução do programa. O produtor precisa ter casos de testes suficientes para cobrir todos os possíveis comportamentos do programa em tempo de execução. Caso os testes não sejam suficientes, o modelo pode não cobrir todos os comportamentos possíveis do programa e sua execução pode ser abortada durante a execução no lado do consumidor.

O extrator de modelo e constituído de dois componentes. O componente online, que intercepta as system calls e as loga juntamente com os argumentos úteis. O componente offline, constrói autômato utilizando o log gerado pelo componente online.

3.3. Verificação

A verificação é realizada antes da execução, no lado do consumidor. A verificação é realizada através das políticas definidas pelo usuário e o comportamento descrito pelo modelo. As políticas definem apenas os comportamentos proibidos, e não os comportamentos permitidos. Para permitir que o usuário readapte suas políticas, uma mensagem é apresentada ao usuário no caso de violação:

open operation on f i l e / tmp / logfile in write mode

socket operation involving the domain PF INTER

Dessa forma o usuário pode ajustar suas políticas e partir ou não para a execução do código.

3.4. Model Enforcement

O correto fluxo do programa é checado com o modelo em tempo de execução. Esse trabalho e realizado por um módulo do kernel que intercepta as system calls e verifica se os estados de execução do programa estão corretamente espelhados no modelo.

Esta é a principal fonte de overhead no Model Carrying Code. Caso o código viole o modelo, a execução é abortada. As unicas razões para que isso ocorra são o caso de o modelo estar errado ou não cobrir todos os fluxos de execução possíveis.

4. SafeCODE

O SAFECode e uma técnica de compilação desenvolvida e mais detalhadamente explicada em (DHURJATI; KOWSHIK; ADVE, 2006), para proteger os programas de erros de memória. A implementação concreta das técnicas foi realizada em um compilador, também chamado de compilador SAFECode, utilizando a estrutura da LLVM.

O intuito do SAFECode e a utilização de técnicas que permitam que o máximo de análises seja realizada em tempo de compilação (análise estática), e inserindo código para realizar análise em tempo de execução (análise dinâmica) apenas quando necessário. Estes objetivos são alcançados através do desenvolvimento de uma representação de código que permita que a maior parte das análises sejam realizadas estaticamente.

5. SeL4 Microkernel

5.1. Introdução

O seL4 é um microkernel desenvolvido dentro do projeto Trustworthy Systems (ERTOS), do Software Systems Research Group (SSRG), integrante do NICTA.

Dentro do ERTOS, é desenvolvido o seL4, Secure Microkernel Project (Projeto de Microkernel Seguro). O seL4 é fundamental para o desenvolvimento de todos os demais projetos, visto que e a unica parte de todo o software que roda no modo privilegiado do hardware. Ele é desenvolvido baseado na arquitetura da família de microkernels L4, que tem como características o tamanho compacto, alta performance e independente de política, onde independência de política significa que as políticas do Sistema Operacional não são implementadas no kernel em si mas delegadas a servidores ou outras estratégias de user-level. O seL4 ainda extende estas características com um modelo de capabilities, que provê um mecanismo para forcar as garantias de seguranca nos níveis de sistema operacional e aplicação.

A seguir o projeto e suas características serão explicados com uma maior profundidade, baseado em (KLEIN et al., 2009). A segurança e confiabilidade de um sistema computacional só pode ser tão grande quanto as do kernel do sistema operacional a qual está submetido. O kernel é definido como a parte do sistema executando no modo mais privilegiado do processador e possui acesso ilimitado ao hardware. Dessa forma qualquer problema na implementação do kernel tem o pontecial de abalar o correto funcionamento do resto do sistema.

É de senso comum que bugs sempre serão encontrados em softwares de todos os tamanhos. Dessa forma a abordagem mais comum e reduzir a quantidade de código do kernel para diminuir a exposição a bugs. Com um código pequeno o suficiente e possível garantir a não existência de bugs, através de verificação formal realizada por máquina, provendo uma prova matemática de que a implementação do kernel está condizente com a sua especificação e livre de defeitos de implementação induzidos pelo programador.

O seL4 foi projetado exatamente para oferecer esse ultimo degrau de garantia de funcionamento, entretanto ele assume o correto funcionamento do compilador, código assembly, código de boot, gerenciamento de caches e do hardware. Todo o resto é provado. Segundo o paper, o seL4 atinge os seguintes objetivos:

1. E adaptado para o uso prático, atingindo uma performance comparável com os melhores microkernels. (Neste ponto o autor se refere ao fato de que o desempenho não é degradado pelo fato de o kernel ser provado formalmente).

2. O seu comportamento é precisamente formalizado em um nível abstrato.

3. O seu projeto formal é utilizado para provar propriedades desejáveis, incluindo a segurança na finalização e execução.

4. Sua implementação é provada formalmente para satisfazer a especificação.

5. Seus mecanismos de controle de acesso são provados formalmente para provêr fortes garantias de segurança.

Os autores afirmam que a propriedade de funcionamento correto que é provada no seL4 é muito mais robusta e precisa do que técnicas automáticas como model checking, análise estática ou implementações de kernel com linguagens type-safety conseguiriam ser. Além de analisar aspectos específicos do kernel como execução segura, o seL4 fornece uma especificação completa e prova do comportamento preciso do kernel.

5.2. O Modelo de Programação do seL4

Nesta secao será apresentada uma visão das características principais do seL4. O seL4 é um microkernel de terceira geração, quer dizer, um microkernel caracterizado por uma API orientada a segurança, controle de acesso a recursos controlados por capabilities, virtualização como uma preocupação principal, técnicas inovadoras no que diz respeito a gerenciamento de recursos do kernel e um projeto de arquitetura adequado para análise formal. Isso difere bastante do desenvolvimento usual de kernels que se preocupa primariamente com performance.

O microkernel possui abstrações características para espaços de endereçamento virtual, threads, comunicação inter-processo e, diferentemente de outros kernels da família L4, utiliza o mecanismo de capabilities para autorização. Os drivers de dispositivos no seL4, como comumente em outros kernels da família L4, rodam como aplicações normais de usuário. Essa característica possibilita a retirada de código de dentro da área protegida do kernel.

Os drivers de dispositivos possuem acesso a registradores de dispositivos e memória, mapeando o dispositivo no espaço de endereçamento virtual, ou via acesso controlado

para portas de dispositivos no hardware Intel x86. O gerenciamento de memória no seL4 e explícito: Objetos inerentes ao kernel e espaços de endereçamento virtual são protegidos e gerenciados através de capabilities. Através do uso de capabilities, o modelo garante que toda a alocação de memória no kernel é explícita e autorizada.

5.3. O processo de desenvolvimento do kernel

O processo de desenvolvimento adotado pela equipe, adapta as necessidades tanto de desenvolvedores de sistemas operacionais, que utilizam uma metodologia bottom-up para obtençao de performance, uma vez que o hardware deve ser gerenciado de forma eficiente, quanto aos responsáveis pelas provas formais, que utilizam uma metodologia top-down, pois a rastreabilidade das provas é determinada pela complexidade do sistema.

Para garantir esse compromisso com ambas as visões, uma abordagem intermediária foi adotada, utilizando a linguagem funcional Haskell para provêr uma linguagem de programação para os desenvolvedores de Sistemas Operacionais, e ao mesmo tempo provendo um artefato que pode ser automaticamente traduzido para a ferramenta de prova de teorema. A figura 2, extraída do paper, ilustra as iterações do processo:

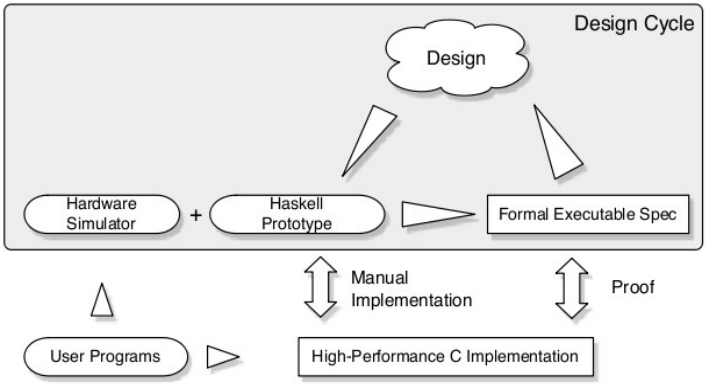

Figure 2. O processo de projeto do seL4.

Os artefatos em quadrados, são artefatos formais que contém um papel direto na prova. As setas bidirecionais representam esforço de prova ou implementação, as unidirecionais representam a influência de projeto/implementação de artefatos em outros artefatos. O artefato central é o protótipo em Haskell. Sua execução é realizada via plataforma de software que simula a plataforma em hardware. Essa estratégia foi adotada, ao invés de produzir a especificação executável diretamente no provador de teoremas, pois isto significaria uma curva de aprendizado bastante ingrime para a equipe de projeto e uma cadeia muito menos sofisticada de execução e simulação.

Foi utilizado na implementação apenas um subgrupo da linguagem Haskell, que pode ser traduzida automaticamente na liguagem utilizada pelo provador de teoremas. Os

detalhes deste subgrupo podem ser encontrados em (KLEIN; DERRIN; ELPHINSTONE, 2009) e (DERRIN et al., 2006).

O protótipo em Haskell e um modelo executável e e a implementaçãodo projeto final, porém o código é todo reescrito manualmente em C por algumas razões. A primeira delas, é que a máquina runtime de Haskell possui uma quantidade bastante significativa de código, muito maior do que o próprio microkernel, que pode ser bastante difícil de verificar. Outra e que o runtime Haskell utiliza-se de garbage collector, o que não é adequado para ambientes de tempo real. Além do mais, a utilização de C possibilita da implementação de baixo nível visando melhorar a performance. Seria possível uma tradução automática de Haskell para C, porém muitas micro-otimizações seriam perdidas no processo.

5.4. Verificação Formal

Na verificação formal é utilizado o provador de teoremas Isabelle/HOL (NIPKOW; PAULSON; WENZEL, 2002). A técnica utilizada e iterativa, auxiliada por máquina e com prova checada por máquina. Técnicas de prova iterativas necessitam de intervenção humana e criatividade para construir e guiar a prova. Formalmente o que está sendo mostrado é Refinamento (DEROEVER; ENGELHARDT, 1999): Uma prova por refinamento estabelece uma correspondência entre a representação em alto nível, com a representação em baixo nível (concreta, ou refinada) de um sistema.

A prova por refinamento garante que todas as propriedades lógicasde Hoare (sistema formal para provas rigorosas de corretude de programas computacionais) no modelo abstrato são mantidas no modelo refinado. Dessa forma, se uma propriedade de seguranca é provada em lógica Hoare sobre o modelo abstrato, o refinamento consegue garantir que a mesma propriedade se aplica para o código fonte do kernel.

A figura 3 mostra as camadas usadas na verificacao do seL4. Elas estão relacionadas através de provas formais. A especificação abstrata e um modelo operacional, ou seja, e a especificação completa do comportamento do sistema. Ela especifica basicamente a interface com o microkernel (systemcalls), seus argumento e os efeitos na chamada de cada uma delas, ou quando uma interrupção ou falha ocorrem. Não descreve em detalhes como é implementado no kernel. A especificacao executável e gerada a partir da implementação em Haskell traduzida para o provador de teorema (processo automático). Essa especificação contém todas as estruturas de dados e detalhes de implementação que são esperadas na implementacao final em C.

Ao final, encontra-se a implementação em C de alta performance do seL4. A verificação termina no código fonte, dessa forma o processo assume que pelo menos o compilador e o hardware funcionam de forma correta. Resumidamente o processo se dá da seguinte forma:

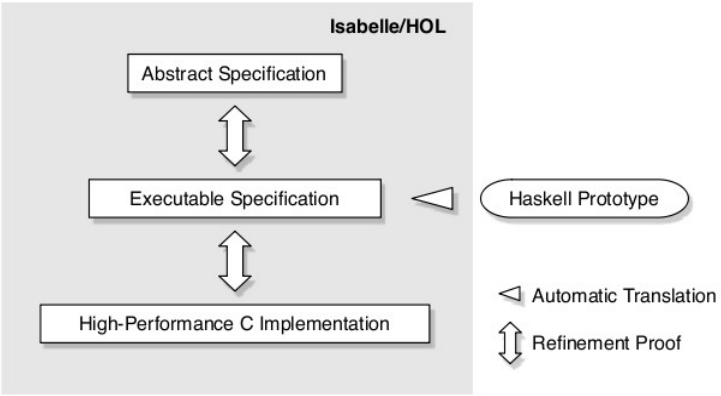

Figura 3. As camadas de refinamento na verificação do seL4.

1. É gerada uma especificação abstrata sobre o que o sistema faz, sem detalhar muito o como é feito.

2. Um modelo em haskell é implementado manualmente, a partir da especificação abstrata.

3. A implementação em Haskell é traduzida automaticamente para a linguagem do provador.

4. O refinamento é feito entre a especificação abstrata e a especificação gerada no passo 3, provando a equivalência entre as duas.

5. A partir desta especificação mais detalhada, a implementação de alta performance em C é realizada.

6. Da implementação em C, é gerada a especificação formal correspondente.

7. Novo refinamento é provado entre a especificação intermediária e a especificação em C.

8. Dessa forma a correlação entre a especificação abstrata e a especificação do código C é estabelecida.

6. Solução Proposta

6.1. Introdução

A construção de um microkernel simples o suficiente para ser verificado formalmente é a base para construção de um sistema seguro, a exemplo do seL4 (KLEIN et al., 2009). O foco principal desta proposta, é construir um microkernel que garanta o isolamento de memória entre as aplicações, com o objetivo de isolar o OpenHSMd e consequentemente as chaves gerenciadas pelo mesmo.

6.2. O Microkernel

Inicialmente precisamos de um sistema operacional com arquitetura de microkernel. A reutilização do seL4 verificado não é possível pois o acesso a seu código fonte é fechado. Poderímos utilizar o próprio L4 ou algum outro kernel desta família de microkernels. Porém vamos utilizar o sistema operacional EPOS, desenvolvido pelo Laboratório de Integração Software Hardware (LISHA) da UFSC. A escolha por este sistema se dá por diversas razões, tanto técnicas quanto políticas. É um sistema desenvolvido dentro da Universidade Federal de Santa Catarina, igualmente ao OpenHSMd, e com arquitetura/código muito bem conhecidos pelo LISHA. Dessa forma ambos são soluções totalmente nacionais.

Outro fator importante é que o LISHA desenvolveu uma técnica de prova formal diferenciada e já realizada sobre o escalonador do EPOS, possuindo o know-how para aplicar esta prova ao microkernel.

As provas do seL4 se tornam interessantes para serem tomadas como base para definir as propriedades que queremos provar. Igualmente importante no seL4 são os exemplos de simplificação de alguns mecanismos do Sistema Operacional de forma a manter o desempenho e facilitar o trabalho de verificação.

Os recursos básicos necessários para o microkernel, e que devem então ser mantidos no espaço de endereçamento privilegiado são: processos, memória, sincronização, comunicação e encapsulamento de I/O. Possivelmente também as Capabilities devem ser mantidas dentro do espaço de endereçamento do kernel. Este microkernel fica na base do sistema, e faz o gerenciamento dos espaços de memória, a arquitetura é ilustrada na figura 4:

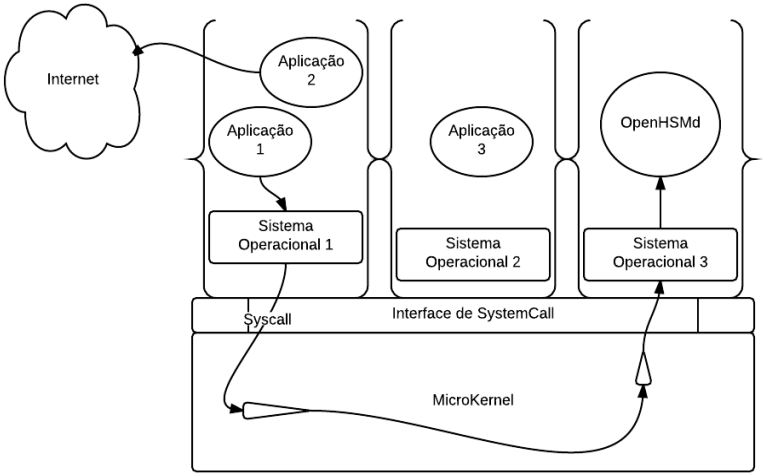

Figura 4. Visão geral da proposta com Microkernel.

Adicionalmente ele realizará comunicação do OpenHSMd com as demais aplicações, no sentido de evitar a comunicação interprocessos e a complexidade de sua prova. Atualmente o EPOS é entregue na forma de bibliotecas. Precisamos entregar estas mesmas funcionalidades numa arquitetura de microkernel. Após realizar esta tarefa, o suporte básico da arquitetura de hardware que integrados ao microkernel que vão nos dar base para garantir a não interferência entre processos são: o duplo modo de endereçamento e a MMU.

6.3. Verificação Formal

Mais importante do que a metodologia ou o ferramental que será utilizado para a verificação formal, são as definições de quais propriedades devem ser verificadas para provar a não-interferência entre processos. Porém algumas ressalvas e um comparativo com o seL4 quanto as técnicas empregadas, cabem neste contexto. A verificação formal nos garante a correlação entre uma especificação e uma implementação, porém e difícil provar, ou verificar, a corretude da especificação. O seL4 utiliza o método de refinamento com HOL (lógica de primeira ordem) para realizar a verificação.

A técnica que iremos inicialmente adotar é o Model-Cheking, seguindo os trabalhos realizados em (LUDWICH; FROHLICH, 2012). Não serão abordados detalhes sobre o ModelChecking neste trabalho, ficam apenas registrados os passos sugeridos para a realização da verificação formal. Mais detalhes sobre o método e exemplos de prova estão em (LUDWICH; FROHLICH, 2012).

Seguindo esta metodologia, as provas são realizadas sem a necessidade de utilização de uma linguagem alternativa, elas são realizadas em C++, linguagem em que foi escrito o EPOS. O estudo se encaminhará no sentido de como descrever as propriedades que desejamos verificar no formato esperado pelo Model-Checker. As propriedades que desejamos verificar, são abstratas por enquanto e bastante ligadas aos tópicos discutidos durante o trabalho:

1. Garantir que um processo seja incapaz de acessar o espaco de endereçamento de outro processo.

2. Garantir que um recurso só possa ser acessado caso seja verificada a posse de uma capability que autorize o acesso.

3. Garantir que processos do usuário não interfiram no espaco de endereçamento do kernel (garantia do duplo modo de endereçamento).

Além dessas propriedades, inicialmente iremos assumir, similarmente ao seL4, o correto funcionamento do compilador e do hardware.

7. Conclusões e Trabalhos Futuros

Ao comecar este trabalho de pesquisa, tínhamos como objetivo propor uma solução para o problema de execução segura de código de modo que o HSM não perdesse seu perímetro de segurança diferenciado ao embarcar novos aplicativos, ou permitir o acesso através de softwares pela rede. Como objetivos específicos, foram listados:

1. Estudar e propôr um mecanismo de detecção (e resolução) de erros de memória em tempo de execução e compilação.

2. Estudar e propôr um mecanismo que não permita que duas aplicações concorrentes no HSM acessem uma o material sensível da outra.

3. Permitir que qualquer aplicação, independente da linguagem, possa rodar no HSM com o mecanismo de execução segura.

Buscando soluções para estes problemas, chegamos a diversas propostas intermediárias e uma solução proposta final, que cobre parte do problema para a execução segura de código no HSM. Nessa solução final foi escolhido como foco principal o problema da segmentação de memória, que no contexto do HSM é um primeiro passo valioso para garantir a não extração das chaves. Utilizando o mecanismo de capabilities também há possibilidade de explorarmos uma questão levantada anteriormente no HSM que e a idéia de políticas de utilização das chaves (de uma forma bastante primária). Listando especificamente os resultados obtidos para os objetivos específicos:

1. Pode ser utilizado o SAFECode para aumentar a confiabilidade na detecção e resolução de erros de memória, atentando ao fato de questões legais que não foram abordados neste trabalho. Entretando existem outras ferramentas com funcionalidade similar que não foram aqui abordadas, mas ficam registradas para pesquisas posteriores. São elas: Soft-Bound, BaggyBounds e Asan.

2. Foram apresentadas alternativas, porém a solucao proposta (microkernel) e a que trata o problema da segmentação de memória, garantindo a não interferência entre os processos.

3. Este tópico ficou parcialmente resolvido com a utilização da infraestrutura da LLVM, com a necessidade de construção de front-ends/back-ends para linguagens/arquiteturas alvo. Não foi explorada uma possível correlação com a construção do microkernel utilizando-se da infraestrutura da LLVM, porém fica também registrada aqui esta possibilidade.

Dessa forma conclui-se que este trabalho atingiu seu objetivo principal de propor uma solução inicial para o problema de execução segura de código no contexto do HSM. A segmentação de memória e construção de um microkernel verificado é com certeza a base para a construção de um sistema seguro. Na continuação deste trabalho, os próximos passos seriam:

1. Levantar os requisitos de comunicação do OpenHSMd para compatibilizar com o microkernel.

2. Verificação formal do microkernel.

3. Estudar a viabilidade de execução do OpenHSMd dentro do microkernel.

Como trabalhos futuros, ficam sugeridos:

1. Verificar a compatibilidade das propostas preliminares com a estrutura

de microkernel proposta.

2. Trabalhar na verificação formal das camadas inferiores, assumidas como corretas (compilador, hardware).

3. Estudar as questões legais de se utilizar projetos como o SAFECode e afins.

4. Estudar a viabilidade de utilizar o SAFECode a partir da aplicação necessária das correções no OpenHSMd.

References

ADVE, V. SAFECode. 2012. <http://safecode.cs.illinois.edu/index.html>.

ATTRIDGE, J. InfoSec Reading Room An Overview of Hardware Security Modules. 2002.

DHURJATI, D.; KOWSHIK, S.; ADVE, V. Safecode: enforcing alias analysis for weakly typed languages. In: PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation. New York, NY, USA: ACM, 2006. p. 144–157. ISBN 1-59593-320-4.

KLEIN, G. The L4.verified project - next steps. In: LEAVENS, G.; O'HEARN, P.; RAJAMANI, S. (Ed.). Proceedings of Verified Software: Theories, Tools and Experiments 2010. Edinburgh, UK: Springer-Verlag, 2010. (Lecture Notes in Computer Science, v. 6217), p. 86–96.

KLEIN, G.; DERRIN, P.; ELPHINSTONE, K. Experience report: seL4 — formally verifying a high-performance microkernel. In: Proceedings of the 14th International Conference on Functional Programming. Edinburgh, UK: ACM, 2009. p. 91–96.

KLEIN, G. et al. sel4: formal verification of an os kernel. In: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles. New York, NY, USA: ACM, 2009. (SOSP '09), p. 207–220. ISBN 978-1-60558-752-3. <http://doi.acm.org/10.1145/1629575.1629596>.

LUDWICH, M. K.; FROHLICH, A. A. System-Level Verification of Embedded Operating Systems Components. 2012.

SEKAR, R. et al. Model-carrying code: a practical approach for safe execution of untrusted applications. SIGOPS Oper. Syst. Rev., ACM, New York, NY, USA, v. 37, n. 5, p. 15–28, out. 2003. ISSN 0163-5980. <http://doi.acm.org/10.1145/1165389.945448>.

REFERÊNCIAS BIBLIOGRÁFICAS

ADVE, V. *SAFECode*. 2012. <http://safecode.cs.illinois.edu/index.html>.

ANLEY, C. et al. *The Shellcoder's Handbook: Discovering and Exploiting Security Holes*. John Wiley & Sons, 2011. ISBN 9781118079126. <http://books.google.com.br/books?id=I0JtgPuXXT8C>.

ATTRIDGE, J. InfoSec Reading Room An Overview of Hardware Security Modules. 2002.

BEREZA JÚNIOR, A. Aprimoramento de um hsm para homologacao na icp-brasil.

BLUM, R. *Professional Assembly Language*. John Wiley & Sons, 2005. (Wrox professional guides). ISBN 9780764595615. <http://books.google.com.br/books?id=Bl0_hWV20TAC>.

DEROEVER, W.; ENGELHARDT, K. *Data Refinement: Model-Oriented Proof Methods and Their Comparison*. New York, NY, USA: Cambridge University Press, 1999. ISBN 0521641705.

DERRIN, P. et al. Running the manual: an approach to high-assurance microkernel development. In: *Proceedings of the 2006 ACM SIGPLAN workshop on Haskell*. New York, NY, USA: ACM, 2006. (Haskell '06), p. 60–71. ISBN 1-59593-489-8. <http://doi.acm.org/10.1145/1159842.1159850>.

DHURJATI, D.; KOWSHIK, S.; ADVE, V. Safecode: enforcing alias analysis for weakly typed languages. In: *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*. New York, NY, USA: ACM, 2006. p. 144–157. ISBN 1-59593-320-4.

DOWD, M.; MCDONALD, J.; SCHUH, J. *The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities*. Pearson Education, 2006. ISBN 9780132701938. <http://books.google.com.br/books?id=t2yA8vtfxDsC>.

FORD, B. et al. Interface and execution models in the fluke kernel. In: *In Proceedings of the third symposium on Operating systems design and implementation*. [S.l.]: USENIX Association, 1999. p. 101–115.

KLEIN, G. The L4.verified project - next steps. In: LEAVENS, G.; O'HEARN, P.; RAJAMANI, S. (Ed.). *Proceedings of Verified Software: Theories, Tools and Experiments 2010*. Edinburgh, UK: Springer-Verlag, 2010. (Lecture Notes in Computer Science, v. 6217), p. 86–96.

KLEIN, G.; DERRIN, P.; ELPHINSTONE, K. Experience report: seL4 — formally verifying a high-performance microkernel. In: *Proceedings of the 14th International Conference on Functional Programming*. Edinburgh, UK: ACM, 2009. p. 91–96.

KLEIN, G. et al. sel4: formal verification of an os kernel. In: *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. New York, NY, USA: ACM, 2009. (SOSP '09), p. 207–220. ISBN 978-1-60558-752-3. <http://doi.acm.org/10.1145/1629575.1629596>.

LATTNER, C. The architecture of open source applications. In: ____. [S.l.: s.n.], 2012. cap. LLVM.

LATTNER, C. *Low Level Virtual Machine*. 2012. <http://www.llvm.org>.

LATTNER, C.; ADVE, V. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In: *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*. Palo Alto, California: [s.n.], 2004.

LUDWICH, M. K.; FROHLICH, A. A. System-Level Verification of Embedded Operating Systems Components. 2012.

MARTINA, J. E. *Projeto de um Provedor de Serviços Criptográficos Embarcado para Infra-estrutura de Chaves Públicas e suas Aplicações Criptográficas no OpenHSM*. Dissertação (Mestrado) — Universidade Federal de Santa Catarina, 2005.

NECULA, G. C. Proof-carrying code. In: *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. New York, NY, USA: ACM, 1997. (POPL '97), p. 106–119. ISBN 0-89791-853-3. <http://doi.acm.org/10.1145/263699.263712>.

NIPKOW, T.; PAULSON, L. C.; WENZEL, M. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. [S.l.]: Springer, 2002. (LNCS, v. 2283).

ONE, A. Smashing The Stack For Fun And Profit. *Phrack*, v. 7, n. 49, nov. 1996. <http://phrack.com/issues.html?issue=49&id=14#article>.

OPENSSL. *OpenSSL*. 2012. <http://www.openssl.org/>.

PATTERSON, D.; HENNESSY, J. *Computer Organization and Design: The Hardware/Software Interface*. Elsevier Morgan Kaufmann, 2009. (The Morgan Kaufmann Series in Computer Architecture and Design). ISBN 9780123744937. <http://books.google.com.br/books?id=3b63x-0P3_UC>.

Rick Lopes de Souza. Modificações no provedor de serviço criptográfico openhsm para atender ambientes de alta demanda.

SEKAR, R. et al. Model-carrying code: a practical approach for safe execution of untrusted applications. *SIGOPS Oper. Syst. Rev.*, ACM, New York, NY, USA, v. 37, n. 5, p. 15–28, out. 2003. ISSN 0163-5980. <http://doi.acm.org/10.1145/1165389.945448>.

SOUZA, T. C. S. de. *Aspectos Técnicos e Teóricos da Gestão do Ciclo de Vida de Chaves Criptográficas no OpenHSM*. Dissertação (Mestrado) — Universidade Federal de Santa Catarina, 2008.

TANENBAUM, A. S. *Modern Operating Systems*. 3rd. ed. Upper Saddle River, NJ, USA: Prentice Hall Press, 2007. ISBN 9780136006633.

WIKIPEDIA. *SQL Injection*. 2012. <http://en.wikipedia.org/wiki/SQL_injection>.