OSCAR ZIBORDI DE PAIVA

**ARQUITETURA DE APLICATIVOS MÓVEIS COM
FLUXO SEGURO DE INFORMAÇÃO**

Dissertação apresentada à Escola Politécnica da Universidade de São Paulo para obtenção do Título de Mestre em Ciências

São Paulo
2016

OSCAR ZIBORDI DE PAIVA

**ARQUITETURA DE APLICATIVOS MÓVEIS COM
FLUXO SEGURO DE INFORMAÇÃO**

Dissertação apresentada à Escola Politécnica da Universidade de São Paulo para obtenção do Título de Mestre em Ciências

Área de Concentração:
Engenharia de Computação

Orientador:
Prof. Dr. Wilson Vicente Ruggiero

São Paulo
2016

Este exemplar foi revisado e corrigido em relação à versão original, sob responsabilidade única do autor e com a anuência de seu orientador.

São Paulo, _____ de _____ de _____

Assinatura do autor: _____

Assinatura do orientador: _____

Catálogo-na-publicação

de Paiva, Oscar Zibordi

Arquitetura de Aplicativos Móveis com Fluxo Seguro de Informação / O. Z. de Paiva -- versão corr. -- São Paulo, 2016.

154 p.

Dissertação (Mestrado) - Escola Politécnica da Universidade de São Paulo. Departamento de Engenharia de Computação e Sistemas Digitais.

1.Segurança da Informação 2.Controle de Fluxo de Informação 3.Aplicativos Móveis I.Universidade de São Paulo. Escola Politécnica. Departamento de Engenharia de Computação e Sistemas Digitais II.t.

Aos meus pais, Carlos e Ana, e ao
meu irmão, David

AGRADECIMENTOS

Ao meu orientador, Prof. Dr. Wilson Vicente Ruggiero, pelos inúmeros ensinamentos e oportunidades que me deu, e pela confiança depositada em mim.

Aos meus pais, que tanto me apoiaram em todos os momentos da minha vida.

A todos os amigos na Scopus Soluções pelo apoio e incentivo, além dos debates que levaram ao desenvolvimento deste trabalho.

RESUMO

A adoção de lojas de aplicativos e *Open APIs* por um número crescente de empresas, muitas nem mesmo atuantes no ramo de tecnologia, revela o interesse das mesmas em exteriorizar a concepção e desenvolvimento de software corporativo. Com isso, as empresas almejam multiplicar as funcionalidades disponíveis a seus clientes, utilizando uma fração do custo e do tempo que seriam tradicionalmente gastos para fazê-lo. Ao mesmo tempo, o acesso a dados e sistemas corporativos por softwares de desenvolvedores potencialmente desconhecidos suscita preocupações de segurança, tornando-se imperativo garantir a adequação desses softwares às políticas de segurança institucionais. Entretanto, carece-se de meios automáticos capazes de garantir a mencionada adequação nas plataformas móveis, seja nos seus ambientes de execução ou em seus kits de desenvolvimento de software. Este trabalho, utilizando de ideias recentes da área de *Controle de Fluxo de Informação*, propõe a arquitetura de um ambiente de execução para aplicativos móveis que garante por construção a adequação dos mesmos a determinadas políticas de confidencialidade e integridade de dados, mesmo na presença de código malicioso. A praticidade de tal arquitetura é validada através da implementação de um aplicativo exemplo. Tal implementação ilustra o funcionamento dos mecanismos de segurança propostos e a compatibilidade dos mesmos a um conjunto de funcionalidades adequado ao cenário de manipulação de dados corporativos.

ABSTRACT

The adoption of application stores and *Open APIs* by a growing number of companies, many of them not even related to the technology business, reveals their interest in externalizing the conception and development of corporate software. By doing so, these companies expect to multiply the number of functionalities available to their customers, spending a fraction of the traditionally required time and cost. On the other hand, access to corporate data and services by software developed by potentially unknown parties raises security concerns, making it imperative to ensure the adequacy of the mentioned software to the institutional security policies. Nevertheless, there is a lack of automatic tools capable of guaranteeing the mentioned adequacy in mobile platforms, either in their runtime environments or in their software development kits. This work, using recent ideas from the *Information Flow Control* area, proposes the architecture of a run-time environment for mobile applications that guarantees by construction their adequacy to some confidentiality and integrity policies, even in the presence of malicious code. The practicality of this architecture is validated by the implementation of an example application. This implementation illustrates the working of the proposed security mechanisms and their compatibility to a set of functionalities relevant to the scenario of corporate data manipulation.

SUMÁRIO

Lista de Ilustrações

Lista de Tabelas

Lista de Abreviaturas e Siglas

Lista de Símbolos

1	Introdução	14
1.1	Motivação	15
1.1.1	Controle de Fluxo de Informação	16
1.1.2	Limitações encontradas na literatura	18
1.2	Objetivos	20
1.3	Contribuições	20
1.4	Organização do Texto	21
2	Aspectos Teóricos: Controle de Fluxo de Informação	22
2.1	Canais de Comunicação e a Implementação de Mecanismos IFC	27
2.1.1	Canais Legítimos	27
2.1.2	Canais Encobertos	31
2.1.2.1	Fluxos Implícitos	32
2.1.2.2	Canais de Recurso Compartilhado	36

2.1.2.3	Canais de Temporização	38
2.1.2.4	Canais de Terminação	42
2.1.2.5	Canais Físicos	43
2.2	Não-Interferência e Desclassificação	44
2.3	Modelos de Políticas de Fluxo de Informação	50
2.3.1	Modelo de Bell-LaPadula	50
2.3.2	Modelo de Biba	51
2.3.3	Modelo Descentralizado	52
2.3.4	Rótulos de Categorias Disjuntas	53
2.4	Considerações finais	55
3	Trabalhos Relacionados	59
3.1	IFC em Aplicações Móveis	59
3.1.1	Maxoid	62
3.1.2	π Box	64
3.1.3	TaintDroid	66
3.1.4	FlowDroid	68
3.1.5	Análise Comparativa	68
3.2	IFC em Aplicações Web	70
3.2.1	IFC em Web-Browsers	72
3.2.2	IFC em Servidores de Aplicações Web	75
3.2.2.1	Asbestos Ok Web Server	79

3.2.2.2	SafeWeb	82
3.2.2.3	SIF/Swift	83
3.2.2.4	Hails	85
3.2.2.5	Análise Comparativa	87
4	Proposta de Arquitetura	89
4.1	Política de Segurança	92
4.2	Arranjo no Servidor	95
4.3	Arranjo no Aplicativo-base	100
4.3.1	O componente de WebView	101
4.3.2	IFC dinâmico em WebViews	103
4.3.3	Label Creep, layer apartado e seccionamento de aplicativos	106
4.3.4	Layer de sanitização	108
4.4	Implementação	110
4.4.1	Android	110
4.4.2	Android WebView	113
4.4.3	Estrutura e Operação do Aplicativo-Base	115
4.5	Funcionamento Integrado	117
4.5.1	Inicialização	119
4.5.2	Recomendação	121
4.5.3	Check-In e Pagamento	124

5	Análise da Proposta	126
5.1	Segurança	126
5.1.1	Limitações do Atacante	127
5.1.2	Garantias de Não-Interferência	129
5.1.2.1	Controle dos canais de temporização e terminação	131
5.1.3	Segurança das operações de desclassificação	134
5.2	Desempenho	136
5.3	Programabilidade	138
5.4	Comparação com Trabalhos Relacionados	141
6	Considerações Finais e Trabalhos Futuros	143
6.1	Publicações	145
6.2	Trabalhos Futuros	145
	Referências	148

LISTA DE ILUSTRAÇÕES

1	Exemplo de operação de um mecanismo IFC genérico	29
2	Fluxo implícito entre processos (configuração inicial)	35
3	Fluxo implícito entre processos (configuração intermediária) . . .	35
4	Fluxo implícito entre processos (configuração final)	35
5	Exemplo de uso de um canal de recurso compartilhado	37
6	Arquitetura Conceitual de um Web server equipado com IFC . .	76
7	Diagrama de Hasse da relação “pode-fluir-para” ($\sqsubseteq$)	93
8	Arquitetura do Web Server da loja de aplicativos	96
9	Arquitetura conceitual do WebView	103
10	Disposição da interface, mostrando a área reservada à Web-View, sobre a qual o layer apartado é desenhado. No exemplo, 3 botões são exibidos no layer.	108
11	Disposição da interface, mostrando a área reservada à Web-View, o layer apartado e o layer de sanitização, exibindo uma mensagem de confirmação	109
12	Arquitetura do Android	111
13	Diagrama E-R dos dados fornecidos às aplicações de terceiro .	118
14	Tela de configurações de privacidade para o aplicativo	118
15	Diagrama E-R dos dados do aplicativo de recomendação de restaurantes	119

16	Tela inicial do aplicativo de recomendações, sendo executado na implementação Android do aplicativo-base	120
17	Exemplo de interação de módulos na arquitetura (inicialização) .	120
18	Tela da funcionalidade de recomendação	122
19	Exemplo de interação de módulos na arquitetura (recomendação)	122
20	Exemplo de interação de módulos na arquitetura	123
21	Telas da funcionalidade de check-in e pagamento	124
22	Layer de sanitização exibido na funcionalidade de pagamento .	125
23	Exemplo de interação de módulos na arquitetura (pagamento) .	125

LISTA DE TABELAS

1	Resumo dos conceitos apresentados neste capítulo	56
2	Comparação dos trabalhos selecionados de IFC em aplicações móveis	69
3	Comparação dos trabalhos selecionados sobre IFC em servidores de aplicação Web	87
4	Matriz de controle de acesso da WebView, definida em função da classe de segurança associada à mesma	105

LISTA DE ABREVIATURAS E SIGLAS

API Application Programming Interface

App Aplicativo

GPS Global Positioning System

IFC Information Flow Control

SQL Structured Query Language

TCB Trusted Computing Base

USP Universidade de São Paulo

XSS Cross-Site Scripting

Web World Wide Web

LISTA DE SÍMBOLOS

SC	Conjunto de classes de segurança
$\sqsubseteq$	Relação “pode-fluir-para” ¹
$\sqcup$	Operação de combinação de classes ²
O	Conjunto de objetos de um sistema
o_i	Elemento de O
$l(o_i)$	Rótulo de segurança de o_i
VN	Conjunto de nomes de variáveis
m	Configuração de memória
LM	Low-memory
LER	Low-equivalence relation
N	Conjunto de níveis de segurança
R	Conjunto de rótulos de segurança
r	Mapeamento de rótulos a objetos
DOM	Relação em R

¹“ $\rightarrow$ ” segundo a notação de Denning (1976)

²“ $\oplus$ ” segundo a notação de Denning (1976)

1 INTRODUÇÃO

O sucesso de plataformas móveis como *Android* e *iOS* está intimamente associado à abundância e a diversidade dos aplicativos disponíveis para as mesmas, e a participação da comunidade global de desenvolvedores na criação desses foi essencial para fazê-lo de forma rápida e com custos reduzidos. Tendo a possibilidade de divulgar e comercializar amplamente seus softwares nas chamadas *lojas de aplicativos* das plataformas, essa comunidade construiu um conjunto de aplicativos numeroso, que adiciona funcionalidades diversas aos dispositivos equipados com tais plataformas e os torna mais atrativos para potenciais consumidores. De forma retroalimentada, a maior atratividade para consumidores torna os dispositivos também mais atrativos para os desenvolvedores. Como resultado, as lojas de aplicativos *Google Play* e *iTunes* (das plataformas *Android* e *iOS*, respectivamente) oferecem, no momento de escrita deste texto, cerca de 2 milhões de aplicativos cada, números atingidos a uma taxa de aproximadamente 300 mil novos aplicativos por ano (STATISTA, 2015).

A recente ascensão do número de API's abertas e de lojas de aplicativos corporativas revela a intenção de empresas, algumas nem mesmo atuantes no ramo de tecnologia, de também criar aplicativos utilizando a criatividade e a mão-de-obra da comunidade global de desenvolvedores. Com isso, as empresas obtêm vantagens competitivas ao disponibilizar novas formas de se interagir com os produtos e serviços por elas oferecidos, frente a um cenário

contemporâneo de crescente demanda por funcionalidades de software. Os desenvolvedores, por sua vez, obtêm o benefício da divulgação e a disponibilização de seus softwares para os clientes das empresas.

Entretanto, essa prática suscita questões de segurança, pois pressupõe a externalização da atividade de desenvolvimento de software corporativo, bem como a potencial desformalização da mesma. Assim, ao fazê-lo, empresas colocam em risco principalmente a confidencialidade e a integridade dos dados manipulados por aplicativos de terceiros. O comprometimento da confidencialidade desses dados pode acarretar em prejuízos decorrentes de obrigações legais ou de danos à imagem institucional (por exemplo, quando se divulgam dados sigilosos de clientes, como fichas médicas ou históricos bancários), ou mesmo de perdas de vantagem competitiva (quando são divulgados dados estratégicos da empresa, por exemplo). O comprometimento da integridade de dados, por sua vez, pode acarretar tanto na manipulação fraudulenta de ativos corporativos ou de clientes, quanto em efeitos na disponibilidade dos sistemas institucionais, dentre outros prejuízos.

1.1 Motivação

Muito embora haja a necessidade de se preservar a confidencialidade e a integridade de dados corporativos no cenário descrito, os mecanismos de segurança presentes nas plataformas móveis atualmente são insuficientes para fazê-lo. Esses mecanismos consistem principalmente de um *sandbox* de execução e um conjunto de permissões que, atribuídas diretamente pelo usuário a um aplicativo, ditam o conjunto de recursos (ou ações) que esse último é autorizado a acessar (ou executar) no sistema. Entretanto, esse conjunto pode conferir ao aplicativo uma capacidade de comunicação que independe do sigilo dos dados por ele recebidos ou transmitidos, fato que possibilita a

perpetração de abusos de privacidade (ENCK, 2010). No cenário em questão, considerando-se que os aplicativos de terceiros necessitem utilizar de algum mecanismo de comunicação para acessar, no mínimo, os dados provenientes de sistemas corporativos, esse mecanismo pode ser igualmente utilizado para violar a privacidade de tais dados. Além disso, com os mecanismos de segurança usuais das plataformas móveis, nada pode se garantir quanto à integridade dos dados submetidos aos sistemas corporativos.

1.1.1 Controle de Fluxo de Informação

Com o intuito de aprimorar a segurança de dados em plataformas móveis, alguns trabalhos¹ propõem a utilização de mecanismos de *controle de fluxo de informação* (ou mecanismos IFC, da sigla em inglês de *information flow control* (LAMPSON, 2015)). Um mecanismo desse tipo tem o propósito de regular a transmissão de informação num sistema computacional, operando essencialmente com base na rotulagem de objetos do sistema com as categorias de informação a eles associadas. Assim, em termos simplificados, objetos que representam pontos de entrada de informação no sistema (ou de saída de informação dele) recebem rótulos fixos, ao passo que os demais objetos recebem rótulos modificáveis. À medida que determinada informação propaga-se no sistema através de seus objetos, o mecanismo atualiza os rótulos dos mesmos, de forma a se evitar que algum deles a envie a um ponto de saída de rótulo incompatível. Considere-se um exemplo com duas categorias de informação, pública e privada, representadas pelos rótulos *h* e *l*, de *high* e *low*. Um mecanismo IFC associa rótulo *h* aos objetos que leem informações de outros objetos com rótulos também *h*, sejam esses últimos objetos pontos de origem de informações ou não. Objetos com rótulo *h* são então impedidos

¹Como os de Arzt (2014), Enck (2010), Lee (2013) e Xu e Witchel (2015)

de enviar informações a pontos de saída de rótulo *l*.

Um mecanismo IFC pode ser implementado em diferentes camadas de um sistema computacional, sendo o sistema operacional, sistemas de tempo de execução (ou de middleware) e compiladores as camadas tradicionalmente selecionadas para tanto. Nos dois primeiros casos, o mecanismo age como um monitor de execução (VOLPANO, 1999), regulando fluxos de informação à medida em que os sujeitos do sistema os efetuam. No último caso, o mecanismo atua como um verificador de adequação dos sujeitos, analisando o comportamento dos mesmos – através do código fonte, usualmente – antes que eles sejam executados. A depender dessa camada de implementação do mecanismo, determina-se o conjunto de possíveis unidades mínimas detentoras de informação, às quais o mecanismo associa rótulos únicos (como processos, threads, segmentos de memória, variáveis, arquivos, conexões de rede, periféricos e outros).

As garantias de segurança fornecidas por um mecanismo IFC são descritas em função do controle que esse exerce sobre diferentes *canais de comunicação*, isto é, meios de propagação de informação num sistema. Esses canais são tradicionalmente classificados como *legítimos* (aqueles que tem a transferência de informação como finalidade, como mecanismos de comunicação inter-processo e comandos de movimentação de dados na memória), ou *encobertos* (aqueles que, apesar de não ter a transferência de informação como propósito, podem ser subvertidos para obtê-la, como o estado de recursos compartilhados entre processos (BISHOP, 2003)). O conjunto de canais monitorados por um mecanismo IFC é usualmente determinado considerando-se os desenvolvedores de software como atacantes, podendo-se dividi-los entre desenvolvedores bem-intencionados ou maliciosos. Presume-se que os desenvolvedores bem-intencionados não façam uso de canais encobertos ati-

vamente – ou seja, a presença desse tipo de canal em seus programas se dá de forma acidental, portanto, a monitorização desse tipo de canal não se faz necessária. Porém, considerando-se o caso de desenvolvedores maliciosos, que podem atuar para subverter a atuação do mecanismo IFC, a monitorização de canais encobertos se mostra crucial.

A despeito da potencial melhoria na segurança de dados, o emprego de mecanismos IFC em um sistema computacional pode causar impactos negativos na programabilidade e no desempenho desse, dentre outros prejuízos. Quanto à programabilidade, verifica-se que um mecanismo IFC pode impossibilitar ou dificultar o desenvolvimento de programas que manipulem dados de diferentes categorias e que necessitem, ainda assim, de suas capacidades de comunicação inalteradas. Além disso, a depender da técnica de implementação utilizada, o mecanismo pode restringir o ferramental disponível para desenvolvimento de software (como linguagens, bibliotecas ou frameworks), bem como exigir a reescrita ou recompilação de programas já desenvolvidos. Quanto ao desempenho, verifica-se que os impactos causados pelo uso de um mecanismo IFC estão associados a sobrecustos decorrentes tanto da execução do próprio mecanismo, quanto das restrições que esse impõe ao sistema. Essas restrições não só reduzem o grau de compartilhamento de recursos entre sujeitos, como também podem exigir desses a execução de comandos adicionais para implementar suas funcionalidades.

1.1.2 Limitações encontradas na literatura

Frente às restrições do cenário considerado neste trabalho, as propostas de uso de mecanismos IFC em ambientes móveis encontradas na literatura mostram-se inadequadas. Como razões para tanto, pode-se listar:

- Alterações nos sistemas operacionais ou de *middleware* das plataformas móveis: a maioria das propostas contempla alterações nessas camadas da pilha de software, mas, neste trabalho, assume-se que as empresas não tenham qualquer influência sobre as plataformas móveis utilizadas por seus clientes, não podendo utilizar-se desse tipo de alteração para proteger os dados corporativos.
- Garantias de segurança insuficientes: muitas das propostas assumem que a utilização de canais encobertos é improvável, relegando o rastreamento dos mesmos. Neste trabalho, porém, assume-se que desenvolvedores de aplicativos são potencialmente maliciosos e, portanto, o rastreio de canais encobertos é necessário.
- Ausência do tratamento de aspectos de integridade: enquanto as propostas desconsideram o tratamento de aspectos de integridade, assume-se, no cenário considerado neste trabalho, que as empresas desejem preservar a integridade de dados submetidos a seus sistemas, pelo menos aqueles que forem entrada de um conjunto de operações críticas.
- Dificuldade de replicação em diferentes plataformas: as propostas frequentemente se concentram na utilização de mecanismos IFC em uma plataforma móvel única, e a transposição desses a outras plataformas não se daria de forma trivial.

Dadas as discrepâncias entre os requisitos do cenário considerado e as propriedades das propostas encontradas na literatura, justifica-se a elaboração de uma nova proposta capaz de contemplá-los.

1.2 Objetivos

Este trabalho tem como objetivos a especificação de um mecanismo IFC para dispositivos móveis e a descrição da arquitetura dos aplicativos executados sob as restrições do mesmo. O mecanismo IFC deve não deve impor qualquer alteração aos sistemas operacionais ou de *middleware* das plataformas móveis, bem como deve ser facilmente transposto entre diferentes plataformas. Além disso, o mecanismo deve permitir a utilização de linguagens de programação, bibliotecas e frameworks de uso habitual no desenvolvimento de aplicativos móveis. O mecanismo deve, ainda, possibilitar a implementação de um conjunto de funcionalidades relacionadas à visualização de dados corporativos ou de clientes – preservando a *confidencialidade* de tais dados –, bem como de manipulação de ativos corporativos ou de clientes – devendo preservar a *integridade* dos dados envolvidos nessas operações, em relação ao que os usuários fornecem como entrada das mesmas.

1.3 Contribuições

Este trabalho tem como principal contribuição a utilização, em aplicativos móveis, de conceitos da área de IFC em aplicações Web. Essa ideia se fundamenta na grande adoção de componentes de visualização Web nas plataformas móveis, fato que fornece um *sandbox* universalmente disponível e que permite a implementação de mecanismos IFC em aplicativos tradicionais das plataformas, sem propor alterações nos sistemas operacionais ou de *middleware* das mesmas. Além disso, o ferramental de programação Web é largamente utilizado por desenvolvedores de aplicativos móveis.

Em adição, este trabalho propõe o controle da execução de aplicativos não-confiáveis através de áreas confiáveis da interface gráfica, que permi-

tem uma melhor estruturação dos aplicativos executados sob as restrições do mecanismo, bem como possibilitam a inspeção de dados fornecidos como entrada de operações críticas, que manipulam dados corporativos ou de clientes.

1.4 Organização do Texto

Este texto está organizado da seguinte forma: no capítulo 2 apresenta-se os aspectos teóricos referentes aos mecanismos de controle de fluxo de informação, e no capítulo 3 as características de trabalhos relacionados são detalhadas. No capítulo 4 é apresentado o mecanismo proposto e no capítulo 5 ele é analisado frente aos requisitos apresentados nesta seção. O capítulo 6 apresenta as conclusões finais e as sugestões de trabalhos futuros.

2 ASPECTOS TEÓRICOS: CONTROLE DE FLUXO DE INFORMAÇÃO

Para proteger a confidencialidade de informações, sistemas computacionais tradicionalmente empregam mecanismos de controle de acesso discricionário – como *Access Control Lists*, de Lampson (1974) –, que operam regulando o acesso a objetos detentores de informação exclusivamente em função de regras definidas pelos donos desses objetos. Porém, tais mecanismos são insuficientes para garantir que um sujeito, depois de ter lido determinada informação, divulgue-a a outros sujeitos com quem possa se comunicar, mas que não possuam o mesmo direito de lê-la. O *Problema do Confinamento*, conforme originalmente enunciado por Lampson (1973), consiste em impedir precisamente esse tipo de violação.

Como alternativa de solução ao problema do confinamento, destacam-se os mecanismos de *Controle de Fluxo de Informação*, ou, como doravante designados, mecanismos IFC (da sigla em inglês de *Information Flow Control* (LAMPSON, 2015)). Esses mecanismos, cujas origens relacionam-se ao *controle de acesso mandatário* (BISHOP, 2003), rastreiam a propagação de informação através dos objetos de um sistema, associando-lhes rótulos de segurança que registram as categorias de informações neles contidas. Assim, atuando com base em um conjunto pré-determinado de regras definidas sobre essas categorias, tais mecanismos podem bloquear as comunicações dos sujeitos, caso essas violem alguma regra desse conjunto.

O cenário descrito no problema do confinamento abrange somente a proteção de confidencialidade, mas os mecanismos IFC podem ser igualmente utilizados para proteção da integridade, assegurando que informações consideradas íntegras não sejam comprometidas. Note-se que, nesse caso, o significado de integridade não é tão bem definido quanto o de confidencialidade, e pode variar a depender do contexto de aplicação do mecanismo (BIRGISSON; RUSSO; SABELFELD, 2010). Entretanto, no restante desse capítulo, a apresentação concentra-se nos aspectos de confidencialidade.

As regras de atuação de um mecanismo IFC são prescritas em uma *Política de Fluxo de Informação*, que, segundo Denning (1976), é formada por uma tripla $\langle SC, \sqcup, \sqsubseteq \rangle$, onde:

- SC : é conjunto de classes de segurança, descrevendo as possíveis categorias de informação.
- $\sqcup$: é a *operação de combinação de classes*, uma operação associativa e comutativa definida em SC , especificando a qual classe de segurança pertencem informações obtidas a partir da combinação de outras.
- $\sqsubseteq$: é a relação *pode-fluir-para* em SC , especificando os fluxos de informação legítimos entre classes de segurança – efetivamente, descrevendo os fluxos de informação legítimos entre objetos do sistema, uma vez que esses tenham sido mapeados às classes de SC .

Denning (1976) estabelece ainda um conjunto de condições sob os elementos da tripla $\langle SC, \sqcup, \sqsubseteq \rangle$ formam um reticulado. Alguns exemplos importantes desse tipo de política, que têm dominado a literatura desde então, são apresentados na seção 2.3. Por ora, os exemplos 1 e 2 ilustram duas políticas simples, respectivamente de confidencialidade e integridade, que formam

reticulados. No restante deste capítulo, salvo quando menção em contrário e sem perda de generalidade, assume-se que os mecanismos IFC ilustrados exercitem as políticas destes exemplos.

Exemplo 1: Política Simples de Confidencialidade

$$SC = \{l, h\}$$

$$\sqcup = \{((l, l), l), ((l, h), h), ((h, l), h), ((h, h), h)\}$$

$$\sqsubseteq = \{(l, l), (l, h), (h, h)\}$$

Nesse exemplo, h (de *high*) e l (de *low*) especificam informações de divulgação restrita ao sistema e de livre divulgação, respectivamente. Informações computadas a partir de outras de uma só classe recebem a mesma classificação dessas, e se calculadas em função de informações de classes diferentes, recebem classificação h . Informações de um objeto de rótulo l podem fluir somente para objetos de rótulo h ou l , mas informações de um objeto de rótulo h podem fluir somente para outro rotulado h .

Exemplo 2: Política Simples de Integridade

$$SC = \{r, u\}$$

$$\sqcup = \{((r, r), r), ((r, u), u), ((u, r), u), ((u, u), u)\}$$

$$\sqsubseteq = \{(r, r), (r, u), (u, u)\}$$

Nesse exemplo, r (de *reliable*) e u (de *unreliable*) especificam informações em que se credita alta e baixa integridade, respectivamente. Informações calculadas a partir de outras de uma só classe recebem a mesma classificação dessas, e se calculadas em função de informações de classes diferentes, recebem classificação u . Informações de um objeto de rótulo r podem fluir somente para objetos de rótulo u ou r , mas informações de um objeto de rótulo u podem fluir somente para outros de rótulo u .

Duas propriedades operacionais importantes para categorizar diferentes implementações de mecanismos IFC são a *granularidade de rastreo* e o *tempo de atuação*. A primeira propriedade especifica o conjunto de objetos que o mecanismo considera como detentores e/ou manipuladores de informação, e os meios através dos quais informações são trocadas entre eles – os chamados *canais de comunicação*. A segunda propriedade relaciona-se ao instante de tempo em que fluxos de informação indevidos são detectados e bloqueados: antes da execução dos sujeitos que os perpetram (os mecanismos *estáticos*) ou no momento de sua execução, quando da ocorrência dos fluxos indevidos (os mecanismos *dinâmicos*).

Ambas as propriedades de granularidade de rastreo e tempo de atuação são largamente influenciadas pela camada do sistema computacional em que os mecanismos são implementados, permitindo-se classificá-los em três tipos:

- *OS-IFC* (de *Operating System IFC*): são mecanismos implementados em sistemas operacionais, possuindo granularidade de rastreo limitada a objetos como processos, *threads*, segmentos de memória, arquivos, conexões de rede, dispositivos de I/O, entre outros. Posto que essas entidades agrupam grandes quantidades de dados, diz-se que a granularidade de rastreo desses mecanismos é grosseira, uma vez que a atribuição de um rótulo de segurança a um desses objetos confere o mesmo rótulo de segurança a todos os dados nele encerrados. Usualmente, expressa-se a granularidade de rastreo desses mecanismos como *em nível de processo ou thread*, que são os sujeitos responsáveis pela execução de fluxos de informação. Tais mecanismos são, por natureza, dinâmicos.
- *RS-IFC* (de *Runtime System IFC*): são mecanismos implementados como controles em um sistema de *runtime* de uma linguagem de pro-

gramação ou em um sistema de *middleware*. Possuem granularidade de rastreo potencialmente mais fina do que mecanismos OS-IFC, sendo capazes de rastrear fluxos de informação internos aos processos até o nível de posições individuais de memória (diz-se, nesse caso, granularidade de rastreo “em nível de variável”). Como no caso anterior, são mecanismos de natureza dinâmica.

- *L-IFC* (de *Language IFC*): são mecanismos implementados como etapas de análise estática em compiladores, atuando como verificadores da adequação dos sujeitos às políticas de fluxo de informação, que são fornecidas conjuntamente como entrada ao processo de compilação. Para que tal análise seja possível, usualmente é necessária a especialização da linguagem de entrada. As granularidades de rastreo desses mecanismos, por sua vez, estão relacionadas à natureza da linguagem, compreendendo, por exemplo, variáveis, comandos, rotinas e argumentos (em linguagens imperativas, definindo-se com granularidade de rastreo *em nível de variável*) ou funções e *monads* (em linguagens funcionais). São os mecanismos de natureza estática mais comumente encontrados na literatura.

Além dos mecanismos das categorias relacionadas, existem ainda mecanismos híbridos, que abrangem implementações em diferentes camadas de um sistema computacional (usualmente, envolvendo implementações RS-IFC e L-IFC). A seção 2.1 detalha as maneiras com que os mecanismos das diferentes categorias tratam os canais de comunicação, e a seção 2.2 trata do ferramental teórico tradicionalmente utilizado para formalizar e classificar as garantias de segurança fornecidas pelos mecanismos ao fazê-lo. A seção 2.4, por fim, sumariza os conceitos apresentados nas demais seções do capítulo e discute os compromissos decorrentes do emprego de mecanismos IFC.

2.1 Canais de Comunicação e a Implementação de Mecanismos IFC

Canais de comunicação são os meios através dos quais as informações são intercambiadas entre objetos num sistema computacional. Para definí-los de maneira abrangente, deve-se recorrer à ideia do desconhecimento da informação, de modo similar ao que fazem Clarkson, Myers e Schneider (2009):

Definição 1. *Fluxo de Informação: imagine-se um sistema computacional onde há um objeto o e um sujeito s . O objeto o contém um valor v , que é desconhecido pelo sujeito s . Entretanto, s possui uma crença acerca de v , representada como uma distribuição de probabilidade $P(V)$ (onde V é o conjunto de valores possíveis de v). Se, após uma sequência de computações c , o sujeito s for capaz de alterar sua crença acerca de v , produzindo uma distribuição de probabilidades $P(V|c)$ diferente de $P(V)$, diz-se que c causa um fluxo de informação de o para s .*

Segundo essa definição, um fluxo de informação ocorre não só quando existe o intercâmbio explícito dos dados que a representam entre duas entidades, mas quando qualquer dado que puder ser trocado entre elas permitir alterar o desconhecimento acerca da informação. Por esse motivo, canais de comunicação podem se manifestar por meios imprevistos, fato que possibilita classificá-los como *canais legítimos* ou *canais encobertos*.

2.1.1 Canais Legítimos

Canais de comunicação legítimos são os meios explicitamente projetados para fornecer comunicação entre sujeitos (ou entre sujeitos e objetos passivos) de um sistema computacional. Canais desse tipo são os de mais fácil monitoramento, já que seu uso se dá através de interfaces bem definidas, em

que se pode anexar as verificações do mecanismo diretamente aos controles de utilização dos mesmos. Por esse motivo, espera-se que qualquer implementação de mecanismo IFC seja capaz de controlar todos os canais desse tipo aos quais os sujeitos do sistema tenham acesso.

Para controlar o uso de canais legítimos, um mecanismo IFC segue usualmente um princípio básico de atuação que é independente da natureza de implementação do mesmo. Como já mencionado, essa atuação fundamenta-se na rotulagem dos objetos do sistema com as classes de segurança de SC , podendo haver objetos que possuem rótulos mutáveis ou imutáveis. Aqueles que possuem rótulos imutáveis usualmente demarcam os limites de atuação do mecanismo – isto é, identificam os objetos através dos quais a informação pode adentrar a “região” de influência do mecanismo ou dela sair – e possuem, por esse motivo, rótulo imutável. Os que possuem rótulo mutável, por sua vez, restringem-se à propagação de informação internamente à região de influência do mecanismo, e geralmente recebem um rótulo de segurança adicional – dessa vez, de forma imutável –, que denota a máxima classe de segurança que pode ser atribuída ao objeto. Esses rótulos de segurança, juntamente com a relação $\sqsubseteq$ e a operação $\sqcup$, são utilizados no controle de utilização de canais legítimos como no exemplo 3, que referencia a Figura 1.

Exemplo 3: Suponha um objeto o_1 , rotulado com classe de segurança $c_{o_1} = h$. Um sujeito s_1 , de rótulo mutável inicialmente dado por $c_{s_1} = l$, possui rótulo máximo $c_{s_1}^{max} = h$. No instante de tempo t_0 , o sujeito s_1 tenta então ler do objeto o_1 , ao que o mecanismo IFC reage, verificando se $c_{o_1} \sqsubseteq c_{s_1}^{max}$ é satisfeita; nesse caso, como a relação é satisfeita, a leitura é autorizada e o rótulo c_{s_1} é atualizado com o valor da operação $c_{s_1} \sqcup c_{o_1} = h$, no instante de tempo t_1 . No instante de tempo t_2 , s_1 tenta escrever em um objeto o_2 de rótulo fixo $c_{o_2} = l$, ao que mecanismo IFC reage, verificando se $c_{s_1} \sqsubseteq c_{o_2}$. Nesse caso, como a

relação não é satisfeita, a comunicação é bloqueada.

Figura 1: Exemplo de operação de um mecanismo IFC genérico

Esse princípio básico de funcionamento é, evidentemente, aplicado de maneiras diferentes em função da natureza de implementação dos mecanismos. Em especial, a propriedade de granularidade de rastreo do mecanismo determina o que são sujeitos e quais canais de comunicação legítimos estão disponíveis aos mesmos. No caso de mecanismos OS-IFC com granularidade de rastreo em nível de processo, os canais disponíveis se restringem àqueles de comunicação inter-processos (isto é, arquivos, sinais, semáforos, memória compartilhada, entre outros) e de entrada/saída para comunicação externa (como interfaces de rede e de homem-computador). Nos mecanismos OS-IFC com granularidade de rastreo em nível de *thread*, aos canais do caso anterior adicionam-se ainda canais de comunicação intra-processo, como, por exemplo, através do uso de segmentos de memória (ou outros recursos) compartilhados. Os mecanismos RS-IFC, por sua vez, usualmente realizam o rastreo de maneira intra-processo, com granularidade em nível de *threads* (caso em que o rastreo é implementado de forma similar à dos mecanismos OS-IFC de mesma granularidade) ou em nível de variável. Nesse último caso, o mecanismo IFC atua rotulando individualmente as variáveis do

processo, e fazendo o rastreio das classes de segurança das informações manipuladas pelo mesmo, desde o momento de sua entrada até sua saída. Assim, os canais de comunicação disponíveis nesse nível de granularidade perfazem a movimentação e transformação de dados (através de comandos aritméticos, de atribuição, chamadas de procedimentos, indexação, entre outros) e de entrada/saída. O trecho de código 1 exemplifica a operação de um mecanismo IFC dessa natureza.

```

1. file fh;           //Arquivo contendo dados de classe h
2. file fl;           //Arquivo contendo dados de classe l
3. var v1, v2;
4.
5. v1 := 0;           //v1 assume rótulo l
6. v2 := fread(fh); //v2 assume rótulo h
7. v1 := v2 + v1;    //v1 assume rótulo h
8. fwrite(fl, v1);  //operação ilegal

```

Trecho de código 1: Exemplo de operação de um mecanismo RS-IFC com granularidade em nível de variável

No exemplo, o arquivo **fh** armazena dados de classe de segurança *h*, ao passo que o arquivo **fl** armazena dados de classe *l*. Na linha 4, associa-se à variável **v1** rótulo de segurança *l*, uma vez que o valor a ela atribuído é de uma constante. Na linha 5, a variável **v2** recebe os dados lidos do arquivo **fh** e, por isso, associa-se a ela rótulo de segurança *h*. Na linha 6, atribui-se a **v1** um valor calculado em função do valor de **v1** e **v2**; assim, associa-se a **v2** um rótulo de segurança dado pelo operador $\sqcup$ sobre os rótulos de segurança de **v1** e **v2**, isto é, $l \sqcup h = h$. A linha 7 representa uma violação de segurança, uma vez que se tenta escrever o conteúdo de **v1** no arquivo **fl**, e a relação $\sqsubseteq$ não é satisfeita entre as classes de segurança de **v1** (*h*) e **fl** (*l*).

O funcionamento de mecanismos L-IFC assemelha-se ao de mecanismos RS-IFC com granularidade em nível de variável (evidentemente, para o caso

de programação imperativa, em que há variáveis; a implementação de IFC em demais paradigmas não é discutida neste texto). Assim, as variáveis de um programa recebem rótulos de segurança individualmente, e a informação flui entre as mesmas (ou para dentro/fora do programa) através dos mesmos meios que no caso anterior. Entretanto, no caso de mecanismos L-IFC, os fluxos de informação devem ser todos previstos em tempo de compilação. Tradicionalmente, tal verificação é executada na etapa de *type-checking* do compilador, o que usualmente requer a modificação do sistema de tipos da linguagem e torna necessário anotar as variáveis, rotinas e classes de um programa com tipos de segurança. Em Sabelfeld e Myers (2006) encontra-se um sumário das pesquisas realizadas acerca da implementação de mecanismos IFC dessa natureza.

2.1.2 Canais Encobertos

Canais encobertos são aqueles que, apesar de não terem sido projetados com a finalidade de estabelecer comunicação, podem ter seu uso subvertido com o intuito de obtê-la. Desse modo, a utilização de um canal de comunicação desse tipo requer a atuação maliciosa dos sujeitos envolvidos (ou do sujeito envolvido) na comunicação. Assim, justamente por sua natureza de utilização extra-protocolar, o controle de canais encobertos apresenta um grau de dificuldade maior do que o controle de canais legítimos. Adiante, algumas categorias de canais encobertos são enumeradas, detalhando-se como os mesmos se manifestam para diferentes granularidades de rastreo, e quais são as alternativas de controle usualmente encontradas na literatura relacionada ao projeto e implementação de mecanismos IFC.

2.1.2.1 Fluxos Implícitos

Este tipo canal encoberto manifesta-se quando o valor de determinada informação influencia implicitamente a ocorrência (ou não-ocorrência) de um evento, revelando-o parcial ou totalmente. A depender da granularidade de rastreio considerada, fluxos implícitos operam através de diferentes meios, podendo basear-se, inclusive, na própria operação regulatória do mecanismo IFC.

No caso de mecanismos de granularidade em nível de variável, fluxos implícitos são ocasionados pela dependência entre os comandos executados por um programa e o estado em que esse se encontra quando dessa execução. Uma vez que esse estado de execução depende dos valores testados pelo programa em comandos de desvio condicional, o conteúdo das variáveis testadas influencia implicitamente os valores assumidos pelas variáveis manipuladas nos comandos executados no estado. Como exemplo, considere-se o trecho de código 2, uma variação do trecho de código 1 da seção 2.1.1. Nesse exemplo, o arquivo **fh** é rotulado com classe *h*, e o arquivo **fl**, com classe *l*. Novamente, estão presentes as variáveis **v1** e **v2**. Note-se que na linha 7 do trecho ocorre a atribuição de um valor constante a **v1**, operação que mantém o rótulo *l* atribuído a **v1** na linha 4. Entretanto, essa atribuição ocorre em função do valor de **v2**, testado na linha 6. Além disso, **v2** possui, no momento do teste, rótulo *h*, adquirido na leitura de arquivo na linha 5. Assim, quando ocorre a atribuição da linha 7, ocorre o vazamento do valor de **v2** ao arquivo **fl** na linha 9.

Para se realizar o rastreio de fluxos implícitos como o apresentado no trecho de código 2, pode-se manter o registro das variáveis testadas em comandos de desvio condicional, bem como de suas respectivas classes de

```

1. file fh, fl;
2. var v1, v2;
3.
4. v1 := 0;
5. v2 := fread(fh);
6. if (v2 = 1) then
7.   v1 := 1;
8. endif
9. fwrite(fl, v1);

```

Trecho de código 2: Exemplo de atuação de um fluxo implícito entre variáveis

segurança, e contabilizá-las como influenciadoras dos comandos subsequentes. Para tanto, geralmente associa-se também ao *program counter* um rótulo de segurança c_{pc} , que é atualizado nos comandos de desvio condicional e utilizado para determinar a classe de segurança de valores armazenados nas variáveis do programa. Como exemplo, referenciando-se novamente o trecho de código 2, tomando-se o valor inicial de c_{pc} na linha 4 como l , para se determinar a classe de segurança do valor a ser armazenado em **v1**, utiliza-se da classe de segurança original do valor (que é l , dado esse é uma constante) combinando-a com o de c_{pc} através da operação $\sqcup$, isto é, $l \sqcup c_{pc} = l$. Analogamente, na linha 5, o valor armazenado em **v2** tem rótulo h , pois é dado pela classe de segurança original do valor (que é h , pois o valor é resultante da leitura de **fh**) combinado com c_{pc} , isto é, $h \sqcup c_{pc} = h$. A partir do escopo do bloco declarado na linha 6, c_{pc} assume valor h , pois a variável **v2** (que tem classe h) é testada para se adentrar o escopo; assim, o novo valor de c_{pc} é calculado em função do valor antigo (que era l) e do rótulo de **v2**, isto é, $l \sqcup h = h$. A partir da linha 8 (onde termina o escopo) o valor de c_{pc} pode ser retrocedido ao anterior, l . Observe-se que na linha 7 determina-se que o rótulo de segurança a ser armazenado em **v1** tem valor h , pois é dado por $l \sqcup c_{pc}$, onde c é a classe de segurança do valor (uma constante) e $c_{pc} = h$. Desse modo, na linha 9, o fluxo ilegal de informação para o arquivo **fl** pode ser detectado.

Uma limitação do método descrito em mecanismos RS-IFC é a de que o fluxo ilegal da linha 9 só é detectado se a atribuição da linha 7 for de fato executada, isto é, se o valor armazenado em **v2** for 1. Entretanto, se a atribuição não é executada, ainda se pode deduzir posteriormente, ao se ler o conteúdo de **fl**, que o valor armazenado em **v2** não era igual a 1. Segundo a definição 1, porém, essa situação ainda constitui fluxo de informação. Para evitá-la, poder-se-ia manter o valor de c_{pc} em h , mesmo que o programa não entre no escopo da linha 7. Entretanto, essa prática leva ao valor de c_{pc} assumir valores monotonicamente mais restritivos, um fenômeno conhecido como *label creep*. Essa limitação, como constatado inicialmente por Denning (1976), não é encontrada em mecanismos L-IFC, já que esses são capazes de prever os fluxos implícitos independentemente de sua ocorrência ou não, pois operam em tempo de compilação. Note-se porém que, mesmo não sendo eficazes no tratamento de fluxos implícitos, os mecanismos RS-IFC de granularidade em nível de variável não são menos seguros do que mecanismos L-IFC, quando o canal de terminação (apresentado na seção 2.1.2.4) é considerado (SABELFELD; RUSSO, 2010).

No caso de mecanismos IFC com granularidades de rastreo mais grosseiras (como em nível de thread ou processo), fluxos implícitos podem se manifestar quando os rótulos de segurança dos sujeitos se alterarem de modo implícito, sem a intervenção do próprio sujeito cujo rótulo é alterado. As figuras 2, 3, e 4, adaptadas de Krohn e Tromer (2009), ilustram a transmissão de um bit entre dois processos p e q , através de processos intermediários q_0 e q_1 . No exemplo, os processos q , q_1 e q_2 têm rótulos de segurança l , enquanto p tem rótulo de segurança h . O processo p , a depender do valor do bit a ser transmitido, envia uma mensagem a q_0 ou q_1 , alterando o rótulo de segurança do processo receptor para h . Na Figura 3, p deseja transmitir o valor 1 a q ,

enviando uma mensagem a q_0 . O processo que não recebeu a mensagem (q_1 , no caso), então, passado um certo intervalo de tempo, envia o valor 1 a q , enquanto o processo anterior nada faz (Figura 4). O processo q , assim, saberá o valor do bit transmitido, sem que tenha o seu rótulo alterado para h . Para transmitir mais bits, q cria novos processos q_0 e q_1 , um para cada bit, e repete o procedimento.

Figura 2: Fluxo implícito entre processos (configuração inicial)

Figura 3: Fluxo implícito entre processos (configuração intermediária)

Figura 4: Fluxo implícito entre processos (configuração final)

Como observação final acerca de fluxos implícitos, note-se que o último exemplo evidencia a própria atuação do mecanismo IFC como meio de manifestação desse tipo de canal encoberto. Assim, a possibilidade de ocorrência de situação semelhante em demais implementações de mecanismos IFC – independentemente de sua natureza – deve ser uma preocupação de seus projetistas.

2.1.2.2 Canais de Recurso Compartilhado

Canais de recurso compartilhado originam-se do isolamento imperfeito entre sujeitos compartilhando recursos de hardware ou software de um sistema computacional. Tais imperfeições relacionam-se muitas vezes às abstrações inerentes às interfaces de acesso a esses recursos, projetadas sem o cuidado de eliminar estados globais visíveis aos sujeitos envolvidos no seu compartilhamento. No limite, o próprio esgotamento da capacidade limitada de um desses recursos pode levar à constituição de um estado global, motivo pelo qual os canais de recurso compartilhado são denominados *canais de exaustão de recursos* por alguns autores.

Como exemplo de manifestação desse tipo de canal, observe-se a Figura 5, onde os sujeitos s_0 e s_1 , com rótulos $c_{s_0} = h$ e $c_{s_1} = l$ respectivamente, compartilham um recurso de memória m , de capacidade limitada 1 MB. Apesar de s_0 e s_1 não poderem se comunicar bidirecionalmente por qualquer canal legítimo (tendo em vista que a relação $c_{s_0} \sqsubseteq c_{s_1}^{max}$ não é satisfeita), ambos podem acordar um protocolo para transmitir uma sequência de bits através de m . Para tanto, os sujeitos valem-se do fato da capacidade de m ser alocada de modo “first-come-first-serve” a qualquer um deles; e do fato de que um sinal de erro é retornado quando se solicita uma quantidade de memória além da disponível. Assim, um protocolo para a transmissão de n bits de s_0 a s_1 pode funcionar da seguinte maneira: s_1 cria o arquivo f , escreve nele uma posição idx do bit a ser transmitido, e põe-se a esperar por uma quantidade de tempo T ; o sujeito s_0 , então, lê f , tomando o conhecimento de idx e, em função do valor do bit nessa posição, aloca/retém ou não aloca/desaloca 1 MB de m . Posteriormente, transcorrido o tempo T , s_1 tenta alocar uma quantidade arbitrária de memória, menor que 1 MB, de m e, a depender do sucesso dessa operação, consegue inferir o valor do bit sendo transmitido. Esse procedimento então é

repetido para um novo índice idx , até que sejam transmitidos todos os n bits.

Figura 5: Exemplo de uso de um canal de recurso compartilhado

Um procedimento análogo pode ser utilizado para transmitir informações internamente em um processo, constituindo uma dificuldade adicional aos projetistas de mecanismos IFC com granularidade de rastreo em nível de variável. Como exemplo, observe-se o trecho de código 3, adaptado do trecho de código 2. A função **malloc** tenta alocar a quantidade de memória fornecida como argumento, retornando **true** caso consiga, e **false** no caso contrário. Assim, o sucesso da alocação realizada na linha 9 depende da ocorrência da alocação na linha 6, que, por sua vez, depende do valor de **v2**. Assim, o programa consegue enviar um valor de **fh** à **fl**, transgredindo as restrições do mecanismo IFC.

Devido à possibilidade da existência desse tipo de canal, implementações de mecanismos IFC devem assumir uma postura conservadora, reduzindo o número de recursos compartilhados entre sujeitos rotulados com classes de segurança incompatível. Quanto aos recursos de compartilhamento inevitável (como a CPU, por exemplo), segundo Bishop (2003), os mecanismos têm como alternativa: exigir dos sujeitos, antes de sua execução, a declaração dos recursos a serem usados, bem como a capacidade necessária de cada um, e mantê-los alocados estritamente segundo essa declaração (o que elimina a


```
1. file fh, fl;  
2. var v1, v2;  
3.  
4. v1 := 0;  
5. v2 := fread(fh);  
6. if (v2 = 1) then  
7.   malloc(1MB);  
8. endif  
9. if (malloc(1KB) = false) then  
10.  v1 := 1;  
11. endif  
12. fwrite(fl, v1);
```

Trecho de código 3: Exemplo de atuação de canal de recurso compartilhado em um processo

dependência da mesma sobre dados sensíveis); ou ofuscar a quantidade de recursos alocados a um sujeito, através da alocação estritamente uniforme de recursos aos sujeitos do sistema, ou através da inserção de aleatoriedade no processo de alocação, tornando o canal ruidoso.

2.1.2.3 Canais de Temporização

Canais de temporização surgem quando a informação é capaz de influenciar o tempo envolvido na realização de uma operação, podendo inclusive alterar o seu resultado. Nos casos em que há essa alteração, o conhecimento da relação entre o tempo transcorrido para realizar a operação e o seu resultado basta para recuperar a informação, enquanto que, se a alteração não ocorrer, necessita-se de um relógio com o qual se possa medir o tempo transcorrido entre o início e fim da operação. Alguns exemplos da utilização bem-sucedida deste tipo de canal para a extração de informações sensíveis são os de Brumley e Boneh (2005), no qual se expõem chaves de algoritmos criptográficos, e Bortz e Boneh (2007), onde se capturam dados privados de usuários de um sistema Web.

Caso ocorra sob as restrições de mecanismos IFC com granularidades de rastreo em nível de thread ou processo, a utilização de canais de temporização está atrelada ao uso de recursos compartilhados, uma vez que os sujeitos não podem diretamente ler a saída uns dos outros – ou o tempo que levam para produzi-las –, se ao fazê-lo, desrespeitarem a política de fluxo de informação. Um exemplo da utilização de canais de temporização nesse contexto pode ser encontrado em Schaefer (1977), onde se descreve um método para a transmissão de dados entre dois processos através da modulação de uso do *time-slot* atribuído ao processo transmissor. As alternativas de inibição ou mitigação de canais de temporização nesse nível de granularidade de rastreo são as mesmas apresentadas na seção 2.1.2.2.

Para a utilização de canais de temporização sob as restrições de mecanismos IFC com granularidade de rastreo em nível de variável, somam-se ao caso anterior ainda duas possibilidades: os canais de temporização externa e interna. No primeiro caso, a informação transmitida pode causar diferenças no tempo necessário para que o processo produza uma saída, permitindo-se medir externamente a influência da mesma sobre as operações executadas (apesar da denominação *externa*, não há restrições quanto à sua utilização para transmissão interna ao processo). Como exemplo, observe-se o trecho de código 4: o programa lê o conteúdo de **fh**, um arquivo com classificação *h*, e escreve em **fl**, um arquivo com classificação *l*, as strings “start” e “end”. Apesar do conteúdo escrito em **fl** ser independente do conteúdo lido de **fh**, a diferença de tempo entre a escrita das duas saídas depende, sim, desse valor, e um sujeito capaz de medi-la – e, evidentemente, ciente dessa dependência – pode inferir o valor.

No exemplo, a diferença de tempo entre as escritas no arquivo **fl** é devida unicamente à diferença no número de instruções executadas em casos

```

1. file fh, fl;
2. var v1;
3.
4. v1 := fread(fh);
5. fwrite(fl, "start");
6. if (v1 = 1) then
7.   for (i := 0; i < 10000; i++) loop
8.     nop;
9.   endloop
10. endif
11. fwrite(fl, "end");

```

Trecho de código 4: Exemplo de atuação de canal de temporização externa

diferentes. Entretanto, essa diferença de tempo pode se dar em função de razões mais sutis, como diferenças no desempenho do sistema de cache ou de predição de desvio, em função da informação transmitida.

Segundo Kashyap, Wiedermann e Hardekopf (2011), mecanismos IFC dispõem de três tipos de técnicas para o controle de canais de temporização externa: mitigação, segundo a qual simplesmente se reduz a taxa de transmissão do canal pela introdução de aleatoriedade no intervalo de tempo entre operações de I/O (no trecho de código 5, por exemplo, poder-se-ia adicionar um tempo aleatório entre as duas medidas escritas em **fl**); restrição de computação (por exemplo, impedindo-se comandos de iteração dependentes de dados sensíveis); e *scheduling*, que envolve a execução simultânea de diferentes versões do programa, cada uma rotulada de forma imutável com classes de segurança distintas, e sincronizadas à medida que realizam operações de I/O (um exemplo desse tipo de técnica pode ser encontrado em Devriese e Piessens (2010)).

Canais de temporização interna, por sua vez, manifestam-se em programas multi-threaded, consistindo uma ameaça à atuação de mecanismos IFC com granularidade de rastreamento em nível de variável. Essa variante de canal de

temporização, ao contrário das anteriormente apresentadas, dispensa a utilização de um relógio medindo o tempo transcorrido na execução de comandos, fundamentando-se na diferença de resultados provocados pela execução concorrente de diferentes threads manipulando variáveis globais. Um exemplo adaptado de Russo (2007), e reproduzido no trecho de código 5, demonstra que o valor da variável **v1** pode ser copiado para a variável **v2**, através da diferença do número de instruções executados pelas threads 1 e 2, ambas manipulando **v2** (o exemplo supõe que o programa é executado sob um escalonamento do tipo *round-robin*): se **v1** tem valor 1, a atribuição da linha 7 da thread 1 ocorre após a atribuição da linha 3 da thread 2. Note-se que essa cópia ocorre sem que o rótulo de segurança de **v1** influencie o de **v2**.

```
Thread 1
1. if (v1 = 1) then
2.   nop;
3.   nop;
4. else
5.   nop;
6. endif
7. v2 := 1;

Thread 2
1. nop;
2. nop;
3. v2 := 0;
```

Trecho de código 5: Exemplo de atuação de canal de temporização interna

Para atuar contra canais de temporização interna, mecanismos IFC dispõem de diferentes alternativas propostas na literatura, variando desde a proibição de modificação de variáveis globais de classe *l* após comandos de desvio condicional envolvendo variáveis rotuladas com classe *h* (BOUDOL; CASTELLANI, 2002), ou de condições de corrida envolvendo variáveis de classe *l* (ZDANCEWIC; MYERS, 2003), até propostas mais sofisticadas, envolvendo téc-

nicas de particionamento e sincronização (RUSSO, 2007; STEFAN, 2012a).

2.1.2.4 Canais de Terminação

Canais de terminação manifestam-se quando a terminação normal de um processo ou thread é determinada pela informação a ser transmitida, e a ocorrência (ou não) de terminação pode ser observada por um sujeito que conhece a relação pela qual essa determinação se dá. Como exemplo, observe-se o trecho de código 6, adaptado de Askarov (2008). A variável **v1** armazena o valor lido do arquivo **fh**, rotulado com classe *h* e, portanto, seu valor não poderia ser escrito no arquivo **fl**, que é associado à classe *fl*. No entanto, assumindo que **v1** é um inteiro sem sinal de 32 bits, a linha 5 itera sobre todos os valores que **v1** pode assumir, escrevendo no arquivo **fl** o valor da iteração corrente. No momento em que o valor da iteração coincidir com o valor de **v1**, as linhas 7-10 evidenciam que o programa divergirá. Entretanto, o conteúdo de **v1** estará ao final de **fl**.

```
1. file fh, fl;  
2. var v1;  
3.  
4. v1 := fread(fh);  
5. for (i := 0; i < 2^32 - 1; i++) loop  
6.   fwrite(fl, i);  
7.   if (v1 = i) then  
8.     while (true) loop  
9.       nop;  
10.    endloop  
10.  endif  
11. endloop
```

Trecho de código 6: Exemplo de atuação de canal de terminação

O canal de terminação constitui ameaça mais proeminente ao controle de mecanismos IFC com granularidade em nível de variável, muito embora seja possível utilizá-la para contornar o controle de mecanismos de granularidade

mais grosseira (caso em que seria necessário trocar **fl** do exemplo pelo estado de algum recurso compartilhado).

Observe-se que, no trecho de código 6, a não-terminação se dá por divergência. Entretanto, ela poderia se dar por motivos menos evidentes, como, por exemplo, a ocorrência de exceções não tratadas. Para controlar a atuação de canais de terminação, os mecanismos IFC dispõem das mesmas técnicas listadas na seção 2.1.2.3 para a proteção de canais de temporização interna: mitigação, restrição de computação e *scheduling*.

2.1.2.5 Canais Físicos

Este tipo de canal surge quando é possível, através da medição de alguma grandeza física do computador executando um programa, inferir qual sequência de estados pela qual este último passa e, em consequência, o valor de alguma informação que o levou a tal sequência. Alguns exemplos de trabalhos utilizando canais desse tipo são os de Carlier (2004) (emanação eletromagnética), Genkin, Shamir e Tromer (2014) (emanação sonora) e Kocher, Jaffe e Jun (1999) (consumo de potência). Muito embora o uso de canais desse tipo pressuponha um modelo de atacante inverossímil, tais trabalhos ilustram que a utilização desse tipo de canal pode ser factível em determinados cenários.

2.2 Não-Interferência e Desclassificação

Não-interferência é uma propriedade tradicionalmente definida para formalizar as garantias de segurança providas por mecanismos IFC. Enunciada com esse fim pela primeira vez no trabalho de Volpano, Irvine e Smith (1996) a partir da enunciação mais genérica de Goguen e Meseguer (1982), a propriedade de não-interferência encerra uma semântica elementar: a de que os dados rotulados com classe de segurança h fornecidos como entrada de um programa ou sistema não devem influenciar os valores rotulados com classe de segurança l produzidos como saída do mesmo. Assim, se um programa ou sistema satisfaz não-interferência, o conhecimento dos seus valores públicos de saída não reduz o desconhecimento acerca dos valores privados fornecidos como entrada ao mesmo – fato que, segundo a definição 1, implica que o programa não efetiva fluxos de informação indevidos.

Em função da necessidade de representar os diferentes modelos de execução considerados nas implementações de mecanismos IFC – e, em consequência, contabilizar os canais de comunicação presentes nesse modelos –, definições diversas de não-interferência surgiram na literatura. Na enunciação mais simples da propriedade, considera-se um modelo de execução em que programas de thread única realizam uma única operação de entrada, computam sobre esses valores, executam uma única operação de saída e terminam. Utilizando de canais legítimos, um possível atacante pode somente manipular os valores de rótulo l fornecidos como entrada ao programa, e observar os valores l produzidos como saída pelo mesmo.

Essa definição mais simples de não-interferência pode ser formalizada – considerando-se uma granularidade de rastreo em nível de variável, como tradicionalmente – com base nos seguintes elementos:

VN , conjunto de nomes de variáveis

$SC = \{h, l\}$, conjunto de níveis de segurança da política 1

M , o conjunto de configurações de memória, formado por triplas do tipo (V, α, β) , em que $V \subset VN$ representa as variáveis nela presentes, $\alpha : V \rightarrow SL$ representa o nível de segurança atribuído a cada uma delas e $\beta : V \rightarrow Z$, o valor atribuído a elas

$LM : M \subseteq 2^{VN}$, função que indica o conjunto de variáveis da configuração de memória que são públicas, tal que, para $m \in M$, $LM(m) = LM((V, \alpha, \beta)) = \{v \in V | \alpha(v) = l\}$

$LER : M \subseteq M$, relação de baixa-equivalência entre memórias, definida de maneira tal que $\forall m_1, m_2 \in M, m_1 = (V_1, \alpha_1, \beta_1), m_2 = (V_2, \alpha_2, \beta_2), m_1 LER m_2 \iff LM(m_1) = LM(m_2)$ e $\forall v \in V_1, \alpha_1(v) = \alpha_2(v)$; diz-se que m_1 e m_2 são baixo-equivalentes, possuindo o mesmo conjunto de variáveis públicas, com os mesmos valores

Utilizando-se desses elementos, pode-se modelar um programa p (que sempre termina) como uma função $p : M \rightarrow M$. Então, pode-se definir não-interferência da seguinte maneira:

Definição 2. *Não-interferência: o programa p satisfaz a propriedade de não-interferência se para qualquer par $m_1, m_2 \in M$ fornecido como entrada de p , tal que $m_1 LER m_2$, então $p(m_1) LER p(m_2)$*

Em termos mais simples, a definição 2 declara que, para qualquer par de conjunto de entradas que concordem nos valores de suas variáveis públicas, os resultados da passagem do par para p também concordam nos valores de suas variáveis públicas – o que encerra a semântica essencial supracitada, de que a computação de valores públicos independe de valores privados.

A definição 2 é equivalente à que Hedin e Sabelfeld (2012) definem como *Não-Interferência Insensível à Terminação*, pois considera que fluxos de informação através do canal de terminação não são visíveis ao atacante. Frente a essa limitação, esses autores definem a *Não-Interferência Sensível à Terminação*, segundo a qual o atacante tem acesso a um *efeito observável*, que pode ser tanto o conjunto de valores de classificação pública saída do programa, quando ele de fato termina, quanto a ciência da não-terminação do mesmo, quando ela ocorrer. Assim, tem-se a definição:

Definição 3. *Não-interferência Sensível à Terminação: um programa p satisfaz a propriedade de não-interferência sensível à terminação se, para qualquer par $m_1, m_2 \in M$ fornecido como entrada de p , tal que $m_1 \text{ LER } m_2$, então o programa não termina com qualquer das duas entradas, ou termina em ambas e $p(m_1) \text{ LER } p(m_2)$.*

Desse modo, um atacante não tem meios de diferenciar os valores privados de m_1 de m_2 somente ao inspecionar os valores públicos de $p(m_1)$ ou $p(m_2)$, ou ao observar a ocorrência de terminação ou não de p com qualquer das entradas.

Entretanto, essa definição não abrange programas interativos, isto é, que executam comandos de I/O intermediários, habilitando um atacante a lhes fornecer entradas ou observar seus resultados, antes do término dos mesmos. Para contabilizar tal cenário, Hedin e Sabelfeld (2012) definem a *Não-Interferência Sensível a Progresso*, que é satisfeita por um programa quando esse, alimentado com memórias m_1, m_2 baixo-equivalentes – e com entradas intermediárias também todas baixo-equivalentes –, fornece resultados intermediários todos também baixo-equivalentes, ou não termina depois de um mesmo resultado ou entrada intermediários em ambas as execuções.

Kashyap, Wiedermann e Hardekopf (2011) fornecem outras definições de

modelos de não-interferência, baseadas na observância dos mesmos aos canais de terminação e temporização. Assim, em relação ao canal de terminação, um modelo de não-interferência pode ser:

1. Insensível à Terminação: quando desconsidera tanto a possibilidade de um programa não terminar, quanto de terminar de modo anormal (quando da ocorrência de, por exemplo, exceções não tratadas), ou a possibilidade do atacante observar a terminação do programa por qualquer um desses motivos. A definição 2 recai nesse caso;
2. Fracamente Sensível à Terminação: quando o modelo considera que o programa pode divergir, sem que execute qualquer operação indicativa da ocorrência de erro. Segundo os autores, todos os modelos que consideram o canal de terminação são desse tipo, e é nesta categoria que recai a definição 3;
3. Fortemente Sensível à Terminação: nesse tipo de modelo, observa-se a possibilidade de não-terminação não somente pelo mesmo motivo da categoria anterior, como também pela ocorrência de erros de execução ou exaustão de recursos. Segundo os autores, esse é o tipo de modelo de implementação mais difícil, e segundo o que observam, ainda não há exemplos na literatura classificáveis nessa categoria.

De maneira similar, os modelos de não interferência podem ser classificados em função da sua observância à ocorrência de fluxo através do canal de temporização da seguinte maneira:

1. Insensível à Temporização: quando considera que o atacante não tem meios de monitorar o tempo de execução de um programa, e, portanto, não é capaz de utilizar esse canal. A definição 2 recai nesse caso;

2. Fracamente Sensível à Temporização: quando o modelo considera que o um atacante pode observar o tempo de execução de um programa, e esse é função somente do número de comandos executados, que tem o tempo de execução sempre igual. Segundo os autores, a maioria dos modelos sensíveis à temporização recai neste caso;
3. Fortemente Sensível à Temporização: nesse tipo de modelo, como no anterior, o atacante é capaz de monitorar o tempo de execução de um programa, que é função agora não somente do número de comandos executado pelo mesmo, mas também da diferença de tempo entre cada um deles, que, por sua vez, é função de características operacionais, como o desempenho de caches, pipelines e preditores de desvio. Analogamente à definição de não-interferência fortemente sensível à terminação, esse é o tipo de modelo de implementação mais difícil, também não havendo exemplos na literatura classificáveis nessa categoria.

Segundo ainda o que argumentam Smith e Volpano (1998), a presença do não-determinismo em programas *multithread* torna inadequadas definições como a 2 ou 3, pois um programa desse tipo, executado duas vezes com as mesmas entradas, pode fornecer como saída resultados diferentes. Por isso, os autores propõe uma definição de *Não-Interferência Possibilística*, segundo a qual um programa é não-interferente se os valores das variáveis privadas fornecidas a ele como entrada não alteram o conjunto de valores que podem ser atribuídos às variáveis públicas de saída. Observando que a definição de não-interferência possibilística é dependente da presença de um *scheduler* de threads aleatório, Sabelfeld e Sands (2000), por sua vez, propõem a definição de *Não-Interferência Probabilística*, segundo a qual, em um programa não-interferente, a distribuição de probabilidades dos valores das variáveis públicas de saída não é alterada pelos valores das variáveis públicas de entrada. Hedin

e Sabelfeld (2012) fornecem uma síntese de trabalhos recentes que versam acerca de não-interferência na presença de computação não-determinística. Por fim, a definição de *fraco sigilo* (ou *weak-secrecy*) de Volpano (1999) relaciona-se à definição de não-interferência, caracterizando a segurança de mecanismos IFC que são vulneráveis a fluxos implícitos (e que, por isso, não satisfazem não-interferência).

Todas as definições de não-interferência apresentadas compartilham a deficiência de serem muito restritivas, tendo-se em consideração a definição 1, de fluxo de informação. Assim, conclui-se que modelos de não-interferência são incapazes de modelar os requisitos de segurança de operações que necessitam *desclassificar* informação (isto é, divulgar informação privada; tratando-se de classes de integridade, a operação análoga é denominada endosso), como rotinas de verificação de senha (em que o resultado da verificação da senha revela informações sobre o valor da senha, que é, obviamente, privado). Mesmo quando esse tipo de operação é executado de uma maneira controlada e relativamente segura, a desclassificação é sempre considerada uma operação insegura sob um ponto de vista de não-interferência. Assim, alguns trabalhos surgiram na literatura propondo a definição de parâmetros de segurança para a operação de desclassificação, como discutido por Hedin e Sabelfeld (2012), Askarov e Sabelfeld (2007) e Askarov e Chong (2012).

2.3 Modelos de Políticas de Fluxo de Informação

Modelos de Políticas de Fluxo de Informação auxiliam a especificação e análise de *Políticas de Fluxo de Informação*. Os modelos apresentados nesta seção são do tipo reticulado, e constituem-se nos mais influentes modelos de políticas de fluxo de informação.

2.3.1 Modelo de Bell-LaPadula

O *Modelo de Bell-Lapadula* (BELL; LAPADULA, 1973) destina-se a descrever políticas de confidencialidade, tendo sido um dos primeiros a versar acerca de controle de acesso mandatário. No modelo, sujeitos e informações são classificados de acordo com diferentes níveis de segurança e categorias. Pode-se descrevê-lo formalmente, de maneira similar ao feito em Bishop (2003), tendo como base os seguintes elementos:

S : Conjunto de sujeitos do sistema

O : Conjunto de objetos do sistema

N : Conjunto totalmente ordenado de níveis de segurança

C : Conjunto de categorias

$R = N \times 2^C$: Conjunto de rótulos de segurança

$r : S \cup O \mapsto R$: Função que mapeia um rótulo de segurança a sujeitos e objetos

$dom : R \mapsto R$: Relação de ordem parcial em R que forma no conjunto um reticulado, definida de maneira tal que $(n, c) dom (n', c') \iff n \leq n' \text{ e } c' \subseteq c$

Os acessos de leitura e escrita são então autorizados segundo duas diretrizes:

1. Condição de segurança simples: um sujeito s é autorizado a ler um objeto o se e somente se $r(s) \text{ dom } r(o)$
2. Propriedade *: um sujeito s é autorizado a escrever num objeto o se e somente se $r(o) \text{ dom } r(s)$

Em termos práticos, essas definições implicam que um sujeito não pode ler uma informação cujo nível de segurança seja maior do que o seu próprio, e/ou cujo conjunto de categorias não esteja contido no conjunto que lhe diz respeito (“*no read-up*”); e um sujeito não pode escrever num objeto cujo nível de segurança seja inferior ao seu, ou cujo conjunto de categorias não seja superconjunto do conjunto de categorias que lhe diz respeito (“*no write-down*”).

2.3.2 Modelo de Biba

O modelo de Biba (1977) é análogo ao de Bell-Lapadula, destinando-se entretanto à modelagem de políticas de integridade. Sendo assim, os sujeitos e objetos são categorizados em diferentes níveis de integridade, que expressam a confiança na correção das informações criadas ou armazenadas por eles.

Utilizando os mesmos elementos da definição 2.3.1 (com N , porém, expressando um conjunto de níveis de integridade), os acessos de leitura, escrita e execução são então autorizados de acordo com as diretivas:

1. Condição de segurança simples: um sujeito s é autorizado a ler um objeto o se e somente se $r(o) \text{ dom } r(s)$

2. Propriedade *: um sujeito s é autorizado a escrever num objeto o se e somente se $r(s) \text{ dom } r(o)$
3. Propriedade de execução: um sujeito s_1 é autorizado a executar outro sujeito s_2 se e somente se $r(s_1) \text{ dom } r(s_2)$

De acordo com essas definições, as diretivas “*no read-up*” e “*no write-down*” do modelo de Bell-Lapadula invertem-se no modelo de Biba, tornando-se “*no read-down*” e “*no write-up*”. Novamente em termos práticos, a integridade das informações é protegida ao se impedir que sujeitos leiam informações ou executem sujeitos de menor integridade e escrevem em objetos de maior integridade.

2.3.3 Modelo Descentralizado

O modelo descentralizado Myers e Liskov (1997) aproxima os mecanismos IFC do controle de acesso discricionário, possibilitando que classes de segurança representem os donos das categorias de informação e as concessões de acesso que esses especificam a elas.

Neste modelo, cada classe de segurança L possui um conjunto de proprietários $\{o_1, \dots, o_n\}$ (que são sujeitos do sistema), denotados por $owners(L)$. Para cada proprietário o_i de L , é especificado outro conjunto de sujeitos leitores $\{r_1, \dots, r_n\}$, denotados por $readers(L, o_i)$, que são os sujeitos a quem o_i concede permissão de leitura de L . O conjunto efetivo de leitores é formado pela intersecção do conjunto de leitores de todos os proprietários $readers(L, o_1) \cap \dots \cap readers(L, o_n)$. A relação $\sqsubseteq$ é definida como segue:

$$L_1 \sqsubseteq L_2 \leftrightarrow owners(L_1) \subseteq owners(L_2) \text{ e} \\ \forall o \in owners(L_1), readers(L_1, o) \supseteq readers(L_2, o)$$

A relação $\sqcup$, por sua vez, é definida como segue:

$$owners(L_1 \sqcup L_2) = owners(L_1) \cap owners(L_2)$$

$$\forall o \in owners(L_1 \sqcup L_2), readers(L_1 \sqcup L_2, o) = owners(L_1, o) \cap owners(L_2, o)$$

O modelo descentralizado especifica ainda que desclassificações podem ocorrer somente quando um sujeito adiciona leitores a uma classe de informação de sua propriedade ou retira-se do conjunto de proprietários de uma classe. Além disso, o modelo requer uma relação de “poder-agir-por” entre sujeitos, formando uma hierarquia de confiança entre eles. Dessa maneira, se um sujeito tem o direito de agir por outro, ele é capaz inclusive de realizar desclassificações em nome desse outro.

2.3.4 Rótulos de Categorias Disjuntas

Stefan (2012b) propõem um modelo capaz de expressar políticas de confidencialidade e de integridade. Segundo o que argumentam os autores, o modelo é mais simples do que o apresentado em (MYERS; LISKOV, 1997), descrito na seção 2.3.3. Diferentemente desse, não requer a especificação de uma hierarquia de confiança entre sujeitos, pois ela já decorre da natureza do modelo.

Um rótulo-DC (associado a uma classe de segurança nesse modelo) é um par $\langle S, I \rangle$ de fórmulas booleanas sobre os sujeitos do sistema (representados como as variáveis das fórmulas), tal que:

1. S e I são expressas na forma mínima como produto-de-somas, e seus elementos são ordenados de maneira tal que elas possuam representação única
2. Nenhuma delas possui um elemento negado

3. S denota os conjuntos de sujeitos capazes de ler a informação, enquanto I representa o conjunto de sujeitos autorizados a modificá-la

A relação $\sqsubseteq$ é definida entre duas classes $L_1 = \langle S_1, I_1 \rangle$ e $L_2 = \langle S_2, I_2 \rangle$ da seguinte maneira:

$$\frac{S_2 \implies S_1, I_1 \implies I_2}{\langle S_1, I_1 \rangle \sqsubseteq \langle S_2, I_2 \rangle} \quad (2.1)$$

Essa definição implica que a relação $\sqsubseteq$ se define entre duas classes quando as restrições de segurança permanecem as mesmas, ou se tornam mais rigorosas. Assim, a relação é definida entre S_2 e S_1 se os sujeitos em S_2 forem especificados já em S_1 , e os sujeitos em I_1 forem especificados em I_2 .

A operação $\sqcup$ é especificada no modelo da seguinte maneira:

$$L_1 \sqcup L_2 = \langle S_1 \wedge S_2, I_1 \vee I_2 \rangle^1$$

Pode-se interpretar que $L_1 \sqcup L_2$ dá direito de leitura àqueles sujeitos que podem ler tanto informações classificadas com L_1 quanto L_2 , e dá direito de modificação aos sujeitos que podem modificar informações classificadas com L_1 ou L_2 .

Por fim, nesse modelo é previsto um mecanismo de desclassificação similar ao modelo descentralizado, desta vez criando-se um novo operador $\sqsubseteq_P$. P denota uma fórmula booleana envolvendo sujeitos, que representa aqueles em nome dos quais um programa age.

$$\frac{P \wedge S_2 \implies S_1, P \wedge I_1 \implies I_2}{\langle S_1, I_1 \rangle \sqsubseteq_P \langle S_2, I_2 \rangle} \quad (2.2)$$

Se P contiver elementos, mesmo que $L_1 \not\sqsubseteq L_2$, é possível que $L_1 \sqsubseteq_P L_2$

¹Seria necessário ainda reduzir as fórmulas a produto de somas e reordenar os elementos

2.4 Considerações finais

Os principais conceitos discutidos nas demais seções deste capítulo são resumidos na tabela 1. Como considerações finais, cabe destacar que o emprego de mecanismos IFC em sistema computacional, apesar do potencial de obtenção de garantias de segurança mais robustas do que com mecanismos de segurança tradicionais, pode acarretar em impactos negativos ao sistema. Tais impactos impõe o estabelecimento de compromissos principalmente entre o grau de segurança requerido – mensurado através das garantias de não-interferência, como mostrado – e alguns aspectos práticos da construção e utilização dos softwares executados sob o controle do mecanismo, podendo-se destacar:

- a *compatibilidade* com funcionalidades diversas (isto é, a quantidade de funcionalidades que podem ser suportadas pelo mecanismo, sem que este julgue o programa que as implementa como inseguros);
- a *programabilidade* disposta aos desenvolvedores (isto é, a facilidade com que se constrói as funcionalidades compatíveis com a operação do mecanismo);
- impactos de desempenho e usabilidade decorrentes da execução sob os controles do mecanismo IFC.

Para analisar os compromissos impostos por diferentes implementações de mecanismos IFC sob os aspectos de compatibilidade e programabilidade, é útil entender a atuação dos mesmos como tarefas de classificação, isto é, tarefas de categorização de fluxos de informação como regulares ou irregulares. Assim sendo, a qualidade de um mecanismo IFC pode ser expressa através

Natureza de implementação	OS-IFC, RS-IFC e L-IFC
Canais de comunicação	Legítimos: mecanismos de comunicação inter-processos, I/O, manipulação de memória Encobertos: fluxos implícitos, temporização, terminação, recursos compartilhados, físicos
Garantias de segurança	Fraco sigilo, não-interferência insensível/sensível à temporização, terminação ou progresso, não-interferência possibilística e probabilística
Modelos de políticas de fluxo de informação	Bell-LaPadula, Biba, descentralizada e categorias disjuntas

Tabela 1: Resumo dos conceitos apresentados neste capítulo

de métricas tipicamente usadas para quantificar o desempenho de classificadores, como a *sensitividade* (no caso, expressando o percentual de fluxos irregulares assim classificados) e *especificidade* (a quantidade de fluxos irregulares classificados como tal). A sensitividade se relaciona às garantias de segurança fornecidas pelo mecanismo frente à proteção da variedade de canais de comunicação disponíveis aos sujeitos (como listado na seção 2.1.2). A especificidade, por sua vez, relaciona-se aos impactos negativos de utilização do mecanismo, tanto pela inviabilização de programas que, apesar de seguros, não são classificados como tal, quanto pela imposição da necessidade de reescrita de programas erroneamente classificados como inseguros.

De posse das métricas de sensitividade e especificidade e dos aspectos práticos listados, pode-se contrastar diferentes técnicas de implementação de mecanismos IFC ao se agrupá-las pela propriedade de granularidade de rastreamento. Assim, considerando-se mecanismos com granularidades de rastreamento em nível de processo ou thread, espera-se que os mesmos possuam baixa especificidade, pois são incompatíveis com funcionalidades que requeiram a manipulação de dados de diferentes classes de segurança por um único pro-

cesso. Ademais, nos casos em que as funcionalidades são compatíveis, pode ser necessário dividir o programa em diferentes módulos, que recebem rótulos de segurança diferentes. Note-se que essa prática, porém, acarreta impactos de desempenho, posto que com ela se reduz o potencial de compartilhamento de recursos obtido em um processo único. Entretanto, a utilização de mecanismos com essa granularidade de rastreamento tem potencial de compatibilidade com softwares já desenvolvidos (se mantidas as interfaces do sistema operacional ou de runtime assumido por esses softwares), bem como facilitam o fornecimento de garantias de segurança mais robustas – isto é, possuem maior sensibilidade –, ao possibilitar a definição de fronteiras entre os sujeitos de forma precisa, através de abstrações já bem conhecidas (notadamente no caso de implementações OS-IFC).

Mecanismos IFC com granularidade de rastreamento em nível de variável usualmente possuem maior especificidade, já que definem variáveis como receptáculos finais de informação. Assim, com esse tipo de mecanismo, não é necessário a divisão de programas em grandes módulos, fato que pode ter consequências positivas sobre os impactos de desempenho e programabilidade acarretados pelos mecanismos. Entretanto, deve-se notar que há diferenças adicionais em função da técnica de implementação dos mesmos, isto é, RS-IFC ou L-IFC. No primeiro caso, os impactos de desempenho podem ser mais significativos – tendo-se em vista que o rastreamento ocorre em runtime –, mas ganha-se a possibilidade de haver a retro-compatibilidade com softwares legados (caso seja mantida a interface do sistema de runtime assumida pelos mesmos). No segundo caso, os impactos de desempenho tendem a ser menores, uma vez que a tarefa de verificação é passada para tempo de compilação; porém, exige-se a reescrita ou recompilação de qualquer software a ser controlado (usualmente, numa linguagem específica com tipos de segurança),

o que confere a L-IFC o menor grau de retro-compatibilidade com softwares legados e, possivelmente, o maior grau de impactos de programabilidades. Some-se a isso o fato de mecanismos IFC, por terem atuação limitada ao tempo de compilação, usualmente são obrigados a tomar decisões conservadoras acerca da regularidade dos fluxos de informação detectados (no limite, a eficiência de tal análise é restrita pelo teorema de Rice (RICE, 1953)), o que tem efeitos negativos sobre o potencial de especificidade desse tipo de mecanismo. Por fim, note-se que fornecer garantias de segurança robustas com mecanismos dessa granularidade, independentemente da técnica de implementação, constitui-se numa tarefa mais complexa do que no caso anterior, já que, nesse caso, há potencialmente menos isolamento entre os receptáculos de informação, e a fronteira entre eles não é tão bem definida.

Frente aos compromissos expostos, os projetistas de mecanismos IFC tradicionalmente estabelecem diferentes cenários de ataque que determinam quais canais de fluxo de comunicação podem ser utilizados para transmitir informação ilegalmente. Canais que possuem pequena taxa de transmissão, ou que serão dificilmente utilizados por um sujeito benigno, são usualmente ignorados, com o intuito de obter-se um compromisso aceitável frente às limitações das técnicas de implementação de mecanismos IFC empregadas.

3 TRABALHOS RELACIONADOS

Muito embora os mecanismos IFC, desde sua proposição inicial nos anos 70 até os dias presentes, não tenham experimentado adoção significativa em sistemas computacionais comerciais, inúmeros trabalhos propõem a utilização dos mesmos em diferentes sistemas, no intuito de aprimorar a confidencialidade e a integridade de informações oferecidas pelos mesmos. Neste capítulo, detalham-se as aplicações de mecanismos IFC em dois cenários: dispositivos móveis e aplicações Web. Assim, ilustra-se o estado da arte em ambos os cenários, que são de fundamental importância para a contextualização da proposta apresentada no capítulo 4. Em ambos os casos, analisam-se as propostas sob os aspectos apresentados na seção 2.4, isto é, as garantias de segurança fornecidas, e os impactos na compatibilidade, programabilidade e desempenho.

3.1 IFC em Aplicações Móveis

Dispositivos móveis são computadores pessoais que podem ser distinguidos, em termos de hardware, pela maleabilidade e interface de toque, e pela presença, num invólucro único, de numerosos tipos de componentes sensoriais (como acelerômetros, giroscópios, bússolas, termômetros, barômetros, sensores de luminosidade e umidade), componentes de comunicação (como de telefonia celular, SMS, Wifi IEEE 802.11, Bluetooth IEEE 802.15, NFC,

transceptores infravermelho, GSM, GPRS, UMTS, HSPA+), GPS, GPUs, câmeras, microfones e alto-falantes. Os sistemas operacionais dos dispositivos móveis, por sua vez, são especializados no gerenciamento e abstração desses componentes de hardware específicos – muitas vezes, assumindo características de sistemas operacionais de tempo real –, pertencendo a um conjunto que majoritariamente abrange, na época de escrita deste texto, Android, iOS e Windows Phone (IDC INC., 2015).

Por sua vez, o desenvolvimento de aplicações para dispositivos móveis – os usualmente denominados *aplicativos móveis* – difere, em alguns aspectos, do desenvolvimento tradicional de aplicações para computadores pessoais. Primeiramente, existem disparidades no padrão de interface homem-máquina entre os dois tipos de dispositivos, sobretudo quanto ao tamanho e à interface de toque como meio predominante para entrada de dados. Além disso, a disponibilidade do conjunto de componentes de hardware anteriormente descritos amplia o conjunto de funcionalidades passíveis de implementação pelos desenvolvedores de aplicativos, sobretudo no que se refere a computação contextual. Ademais, a conectividade quase que permanente dos dispositivos móveis, aliada à grande quantidade de dados capaz de ser produzida por seus sensores – acrescida de suas capacidades de armazenamento muitas vezes limitadas – combina-se harmoniosamente com as tendências de utilização de *cloud computing*.

Uma das diferenças mais notórias, porém, entre o desenvolvimento de aplicações tradicionais e de aplicativos móveis advém das restrições de segurança sob as quais os últimos são executados, limitando o acesso a dados e serviços no dispositivo. Assim, para acessar uma determinada categoria de dados sensíveis (por exemplo, dados de leitura do GPS, ou a lista de contatos do aparelho), ou utilizar um serviço possivelmente pago (como o serviço de

telefonía), os aplicativos devem ser explicitamente autorizados a fazê-lo pelo usuário, que concede a cada aplicativo um conjunto de permissões. Além disso, cada aplicativo tem por padrão acesso de leitura e escrita a um diretório privado, onde a integridade e confidencialidade de seus próprios dados é, ao menos em teoria, assegurada contra a ação dos demais aplicativos. A comunicação inter-aplicativos é também restrita, de forma a se garantir, em princípio, o cumprimento de um conjunto simples de regras de controle de acesso, verificadas pelo sistema operacional, quando da sua ocorrência. Por fim, o modelo de distribuição de aplicativos baseada em lojas – que é característica dos dispositivos móveis – possibilita a submissão dos aplicativos a um processo de auditoria, no qual se pode verificar a adequação dos mesmos a um conjunto de restrições de segurança.

Muito embora as políticas e mecanismos de segurança em dispositivos móveis sejam melhor idealizados do que em computadores pessoais tradicionais, verifica-se que tais políticas e mecanismos são, em essência, pouco granulares. Por esse motivo, um aplicativo pode abusar das suas permissões de segurança, comprometendo a confidencialidade dos dados aos quais obtém acesso. Como exemplo, um aplicativo pode enviar a entidades externas dados de geolocalização do dispositivo (ou dados sensíveis de outro aplicativo), bastando, para isso, que o aplicativo possua permissões para acessar a internet, o GPS (ou consiga fazer com que o outro aplicativo compartilhe com ele dados sensíveis) (IBTIMES, 2016).

Diante desse cenário, diversos trabalhos foram propostos no intuito de aperfeiçoar os mecanismos de segurança de dispositivos móveis com o uso de IFC. Tais trabalhos utilizam-se das diferentes técnicas apresentadas no capítulo 3, propondo modificações a diferentes camadas de software – sistema operacional, sistema de runtime, ou linguagens de programação e compilado-

res – para rastrear a propagação de informações sensíveis entre aplicativos, e entre aplicativos e entidades externas, com diferentes garantias de segurança. Nas seções que se seguem, detalham-se alguns desses trabalhos, de forma a ilustrar o estado da arte na área.

3.1.1 Maxoid

Dispositivos móveis são dotados, como já mencionado, de diretórios privados de armazenamento, onde aplicativos podem manter dados sensíveis salvaguardados da leitura ou manipulação indevida. Em alguns casos, porém, é desejável que os aplicativos interajam e compartilhem dados sensíveis, como, por exemplo, quando um aplicativo cliente de emails necessita exibir arquivos PDF com o auxílio de outro, especializado nessa função. Nesse caso, o primeiro aplicativo deve disponibilizar ao segundo os conteúdos a serem exibidos, seja enviando-os diretamente através de alguma forma de comunicação inter-processo, ou escrevendo-os em um diretório de armazenamento público. Entretanto, o segundo aplicativo, sendo executado com o conteúdo sensível do primeiro, continua integralmente com suas permissões de segurança originais, podendo divulgar a terceiros tal conteúdo se as referidas permissões assim possibilitarem.

Maxoid, de Xu e Witchel (2015), é uma proposta de mecanismo OS-IFC que visa a proteção da confidencialidade e integridade de dados sensíveis em aplicativos móveis, precisamente no cenário de compartilhamento descrito. Na nomenclatura de Maxoid, quando um aplicativo requisita serviços de outro, o primeiro é chamado de “iniciador” e o segundo de “delegado”. O mecanismo de contenção de Maxoid é implementado como uma modificação do sistema de arquivos que altera o conjunto de arquivos que pode ser lido ou escrito por um aplicativo quando este é executado como delegado de algum outro. Esse

estado do sistema de arquivos é denominado de “view”. Quando um delegado B roda em nome do aplicativo A, B tem acesso às pastas privadas de tanto de B quanto de A, e às pastas públicas a que B tem acesso e a que A tem acesso. Qualquer alteração que B executar no sistema de arquivos é contida à view fornecida a B quando executado como delegado de A, que, inclusive, é livre para editá-las, assim que B terminar de executar. Isso, combinado ao fato de que B perde o acesso à rede quando executado como delegado, implica que qualquer dado enviado de A para B é confinado, não podendo sair dos limites da memória de B e dos arquivos fornecidos por ele na sua visualização. Note-se que B não obterá acesso a essa visualização ao ser executado de forma independente ou como delegado de outro aplicativo C.

Maxoid é implementado de forma a manter-se a retrocompatibilidade com aplicativos já existentes, executados no papel de delegados. Desse modo, funções já existentes das APIs de acesso a arquivos são mantidas, muito embora tenham sido adicionadas novas, que dão aos aplicativos melhor uso das funcionalidades de Maxoid (desde que os mesmos, porém, sejam modificados para tanto). Note-se que a retrocompatibilidade é mantida também para os aplicativos executados como iniciadores, mas eles devem ser alterados caso se deseje utilizar funcionalidades adicionais do Maxoid. Por fim, observe-se que o mecanismo é essencialmente incompatível, porém, com aplicativos delegados cujas funcionalidades dependam do acesso à rede, uma vez que essa permissão é removida dos mesmos.

De acordo com os autores, o sobrecusto imposto por Maxoid é pouco perceptível, dado que o mecanismo age somente quando estes executam I/O, e a maior carga de processamento de aplicativos móveis relaciona-se ao processamento interno e à interface homem-máquina. Em relação às garantias de segurança oferecidas pela proposta, muito embora o trabalho não contenha

sua formalização em termos de não-interferência, as restrições impostas aos aplicativos delegados indica que os mesmos não sejam capazes de utilizar fluxos implícitos, estando restritos à utilização de canais encobertos habilitados pelo compartilhamento de recursos.

3.1.2 π Box

π Box, de Lee (2013), tem por objetivo impedir que aplicativos móveis divulguem indevidamente informações sensíveis às quais têm acesso, sejam essas originárias dos próprios dispositivos (fornecidas tanto pelo sistema operacional, quanto pelo usuário) ou de sistemas remotos. Para tanto, π Box utiliza-se de um mecanismo OS-IFC que impede os aplicativos executados sob seu controle de, em termos simplificados, realizar comunicação inter-aplicativos, acessar através da rede qualquer outro sistema além de um servidor confiado pelo mecanismo, ou escrever em diretórios do sistema de arquivos que não sejam os seus próprios diretórios privados. Tais restrições equivalem a atribuir classes de segurança diferentes a cada aplicativo, tais que a relação $\sqsubseteq$ não seja satisfeita entre quaisquer duas delas.

O servidor mencionado, confiado por π Box, pode abrigar o código *server-side* dos aplicativos, porém restringindo também os privilégios de comunicação de módulos responsáveis por atender requisições de usuários diferentes (potencialmente, utilizando-se de mecanismos IFC como descritos na seção 3.2). Assim, nesse servidor, são fornecidas aos aplicativos as funcionalidades de armazenamento e comunicação inter-usuários ou desenvolvedor-usuários através de mecanismos específicos. As funcionalidades de armazenamento são executadas por um módulo de armazenamento privado (que é isolada por um rótulo particular para o par aplicativo/usuário) e um de armazenamento estático, que é de somente leitura e é acessível por todos os usuários. Esse

último módulo pode ser utilizado pelos desenvolvedores de aplicativos para fornecer conteúdo visível a todos os usuários. Os usuários, por sua vez, podem trocar mensagens entre si através de um módulo de compartilhamento de dados, ou receber notificações dos desenvolvedores. π Box oferece também a possibilidade dos desenvolvedores extraírem estatísticas de uso de seus aplicativos (para, por exemplo, contabilizar a exibição de anúncios), sem que haja o vazamento de dados privados dos usuários nos relatórios contendo tais estatísticas. Para garantir a anonimidade dos dados divulgados, π Box utiliza algoritmos de *privacidade diferencial sob observação contínua*.

Para dar suporte às funcionalidades presentes no servidor, π Box especifica dois serviços presentes nos dispositivos móveis, dando suporte aos aplicativos. Um deles fornece um mecanismo de autenticação para os aplicativos se comunicarem com o servidor. Outro serviço intercepta requisições de compartilhamento de dados entre usuários, assumindo o controle da execução quando da ocorrência de uma delas. Um aplicativo confiável, então, exibe o conteúdo e destinatário da mensagem ao usuário, que pode conferir a adequação da mesma (muito embora um aplicativo malicioso possa enviar dados privados a outros usuário através de esteganografia).

O mecanismo descrito, como observam seus próprios autores, não é compatível com todas as categorias de funcionalidades de aplicativos. Em especial, aplicativos colaborativos, redes sociais, de comunicação e compartilhamento de conteúdo são incompatíveis com a arquitetura de π Box, devido às restrições de comunicação impostas. A retro-compatibilidade também é afetada nos casos em que aplicativos necessitem acessar sistemas remotos, exigindo-se não só que os aplicativos sejam alterados para fazê-lo sob as restrições no dispositivo, como também que o lado do servidor seja adaptado para funcionar sob as restrições do servidor confiável, e somente nele sejam

executados. Apesar de tais restrições, a arquitetura de π Box adequa-se muito bem em cenários corporativos, em que empresas permitem a utilização de seus dados em aplicativos de terceiros arbitrários nos aparelhos dos funcionários, mas sem incorrer nos riscos de privacidade associados a essa prática.

Os autores de π Box não formalizam em seu trabalho as garantias de segurança oferecidas pela plataforma, muito embora possa se argumentar que, devido às restrições de comunicação por ela impostas aos aplicativos, os mesmos sejam incapazes de utilizar fluxos implícitos, restando aos mesmos a utilização de canais encobertos habilitados pelo compartilhamento de recursos. Em termos de sobrecustos de desempenho, os autores argumentam que os efeitos são imperceptíveis aos usuários, muito embora tenham estudado com rigor somente os efeitos do mecanismo server-side.

3.1.3 TaintDroid

TaintDroid (ENCK, 2010) utiliza-se de um mecanismo RS-IFC para empreender o rastreo de dados sensíveis dentro de aplicativos móveis no sistema operacional Android, e evidenciar o envio dos mesmos dados através da rede. A granularidade de rastreo empregada é em nível de variável, e o mecanismo é implementado como adição ao *middleware Dalvik VM*, em que aplicações Android são executadas, e que é derivado da tecnologia Java. O mecanismo define rótulos capazes de distinguir entre 32 diferentes possíveis fontes de dados sensíveis, dentre sensores, bases de dados (de SMS, contatos) e APIs de acesso a informações do aparelho. Ao adentrar um aplicativo, os dados vindos de uma dessas fontes são rastreados, como tradicionalmente em mecanismos desse nível de granularidade, através de comandos lógico-aritméticos, atribuições, chamadas de procedimento, entre outros. A comunicação inter-aplicativos no TaintDroid é modificada de forma a preservar os rótulos

de segurança de dados trocados entre diferentes aplicativos, bem como o sistema de arquivos é alterado para associar rótulos a arquivos, mantendo-se o registro da classe de segurança dos dados neles armazenados.

A operação do mecanismo IFC de TaintDroid foi projetada de forma a ser transparente para os aplicativos executados sob seu controle, não introduzindo incompatibilidades com a maioria dos aplicativos já existentes (com a exceção de, em alguns casos, aplicativos que utilizem JNI (ORACLE INC., 2014) para executar código nativo). TaintDroid também não impede a execução de funcionalidades que de fato requeiram o envio de dados sensíveis através da rede (considerando-se que o objetivo da proposta é, primordialmente, apenas evidenciar o envio, muito embora uma atuação imprecisa possa levar à ocorrência de falsos-positivos em demasia). Entretanto, a utilização de RS-IFC com granularidade de rastreo em nível de variável atribui a TaintDroid sobrecargas de execução maiores, se comparados comparação aos de outros trabalhos apresentados nesta seção. Porém, os autores afirmam que o mecanismo introduz prejuízos de somente 14% em relação à execução regular (segundo métricas e aferições dos mesmos), não acarretando em quedas perceptíveis de usabilidade dos aplicativos.

Por fim, note-se que TaintDroid não fornece garantias de não-interferência, sendo vulnerável a fluxos implícitos e outros canais encobertos. Essa característica, porém, é compatível com o modelo de atacante assumido pelos autores, em que se desconsidera o uso desses canais. Ainda assim, a natureza dinâmica do mecanismo introduz a necessidade de geração de execuções de diferentes *traces* dos aplicativos para evidenciar todos os fluxos ilegais, mesmo que executados por canais legítimos.

3.1.4 FlowDroid

FlowDroid (ARZT, 2014) propõe um mecanismo de rastreamento de informações sensíveis em aplicações Android similar ao proposto para TaintDroid, estabelecendo um conjunto de fontes onde tais dados podem se originar. Entretanto, FlowDroid opera de modo estático, rastreando a propagação de dados sensíveis desde suas fontes até destinos considerados inseguros (como a rede e diretórios de acesso público). O mecanismo empregado pode ser caracterizado como L-IFC na terminologia apresentada neste trabalho, muito embora sua operação se baseie na análise de código binário, já compilado, dos aplicativos.

O modelo de atacante assumido pelos autores do FlowDroid é o mesmo do TaintDroid e, portanto, o mecanismo IFC empregado não fornece garantias de não-interferência, sendo vulnerável a fluxos implícitos e demais canais encobertos. Entretanto, por operar de forma estática, FlowDroid é capaz de antecipar todos os fluxos ilegais passíveis de serem executados pelo aplicativo – algo que, como já mencionado, o TaintDroid é incapaz de fazer.

3.1.5 Análise Comparativa

A tabela 2 resume as características dos trabalhos apresentados nesta seção, estabelecendo um panorama comparativo em função de quatro aspectos: a técnica de implementação, as garantias de segurança, a retro-compatibilidade com aplicações já escritas (e a compatibilidade com funcionalidades, de modo geral) e os impactos de desempenho esperados. Em relação às garantias de segurança, atribuiu-se caracterização forte aos mecanismos que utilizam granularidade de rastreo mais grosseira, em razão do isolamento mais terminante entre os sujeitos (a despeito da ausência de tratamento formal nos

	Técnica de Implementação	Garantias de Segurança	(Retro-)Compatibilidade	Impactos de Desempenho
Maxoid	OS-IFC	Fortes	Média	Pequenos
π Box	OS-IFC	Fortes	Pouca	Pequenos
TaintDroid	RS-IFC	Fracas	Alta	Altos
FlowDroid	L-IFC	Fracas	Alta	Nenhum

Tabela 2: Comparação dos trabalhos selecionados de IFC em aplicações móveis

trabalhos); e caracterização fraca aos demais, por não tratar fluxos implícitos. Em relação à compatibilidade, atribuiu-se caracterização alta ao FlowDroid e ao TaintDroid, pois ambos não impedem a utilização de aplicativos já desenvolvidos, e introduzem poucos impactos de programabilidade. Nesse quesito, Maxoid recebeu caracterização média por exigir a especialização dos aplicativos utilizados como iniciadores, e ser incompatível com aplicativos dependentes de acesso a rede, executados como delegados; para o π Box, a caracterização menor deve-se às restrições de comunicação e à necessidade de adaptação do código do lado do cliente. Quanto aos impactos de desempenho, atribui-se nenhuma a FlowDroid, em função de sua natureza estática; pequenas, a Maxoid e π Box, devido à sua atuação episódica em runtime; e médios a TaintDroid, pois, apesar da sua atuação em tempo de execução (o que lhe conferiria altos impactos de desempenho), os sobrecustos de execução são ainda pequenos (segundo medições de seus autores).

3.2 IFC em Aplicações Web

Aplicações Web são aplicações distribuídas de acordo com o modelo cliente-servidor. Os servidores provêm serviços e recursos a todos os clientes, frequentemente interagindo com outros sistemas (como gerenciadores de bases de dados e mainframes), que usualmente detêm, de fato, os recursos e serviços aos quais uma aplicação Web dá acesso. Normalmente, os servidores são acessíveis publicamente através da Internet. Os clientes, por sua vez, são os *Web browsers*, que renderizam o conteúdo de páginas HTML (*HyperText Markup Language*) frequentemente estilizadas através de CSS (*Cascading Style Sheets*), exibem imagens e reproduzem áudio e vídeo referenciados nessas páginas. Os browsers também dotam as páginas de comportamento dinâmico, ao executar código Javascript contido nelas, ou ao iniciar a execução de plug-ins para exibir alguns dos conteúdos nelas referenciados.

A comunicação entre clientes e servidor numa aplicação Web se dá através de pares de mensagens de requisição/resposta HTTP. Dado que o protocolo HTTP não é dotado de estado, o servidor depende da colaboração dos clientes para manter o estado da execução de cada um deles. Para tanto, cada cliente recebe um valor denominado *cookie* de sessão, a ser enviado em cada requisição. O session cookie identifica univocamente o navegador no contexto da execução da aplicação Web, o que permite associá-lo a um conjunto de valores denominado *sessão*. Esses valores são utilizados para, entre outros, implementar os controles de fluxo e de acesso da aplicação.

Como em toda aplicação cliente-servidor, o cliente guia a execução de uma aplicação Web. Em específico, o browser, em função das entradas do usuário e dos próprios códigos e dados recebidos do servidor, dispara requisições a esse. No servidor, o processamento de uma requisição HTTP pode ser

entendida como a execução de uma rotina p que recebe como entrada os parâmetros da requisição e os dados associados à seção do cliente associado. Para produzir uma resposta, p ainda pode ler valores de variáveis globais do servidor e consumir dados e processamento de outros sistemas, e efeitos colaterais desse processamento podem surgir tanto no próprio servidor quanto nesses sistemas.

No que se refere à implementação de controle de fluxo de informação, dois aspectos das aplicações Web se notabilizam: a pluralidade de políticas e a natureza distribuída. Quanto ao primeiro aspecto, observa-se que as políticas de fluxo de informação em aplicações Web podem geralmente ser expressas em termos de relações de posse entre dados e membros de conjuntos de usuários dinamicamente variáveis (que pode incluir, inclusive, sistemas), e que a complexidade de tais políticas varia das mais simples (como as encontradas em sites de *e-commerce* ou de *internet-banking*) às mais intrincadas (por exemplo, como as encontradas em redes sociais, em que os dados podem pertencer a diferentes subconjuntos de usuários). Devido à capacidade de representar justamente esse tipo de relação, e com expressividade adequada, o modelo descentralizado assume um papel de destaque no contexto das aplicações Web. Quanto ao aspecto da natureza distribuída, nota-se que qualquer aplicação que a possua – como uma aplicação Web – contém canais de comunicação em todos os nós que a executam, o que possibilita à informação entrar na aplicação, ou sair dela, por qualquer um dos nós, propagar-se dentro deles e ser intercambiada entre eles (uma definição das propriedades de segurança para aplicações distribuídas pode ser encontrada em Gorrieri, Martinelli e Matteucci (2009)). Para se implementar um política de controle de fluxo de informação em tal arranjo, um mecanismo IFC deve regular não somente os fluxos localmente em cada nó, mas também globalmente, assegu-

rando que a informação é corretamente rotulada quando trocada entre nós e que o rótulo do contexto da execução em que a troca ocorre é mantido (o que impede a manifestação de uma versão distribuída de fluxos implícitos).

Devido às diferenças entre os ambientes de runtime de clientes e servidores, a exposição dos trabalhos relacionados nesta seção é a seguir dividida em duas sub-seções. Tal divisão corresponde à divisão usualmente observada na literatura acerca da implementação de mecanismos IFC em aplicações Web. No entanto, dá-se maior ênfase aos mecanismos IFC de servidores, cuja relevância é maior para a proposta apresentada no capítulo 4.

3.2.1 IFC em Web-Browsers

Browsers são atualmente verdadeiros sistemas de middleware, permitindo que as aplicações web não só possuam interfaces gráficas sofisticadas, como também executem partes significativas de seus algoritmos no próprio cliente. Tal capacidade contemporânea dos browsers difere da que eles possuíam originalmente – quando essencialmente só exibiam conteúdo estático –, tendo sido arquitetada ao longo da história da web. Em especial, a interpretação de Javascript nos browsers possibilitou às aplicações: manipular os elementos gráficos das páginas, e associar *listeners* de eventos de interação aos mesmos, através da API *DOM* (*Document Object Model*); executar rotinas e requisições HTTP (*Hypertext Transfer Protocol*) assincronamente; armazenar dados de maneira não-volátil através de APIs de persistência (*webstorage*); desenhar elementos gráficos complexos através de APIs de desenho; dentre outras funcionalidades. Em adição à interpretação de Javascript, os browsers são capazes ainda, como já mencionado, de reproduzir áudio e vídeo, prover a estilização de páginas com CSS e executar plugins.

A segurança na execução de aplicações Web em browsers naturalmente se fundamenta no isolamento entre objetos pertencentes a aplicações diferentes. Tal isolamento é baseado na *política de mesma origem*, que se constitui como modelo básico de segurança para a implementação de funcionalidades nos browsers. Nesse modelo, define-se origem como a tripla formada pelo protocolo, *host-name* e porta das URLs de origem dos objetos. Elementos como *cookies* de sessão, arquivos no *webstorage*, *frames*, funções Javascript, folhas de estilo CSS, dentre outros, são associados a uma dessas triplas, e o browser impede o acesso a um desses se a origem do acessante difere da do objeto acessado.

Os mecanismos de segurança presentes nos browsers têm se mostrado, porém, insuficientes diante da estruturação atual das aplicações Web, largamente baseada no compartilhamento de código e dados entre origens diferentes (DE RYCK, 2012). Tal compartilhamento tem como propósito a implementação de páginas denominadas *mash-ups*, que podem tanto agregar dados e funcionalidades de diferentes sistemas (como serviços de mapas, comentários, redes sociais e de anúncios) quanto bibliotecas de código Javascript (por exemplo, *jQuery*). A natureza discricionária dos mecanismos de segurança dos browsers, porém, obriga os desenvolvedores de aplicações a confiar plenamente seus dados a outros possivelmente desconhecidos, podendo-se comprometer a confidencialidade e a integridade desses dados.

Nesse contexto, mecanismos IFC diversos têm sido propostos na literatura com o intuito de aperfeiçoar as garantias de segurança fornecidas por browsers ao executar código Javascript. Em função da característica dinâmica da linguagem – que possibilitando que os comandos a serem executados só sejam conhecidos em runtime – tais mecanismos têm, via de regra, natureza RS-IFC. Em relação à granularidade de rastreo, porém, os mecanismos IFC

em browsers fogem ligeiramente à classificação apresentada no capítulo 2, podendo ser em nível de variável (mais granulares) ou em nível de *frame* (mais grosseiros).

Os frames de um Web-browser são análogos aos processos de um sistema operacional e, por isso, mecanismos IFC com granularidade de rastreo em nível de frame são análogos a mecanismos OS-IFC que têm essa propriedade em nível de processo. Um *frame* em HTML corresponde a uma área retangular da interface do browser, onde se carrega o conteúdo de uma página. Frames podem ser aninhados de acordo com o conteúdo da página que exibem, e a cada frame – bem como aos elementos em seu interior – atribui-se a origem da página nele carregada. Apesar de uma página poder incluir frames de qualquer outra origem, a comunicação entre dois frames só é permitida se ambos forem da mesma origem, ou concordarem em um protocolo para intercambiar dados através da função Javascript *postMessage*. Esse artifício é justamente uma das maneiras através das quais aplicações Web podem intercambiar dados, havendo, para tanto, uma página raiz que carrega em um de seus frames outra página (por exemplo, de um serviço de mapas), e envia a ela ou dela recebe dados. No entanto, note-se que os frames podem divulgar irrestritamente os dados de outros aos quais obtêm acesso, o que exige necessariamente o estabelecimento de uma relação de confiança, que não é necessariamente justificada, entre as aplicações Web envolvidas nesse intercâmbio. Sendo assim, trabalhos como COWL (STEFAN, 2014) e BFlow (YIP, 2009a) propõem a associação aos frames de rótulos de segurança provenientes de políticas de modelo descentralizado (2.3.3), denotando a origem dos dados manipulados pelos mesmos. Em ambos os trabalhos, a capacidade de comunicação dos frames é limitada em função de seus rótulos.

Mecanismos IFC com granularidade de rastreo em nível de variável têm

usualmente o objetivo de proteger aplicações que inserem código de terceiros em suas próprias páginas. Essa prática representa risco maior do que no caso anterior porque, nesse caso, atribui-se ao código inserido a mesma origem da página insersora (SPETT, 2005), tendo o código inserido a capacidade de manipular inteiramente os dados e o próprio funcionamento da página insersora. Assim, trabalhos como JSFlow (HEDIN, 2014) e FlowFox (DE GROEF, 2012) propõem a associação de rótulos de segurança, também de políticas descentralizadas (2.3.3), às variáveis de código Javascript, identificando-se a origem dos dados por elas armazenadas. Com isso, pode-se prevenir o vazamento de informações a origens não autorizadas em requisições futuras.

3.2.2 IFC em Servidores de Aplicações Web

Servidores de aplicações Web (ou simplesmente *Web servers*, como doravante designados) são sistemas de middleware que têm o propósito tanto de executar aplicações Web, como de facilitar o desenvolvimento e gerenciamento da execução das mesmas. Para tanto, Web servers são projetados de modo a abstrair detalhes comuns de funcionamento das aplicações Web que são independentes das suas funcionalidades, estando via de regra associados a frameworks de software que incluem APIs para, por exemplo, tratamento de requisições HTTP, gerenciamento de sessões, formatação de HTML e conexão a sistemas de bases de dados. Além disso, Web servers são também dotados de mecanismos de otimização de desempenho, como *pools* de recursos (por exemplo, de conexões à base dados ou de threads).

A despeito das abstrações fornecidas por Web servers e pelos frameworks associados, aplicações Web falham frequentemente na implementação de suas políticas de segurança server-side, como evidenciado em relatórios recentes (WHITE HAT SECURITY, 2013; TRUSTWAVE, 2014). Vulnerabilidades como

vazamento de dados, *cross-site scripting* (XSS) e *SQL injection* figuram entre as mais frequentes, colocando em risco a confidencialidade e integridade dos dados manipulados pelas aplicações Web.

Nesse cenário, numerosos trabalhos foram produzidos propondo a utilização de mecanismos IFC na melhoria da segurança de aplicações Web. Muitos deles sugerem a aplicação de formas simplificadas de IFC para a manutenção de integridade de comandos sensíveis utilizados pelas aplicações, como consultas SQL ou o próprio conteúdo de suas páginas (cujas integridades são comprometidas, respectivamente, em ataques de SQL injection e XSS). Nessa configuração, os mecanismos IFC associam aos dados provenientes de clientes rótulo u (do exemplo 2), e aos parâmetros de comandos sensíveis, rótulo r , de forma que sejam necessárias operações de endosso para que tais parâmetros possam receber dados originários de clientes. Desse modo, os mecanismos IFC atuam, na verdade, como ferramentas de auxílio à programação segura, explicitando o fluxo de informação indevido – que pode ser, de fato, difícil de evidenciar manualmente nas tecnologias de desenvolvimento Web, nas quais a distinção entre dados e comandos é comumente intrincada.

Figura 6: Arquitetura Conceitual de um Web server equipado com IFC

Esta seção concentra-se em trabalhos que propõem a utilização de formas mais sofisticadas de mecanismos IFC em aplicações Web, que funcionam de fato como mecanismo de segurança – podendo alguns, inclusive, atuar na proteção de informações contra próprias aplicações Web maliciosas. Nesses casos, independentemente da técnica de implementação usada, os servidores Web equipados com mecanismos IFC apresentam um conjunto comum de características, seguindo aproximadamente a arquitetura conceitual apresentada na Figura 6. Nela, os *Sistemas Internos* (como sistemas de bancos de dados ou de *mainframe*) mantêm os dados sensíveis e/ou executam serviços aos quais aplicações Web provêm acesso. Os *Sistemas Externos* são sistemas mantidos por terceiros, e provêm funcionalidades utilizadas pelas aplicações Web, em suas porções client-side ou server-side. Na figura, dentro do servidor Web, os blocos cinzentos denotam módulos executados sob as restrições do mecanismo IFC, enquanto que os blocos em branco denotam módulos que provêm interface e suporte ao mecanismo IFC, e são, portanto, parte da base de código confiável do servidor. O *Front-End* intercepta requisições HTTP e associa a elas autoridades de usuários dos sistemas internos, caso esses usuários já tenham se autenticado nas seções associadas às requisições (caso contrário, o *Front-End* pode requisitar deles tal autenticação, ou associá-los a uma autoridade pública, não identificada). O *Front-End* pode ainda associar rótulos denotando baixa integridade aos dados anexados às requisições, como anteriormente, para também fazer o rastreio da integridade dos dados. O tratamento de uma requisição, uma vez associada a uma autoridade, é delegado a um *Módulo de Aplicação*, que é executado sob as restrições do mecanismo IFC. Ao tratar a requisição, o módulo pode consumir dados ou serviços dos *Sistemas Internos* através do *Gateway Interno*, que associa rótulos a dados vindos desses sistemas. Associações de rótulos semelhantes são feitas pelo *Gerenciador de Sessões* e pelo *Gateway Externo*,

que provêm funcionalidades de persistência de dados entre requisições e comunicação com os *Sistemas Externos*, respectivamente. Operações de desclassificação e endosso podem ser conduzidas somente nos módulos *Desclassificadores/Endossadores*, de modo a que os efeitos dessas operações sejam controlados. Ao final do tratamento de uma requisição, o *Front-End* pode ainda inspecionar a resposta de forma a verificar a adequação do código client-side à política de controle de fluxo de informação.

Um servidor Web com tal arquitetura desonera as aplicações da responsabilidade de preservar a confidencialidade e integridade de informações, e a atribui ao mecanismo IFC e a seus módulos de suporte. Com isso, possibilita-se que o conjunto formado pelo servidor e as aplicações tenha um TCB (ou *trusted computing base*¹) menor, independente de aplicação e mais facilmente inspecionável e manutenível. Assim, dado que a informação é corretamente rotulada nos módulos de suporte, e que esses também atribuem a autoridade correta a entidades externas, as políticas de fluxo de informação podem ser corretamente exercidas, independentemente de inadequações das aplicações, como bugs, vulnerabilidades ou mesmo código malicioso.

Por fim, note-se que, para a proteção de informações contra aplicações possivelmente maliciosas, torna-se imperativo verificar a adequação do código client-side produzido pelas mesmas. Isso é necessário porque, através do cliente, as mesmas podem transmitir informações diretamente a sistemas externos, ou re-submetê-las em novas requisições, como parâmetros de classe de segurança inferior. Assumindo-se um cenário de funcionamento em que os clientes são browsers tradicionais, sem mecanismos de IFC – como todos os trabalhos a serem detalhados nessa seção assumem –, a verificação de adequação de saída das aplicações pode ser feita pelo módulo de *Front-End*.

¹Porção de código dedicada a mecanismos de segurança (BISHOP, 2003)

Dadas essas considerações, detalham-se a seguir quatro trabalhos que propõem a utilização de mecanismos IFC em servidores Web.

3.2.2.1 Asbestos Ok Web Server

O funcionamento do *Asbestos Ok Web Server* baseia-se nos mecanismos OS-IFC do Asbestos OS (EFSTATHOPOULOS, 2005), um protótipo de sistema operacional que aplica políticas de modelo descentralizado definidas pelos próprios processos. Os mecanismos do sistema operacional são utilizados pelo servidor Web para isolar seções de código executadas com autoridades de diferentes usuários, de forma que informações pertencentes a um usuário não sejam acessadas por seções de códigos executando com autoridade de outros usuários.

No Asbestos OS, cada processo P é rotulado com um rótulo de envio P_S e um rótulo de recebimento P_R . Esses rótulos são definidos como funções $L(h)$, com domínio em um conjunto de categorias (*Handles*) e contra-domínio num conjunto de níveis (*Levels*). Os *Handles* são números aleatórios sorteados pelo sistema operacional a pedido dos processos. Os processos, por sua vez, são responsáveis por atribuir significado aos rótulos, podendo representar, por exemplo, as classes de segurança de diferentes usuários. O conjunto de *Levels* é ordenado através de uma relação que denota diferentes graus de privilégio, sendo o privilégio de “desclassificação” o mais privilegiado, e o de “contaminação”, o menos, com diferentes níveis entre eles. Tais níveis são convenientes para a implementação de diferentes políticas de confidencialidade e integridade. A relação “pode-fluir-para” ($\sqsubseteq$ entre dois rótulos L_A e L_B é definida se $L_A(h) \leq L_B(h)$, para todo *Handle* h definido no sistema. De modo similar, a relação de combinação de classes ($\sqcup$) entre dois rótulos L_A e L_B é definida de modo que $L_A \sqcup L_B(h) = \max(L_A(h), L_B(h))$.

A aplicação desse modelo funciona, em termos simplificados, de acordo com o descrito na seção 2.1.1: um processo P (rotulado com P_S e P_R) pode se comunicar com outro processo Q (rotulado com Q_S e Q_R) apenas se $P_S \sqsubseteq Q_R$. Depois da comunicação, $Q_S(h)$ pode ser atualizado para $Q_S(h) = Q_S(h) \sqcup P_S(h) = \max(Q_S(h), P_S(h))$ (para todo *Handle* h definido no sistema), ou permanecer inalterado, se Q possuir direito de desclassificação sobre h . O processo criador de um *Handle* h recebe o direito de desclassificação sobre o mesmo, tornando-se capaz de ajustar os rótulos de envio e recebimento de outros processos em relação a h .

Como em todo mecanismo OS-IFC, a granularidade de rastreo grosseira força desenvolvedores a compartimentar seus programas em blocos isolados com diferentes classes de segurança, reduzindo-se o compartilhamento de recursos e, com isso, o desempenho. Para mitigar tal efeito, o Asbestos OS introduz um novo tipo de sujeito denominado *Event Process*, que são cópias de um processo-pai, tendo um diferente espaço de endereçamento, mas capazes de compartilhar páginas de memória entre si e com o processo-pai. Assim, alcança-se isolamento entre o processo-pai e seus *event processes*, reduzindo-se a sobrecarga na utilização de memória.

No Asbestos Ok Web Server, o código das aplicações são executados por *workers*, que são entidades do tipo *event process*. Cada *worker* é rotulado com um único *handle*, associado a um usuário da aplicação Web. Um processo chamado *demux* recebe os pedidos de conexão e, se o usuário associado a ela encontra-se já autenticado, entrega o tratamento da mesma ao respectivo *worker*; se não, o *demux* o entrega a um processo denominado *idd*, responsável por autenticar o usuário e criar um novo *worker* rotulado com um *handle* associado à autoridade do usuário, para tratar suas futuras requisições. Um processo denominado *db-proxy* gerencia conexões a bases de

dados, cujos dados estão também associados a usuários da aplicação, de modo que o processo possa verificar a autoridade necessária dos *workers* ao efetuar consultas, e rotular corretamente os dados resultantes delas. Outros processos *declassifiers* são usados para desclassificar dados dos usuários (isto é, remover o rótulo de segurança), mas somente do usuário em nome do qual eles são executados. Os processos *demux*, *idd* and *db-proxy* são todos considerados confiáveis, ao passo que os *workers* não o são, e os *declassifiers* são, por sua vez, considerados semi-confiáveis.

Como se vê, a arquitetura do Asbestos Ok Web Server é muito similar à arquitetura da Figura 8. Os processos *demux* e *idd* executam as funcionalidades do módulo de *front-end* (exceto pela análise do código client-side), enquanto que os módulos da aplicação são executados por *workers* e *declassifiers*. A funcionalidade do *session manager* é executada naturalmente pelo sistema operacional, ao manter o estado de execução dos *workers*. O *db-proxy* provê as funcionalidades do *internal gateway*, enquanto que não há módulos que executam as funcionalidades do *external gateway*, muito embora essas poderiam ser implementadas da mesma maneira que o módulo anterior.

Dentre outros trabalhos que propõem o uso de mecanismos OS-IFC para a segurança de aplicações Web, pode-se listar HiStar (ZELDOVICH, 2006) e Flume (KROHN, 2007). Ambos são derivados de Asbestos, e como esse, propõem o exercício de políticas de fluxo de informação pelo sistema operacional a partir das definições dos seus processos. HiStar implementa mecanismos IFC em uma camada de mais baixo nível (em relação a Asbestos), de modo a reduzir o *trusted computing base* dos mesmos. Flume, por sua vez, implementa seus mecanismos OS-IFC através de um programa em nível de usuário, o que mantém a interface padrão do OS e, portanto, preserva a compatibilidade com programas e sistemas de programação já existentes.

3.2.2.2 SafeWeb

SafeWeb (HOSEK, 2011) é um middleware Ruby especializado na execução de aplicações Web e equipado com mecanismos RS-IFC. Os módulos de aplicação no SafeWeb podem ser de dois tipos: unidades de *Event Processing* (EPU's) e unidades do *Web Front-End*. Essa divisão objetiva isolar as lógicas de processamento de requisições HTTP e de manipulação de dados privados. Assim, a lógica das aplicações é executada dentro de EPUs, em um componente denominado *Event Processing Engine*, que rastreia fluxos de informação com granularidade de rastreo em nível “de evento”. Esse componente, portanto, restringe a comunicação de EPUs, que geram, filtram e processam eventos, e podem se comunicar somente através da troca de eventos manipulada por um componente denominado *Event Broker*, que verifica os rótulos dos eventos e das EPUs ao direcioná-los. Algumas EPUs são privilegiadas e podem se comunicar através de operações de I/O comuns, possuindo, portanto, o privilégio de desclassificação sobre a classe de dados que manipulam. As EPUs escrevem os resultados de seu processamento em uma base-de-dados denominada *Application Database*, à qual as unidades de *Web Front-End* têm acesso do tipo somente-leitura (note-se que essa restrição cria um fluxo de informação unidirecional, de EPU's para unidades de web front-end). Outra base-de-dados, denominada *Web Database*, é utilizada para armazenar os dados processados pelas unidades de *Web Front-End*. Ambas as bases armazenam dados rotulados, e o mecanismo IFC, atuando com granularidade de rastreo em nível de variável nos módulos de *Web-Front-End*, garante que dados não pertencentes ao usuário de uma sessão sejam a ele enviados numa resposta HTTP. O *Web Front-End* age, portanto, como o *front-end* da arquitetura conceitual desta seção, interceptando as requisições e associando a elas autoridades em função de uma autenticação prévia (sem

analisar o código client-side enviado como resposta, porém), e também como *session-manager*, ao fornecer acesso a dados de sessão armazenados no *Web Database*.

Como outros trabalhos com abordagem similar – isto é, RS-IFC com poucas ou nenhuma modificação à linguagem, objetivando alta especificidade –, pode-se listar DEFCon (MIGLIAVACCA, 2010) e DIFCA-J (YOSHIHAMA, 2007), para Java, e Resin (YIP, 2009b), para PHP.

3.2.2.3 SIF/Swift

SIF é um framework para a construção de aplicações Web, escrito na linguagem de programação com tipos de segurança JIF 3.0 (MYERS, 1999), que implementa políticas de modelo centralizado, abrangendo confidencialidade e integridade. SIF é baseado no framework Java JEE (ORACLE INC., 2015), e os programas escritos em SIF podem ser executados em máquinas virtuais Java (ORACLE INC., 2013) comuns, bem como aplicações construídas com SIF podem ser executados em *containers* JEE também comuns.

No SIF, o sistema de tipos do JIF 3.0 é usado para assegurar que, quando um *servlet* (ORACLE INC., 2015) responde a uma requisição, apenas dados que podem ser lidos por esse cliente são enviados na resposta. De modo semelhante, dados originários de clientes recebem rótulos de integridade apropriados, de forma a detectar o fluxo desses comandos que requerem dados íntegros. Fluxos de informação originários de (ou destinados a) uma fonte externa – como arquivos, sistemas de bases de dados, ou sistemas isolados – podem ser rastreados desde que se escrevam interfaces em JIF 3.0 para controlar tais comunicações.

Fluxos de informação ocorrendo entre diferentes requisições são rastreadas no SIF limitando-se a generalidade de respostas que podem ser produzi-

das pela aplicação. No SIF, as requisições HTTP são recebidas por objetos *Servlet*, que delegam o tratamento à rotina *invoke* de objetos *Action*, que são mapeados como endpoints da aplicação. Um *Action*, em vez de enviar as respostas HTTP diretamente à conexão com o cliente, deve especificá-las como uma árvore de objetos da linguagem JIF, cada um representando uma tag HTML. Desse modo, a montagem dessas árvores é sujeita à verificação de tipos da linguagem JIF, e os tipos especificados no framework mapeiam os fluxos de informação que ocorrem entre cliente e servidor. Assim, elementos que podem dar origem a novas requisições (como os equivalentes às tags “form” e “a”) são mapeados a objetos *Action*, de forma que os parâmetros envolvidos nessas requisições (como os equivalentes às tags “input” ou parâmetros de consulta inseridos nas URLs) sejam mapeados aos argumentos da rotina de tratamento do *Action* correspondente. Como resultado, a verificação de tipos não só pode averiguar a adequação entre os parâmetros das requisições e os argumentos – evitando a submissão de dados com nível de classificação inferior –, como também perpetua o contexto da execução L_{pc} (do momento da inserção do elemento na árvore) entre diferentes requisições, evitando que haja fluxos implícitos entre elas.

Note-se que esse método para verificação de fluxos entre requisições é, porém, subjugado facilmente através de código client-side malicioso. Com o intuito de mitigar essa limitação, o SIF restringe a inserção de Javascript e CSS em páginas apenas como elementos estáticos, de forma que a adequação dos mesmos possa ser pré-verificada pelo programador.

Comparando-se a arquitetura do SIF à arquitetura da Figura 8, constata-se que o *front-end* e o *session-manager* são implementados como extensão do framework JEE, enquanto que os *internal* e *external gateways* não são previamente especificados, sendo responsabilidade dos programadores das

aplicações desenvolvê-los. Os *declassifier/endorser* são providos pela própria linguagem JIF, e os módulos das aplicações, confinados naturalmente pela linguagem JIF, são acessíveis somente pelo método *invoke* dos *Actions*. A análise do código client-side é parcialmente garantida pelo uso do sistema de tipos e pela inspeção manual.

Como trabalhos similares ao SIF, pode-se citar Swift (CHONG, 2007) e UR/Web (CHLIPALA, 2015). Swift adiciona Ajax (W3C, 2006) ao SIF, tornando-o capaz de produzir interfaces mais responsivas. Além disso, o compilador Swift apresenta um modelo de programação unificado entre cliente e servidor – isto é, programa-se para os dois num modelo de execução local –, distribuindo comandos entre ambos de forma a restringir a disposição de operações críticas ao servidor. A essa distribuição, aplica-se um algoritmo de otimização, de forma a minimizar o número de mensagens trocadas entre cliente e servidor. Por fim, o código-fonte inicial é compilado para HTML, CSS Javascript e Java, utilizando o framework Google GWT (GOOGLE INC., 2006). UR/Web, como Swift, apresenta um modelo de programação unificado entre cliente e servidor, baseando-se porém em uma linguagem de programação específica equipada com IFC. UR/Web, ao contrário do Swift e SIF, já foi utilizada para desenvolvimento de aplicações reais. UR/Web e Swift são aplicáveis a aplicações benignas, já que não fazem rastreo de canais encobertos como os de temporização e terminação.

3.2.2.4 Hails

Hails (GIFFIN, 2012) é um framework concebido para a construção de aplicações Web extensíveis, isto é, composta por módulos de diferentes desenvolvedores, mas que não necessariamente confiam uns nos outros. O modelo de política exercido no framework é o de rótulos de categorias disjuntas, apre-

sentado na seção 2.3.4.

Hails segue uma arquitetura derivada do padrão MVC (SCHMIDT, 2013), denominada MP-VC, em que os módulos das aplicações podem ser de dois tipos: MP (de *model-policy*) e VC (de *view-controller*). MPs são responsáveis pela persistência e recuperação de dados, e também por rotular tais dados de acordo com uma política de fluxo de informação. VCs são responsáveis por tratar as requisições HTTP dos clientes, e ao fazê-lo, podem se comunicar com um ou mais MPs ou com sistemas externos. Cada VC é um processo distinto, encapsulando um servidor HTTP que intercepta as requisições e distribui seu processamento a diferentes threads, que executam o código da aplicação. Para cada thread, é mantido um rótulo de segurança que é atualizado assim que a thread lê dados dos MPs. O mecanismo IFC garante que a política de controle de fluxo é respeitada quando a aplicação se comunica com clientes ou sistemas externos, e que os módulos tem a autoridade necessária para desclassificar dados.

O mecanismo IFC utilizado por Hails é dinâmico e implementado como uma biblioteca da linguagem de programação Haskell (STEFAN, 2011). A granularidade de rastreamento dos mecanismos é “nível de thread” e, portanto, o framework exige a mesma modularização de aplicações exigidas por mecanismos OS-IFC. Em adição ao mecanismo IFC, Hails emprega técnicas de isolamento de processos para confinar processos criados pelos VCs durante o tratamento de requisições. No client-side, Hails requer a instalação de uma extensão para interceptar requisições e verificar se os seus destinatários têm autoridade de leitura dos dados presentes na página.

Ao se comparar a arquitetura de Hails com a arquitetura da Figura 8, constata-se que as funcionalidades dos componentes de *front-end*, *session manager* e *external gateway* são executados pelo framework dentro dos módulos

	Técnica de Implementação	Garantias de Segurança	Proteção do Client-side	Programabilidade
Asbestos	OS-IFC	Poucas	Nenhuma	Média
SafeWeb	RS-IFC	Poucas	Nenhuma	Alta
SIF	L-IFC	Médias	Baixa	Baixa
Swift	L-IFC	Médias	Média	Baixa
Hails	RS-IFC	Fortes	Alta	Média

Tabela 3: Comparação dos trabalhos selecionados sobre IFC em servidores de aplicação Web

VC. A inspeção de código client-side, porém, não é executada, sendo uma extensão de browser responsável por garantir parcialmente a adequação de requisições efetuadas pelo código client-side. As funcionalidades do *Internal gateway* são executadas internamente a MPs, já que esses são responsáveis pela persistência e recuperação de dados.

3.2.2.5 Análise Comparativa

A tabela 3 resume as características dos trabalhos apresentados nesta seção. Em relação ao quesito de garantias de segurança, classifica-se com “poucas” os trabalhos que são sujeitos a fluxos implícitos (Asbestos e SafeWeb), com “médias” os que fornecem garantias de não-interferência insensível à terminação e temporização (SIF e Swift), e com “altas”, o Hails, por proteger canais de temporização e terminação. Em relação à proteção do client-side (que indica se as garantias de segurança no servidor podem ser subvertidas no cliente), atribui-se classificação “nenhuma” a Asbestos e SafeWeb, “pouca” a Hails (por requerer a instalação de uma extensão de browser para fazê-lo), “média” a SIF (por delegar a proteção de CSS e Javascript à inspeção manual) e “alta” a Swift (por apresentar um modelo de programação de mais alto nível, englobando cliente e servidor). Quanto à programabilidade, atribui-se caracterização “baixa” a SIF e Swift (por restringirem drasticamente a generalidade

das páginas das aplicações, e por serem de natureza L-IFC, que exigem uso intensivo de anotações de segurança), “média” a Asbestos (que, pela sua natureza OS-IFC, possibilita a utilização de sistemas de programação comuns, mas exige a compartimentalização das aplicações) e Hails (que também exige a compartimentalização das aplicações que controla) e “alta” a SafeWeb (devido à utilização de granularidade de rastreio em nível de variável, que reduz a necessidade de modularização das aplicações, além da ausência de alterações à sintaxe da linguagem de programação utilizada).

Além desses critérios, pode-se analisar os trabalhos sob ainda mais dois importantes aspectos: suporte a aplicações legadas (que se relaciona aos impactos de programabilidade) e as sobrecargas de desempenho. Em relação à primeira dimensão, dentre os trabalhos apresentados, somente os que se baseiam em mecanismos OS-IFC (ou RS-IFC, como Yoshihama (2007)) são capazes de proteger código de aplicações legadas (talvez ainda necessitando de modificações). Outros trabalhos necessitam da reescrita completa das aplicações. Em relação às sobrecargas de desempenho, mostrou-se difícil a comparação dos trabalhos sem um *benchmark* comum. Por esse motivo, omitiu-se esse tipo de discussão de forma mais detalhada. Entretanto, verifica-se, dentre os trabalhos que investigam os impactos de desempenho de suas próprias propostas (EFSTATHOPOULOS, 2005; HOSEK, 2011; GIFFIN, 2012)), que os sobrecustos de execução impostos pelas mesmas são similares aos de tecnologias e frameworks Web populares.

4 PROPOSTA DE ARQUITETURA

Considera-se neste trabalho um cenário que envolve os seguintes participantes: uma *instituição* pública ou privada, as *pessoas associadas* à mesma (por exemplo, clientes, funcionários, cidadãos) e *desenvolvedores de software terceiros* (isto é, vinculados a outra instituição, ou independentes). De autoria desses desenvolvedores terceiros, aplicativos móveis são utilizados pelas pessoas associadas para, a partir de seus próprios dispositivos, manipular dados e ativos da instituição através de um conjunto de APIs fornecido pela mesma. Em adição, esses aplicativos podem ainda utilizar dados e serviços de outras instituições, bem como de componentes de hardware e software dos dispositivos que os executam.

A arquitetura descrita neste capítulo utiliza-se, naturalmente, de mecanismos IFC para proteger os dados institucionais manipulados pelos aplicativos de terceiros. Requer-se desses mecanismos o fornecimento de garantias de segurança robustas, uma vez que a ausência de vínculo entre desenvolvedores e instituição potencializa o risco da presença de código malicioso nos aplicativos. Assume-se, em adição, que tanto instituição quanto desenvolvedores terceiros não possuem privilégios especiais nos processos de desenvolvimento e distribuição de aplicativos das plataformas móveis – sendo incapazes de influenciar sistemas operacionais, middlewares e bibliotecas de runtime característicos das mesmas. Assume-se ainda o requisito de os aplicativos de terceiro serem desenvolvidos com linguagens, frameworks e biblio-

tecas de uso habitual dos desenvolvedores das plataformas móveis, tal que os mesmos não necessitem capacitar-se em novas tecnologias para implementar aplicativos para a instituição. Por fim, deseja-se a implementação de um mecanismo de fácil replicação em diferentes plataformas móveis.

As restrições descritas nesse cenário impõem dificuldades adicionais à implementação de mecanismos IFC. A utilização de mecanismos OS-IFC torna-se inviável pois, como mencionado, instituição e desenvolvedores não possuem influência sobre o ambiente de execução nativo das plataformas móveis. Já a utilização de mecanismos L-IFC mostra-se de alta complexidade frente à variedade de ambientes de desenvolvimento das plataforma móveis, não havendo maneira de replicá-los facilmente nas diferentes linguagens, frameworks e bibliotecas associadas às mesmas. A utilização de mecanismos RS-IFC sofre do mesmo revés, além de ser pouca promissora em relação aos impactos de desempenho produzidos.

Felizmente, as plataformas móveis possuem um componente de software em suas bibliotecas associadas com o qual se pode implementar mecanismos IFC atendendo-se às restrições apresentadas: o *WebView*. Tal componente é capaz de exibir páginas HTML completas – isto é, contendo Javascript, CSS, imagens, vídeos e áudio –, além de fornecer APIs para monitoramento e controle dos processos de carregamento e renderização de páginas, e para comunicação bidirecional entre o Javascript executado em seu interior e o código nativo das aplicações em que é inserido. Sendo universalmente utilizado para a implementação de aplicações híbridas (isto é, aplicações constituídas de código nativo e páginas HTML, com o objetivo de tornar o código presente nas páginas portátil entre as diferentes plataformas), o funcionamento do *WebView* é de conhecimento comum entre os desenvolvedores de aplicativos móveis. Além disso, o componente fornece um mecanismo natural de confina-

mento de código intra-aplicativo, provendo meios para se controlar o acesso, pelos scripts executados, ao sistema de arquivos, à rede e a outros recursos.

Assim sendo, o componente de WebView é utilizado nesta proposta para a construção de um mecanismo IFC no interior de um aplicativo nativo – denominado aplicativo-base – que age como middleware para a execução dos aplicativos de terceiros, construídos como aplicações Web. Muito embora a adoção desse modelo remeta aos trabalhos apresentados na seção 3.2.1, versando acerca de mecanismos IFC em Web-browsers, restringe-se aqui à utilização do componente de WebView inalterado, criando-se uma categoria de granularidade de rastreo inédita a esses trabalhos: em nível de “WebView”. Assim, pode-se enunciar que o aplicativo-base é dotado de *mecanismos IFC dinâmicos com granularidade de rastreo em nível de WebView*.

Além do aplicativo-base, a arquitetura aqui descrita especifica um servidor de aplicações Web que dá acesso aos sistemas institucionais. Executado no ambiente computacional da instituição, o servidor é dotado também de mecanismos IFC – como os apresentados na seção 3.2.2 –, abrigando módulos *server-side* dos aplicativos de terceiro que dão suporte à suas execuções no aplicativo-base.

Nas seções seguintes, descreve-se a política de segurança executada pelos mecanismos (seção 4.1), a arquitetura do servidor (seção 4.2) e do aplicativo-base (seção 4.3). Na seção 4.4, descreve-se a implementação do aplicativo-base como um aplicativo nativo do sistema operacional Android.

4.1 Política de Segurança

A política de segurança exercida globalmente pelos mecanismos no aplicativo-base e no servidor reflete os interesses dos diferentes participantes envolvidos na execução dos aplicativos. Assume-se, assim, que os desenvolvedores terceiros desejam manter seus dados isolados de aplicativos de outros desenvolvedores, externos ou internos ao ambiente de execução; que as pessoas desejam não divulgar alguns de seus dados aos desenvolvedores terceiros; e que a própria instituição deseje também não divulgar alguns de seus dados, associados a pessoas ou não, aos desenvolvedores terceiros.

Essas restrições impõe um caráter federativo à política, o que a diferenciando dos modelos apresentados na seção 2.3, cujos elementos constituem um reticulado. Ela pode ser descrita com base nos conjuntos *APS* aplicativos (com n elementos), *USRS* de usuários (com m elementos provenientes do conjunto de pessoas associadas à instituição) e *NS* de níveis de confidencialidade, especificados respectivamente da seguinte maneira:

$$APS = \{app_1, app_2, \dots, app_n\}, USRS = \{u_1, u_2, \dots, u_m\}, NS = \{l, h\} \quad (4.1)$$

Os elementos l e h de *NS* especificam níveis de confidencialidade pública e privada, respectivamente. O conjunto *SC* de classes de segurança, a relação $\sqsubseteq$ (pode-fluir-para) e a operação $\sqcup$ (combinação de classes) podem ser então especificadas como segue:

$$SC = \{global, app_i, (app_i, u_j, ns) | 1 \leq i \leq n \wedge 1 \leq j \leq m \wedge ns \in NS\} \quad (4.2)$$

$$\begin{aligned} \sqsubseteq = & \left\{ \left(global, global \right), \left(global, app_i \right), \left(global, (app_i, u_j, ns) \right) \right. \\ & \left(app_i, app_i \right), \left(app_i, (app_i, u_j, ns) \right), \left((app_i, u_j, l), (app_i, u_j, l) \right), \\ & \left((app_i, u_j, l), (app_i, u_j, h) \right), \left((app_i, u_j, h), (app_i, u_j, h) \right) \\ & \left. \left| 1 \leq i \leq n \wedge 1 \leq j \leq m \wedge ns \in NS \right\} \end{aligned} \quad (4.3)$$

$$SC_1 \sqcup SC_2 = \begin{cases} \text{se } SC_1 \sqsubseteq SC_2, & \text{então } SC_2 \\ \text{se } SC_2 \sqsubseteq SC_1, & \text{então } SC_1 \\ \text{indefinida, caso contrário} \end{cases} \quad (4.4)$$

Observe-se que, uma vez que a operação $\sqcup$ não se define para qualquer par de classes de segurança, a política não pode formar um reticulado. O diagrama de Hasse da figura 7 ilustra a estrutura formada pela relação “ $\sqsubseteq$ ” nos elementos de SC .

Figura 7: Diagrama de Hasse da relação “pode-fluir-para” ($\sqsubseteq$)

Referenciando-se a figura 7, a semântica das diferentes classes de informação pode ser descrita da seguinte maneira. A classe *global* reúne informações que podem ser lidas por qualquer aplicativo, sob a autoridade de qualquer usuário e podem se referir a informações de configuração global do sistema, por exemplo. As classes do tipo app_i denotam informações específi-

cas do aplicativo app_i , podendo ser lidas somente por componentes do próprio e podem representar, por exemplo, informações de configurações ou bases de dados globais do aplicativo. As classes do tipo (app_i, u_j, l) e (app_i, u_j, h) reúnem informações de usuários específicos, e são, respectivamente, confiadas e não confiadas pelo usuário u_j para divulgação externa por app_i .

4.2 Arranjo no Servidor

Mantido sob controle da instituição, o servidor tem o propósito de estender e auxiliar o funcionamento do aplicativo-base e de seu mecanismo IFC, bem como de sustentar a execução dos próprios aplicativos de terceiros, abrigando componentes *server-side* responsáveis por atender a requisições efetuadas pelas páginas desses aplicativos. Através dos componentes server-side, os aplicativos têm acesso, ainda, aos dados provenientes dos sistemas internos da instituição.

Com o objetivo de exercer as políticas de confidencialidade e integridade institucionais, a arquitetura do servidor fundamenta-se na utilização de mecanismos IFC com granularidade de rastreo em nível de processo ou thread (tal característica se deve à granularidade de rastreo do aplicativo-base, que, por ser grosseira, invalida a utilização de mecanismos com granularidade mais fina no servidor). A figura 8 ilustra a disposição dos componentes na mesma, que é similar à arquitetura apresentada na seção 3.2.2. Observa-se que tais componentes dividem-se em dois tipos: módulos auxiliares e módulos de aplicativos.

Os módulos auxiliares têm por objetivo assistir a execução do aplicativo-base e, como na arquitetura apresentada na seção 3.2.2, comandar a atuação do mecanismo IFC, estabelecendo as classes de segurança e rotulando com elas outros módulos e dados que adentram os limites de atuação do mecanismo. Todos têm acesso a uma base de dados administrativa, que armazena dados relacionados aos usuários e aos aplicativos executados. As funcionalidades desses módulos podem ser descritas, especificamente, como adiante:

- Serviços auxiliares: fornecem serviços de autenticação de usuários, modificação de configurações de privacidade e de acesso a rotinas críticas

Figura 8: Arquitetura do Web Server da loja de aplicativos

dos sistemas internos, sendo de uso exclusivo do aplicativo-base. O serviço de autenticação de usuários associa um usuário dos sistemas internos a cada uma das instalações do aplicativo-base, relacionando às mesmas números secretos – denominados *install_token* – que as identificam em futuras requisições (de forma similar a cookies de sessão de Web-browsers). O serviço de modificação de configurações de privacidade é utilizado para que os usuários especifiquem quais de seus dados podem ser fornecidos a módulos do aplicativo i que têm acesso à internet (especificamente, com rótulos app_i e (app_i, u_j, l) , como se detalhará adiante). Ambos os serviços armazenam os resultados de suas operações na base de dados administrativa, de forma que esses possam ser utilizadas futuramente pelo próprio módulo de serviços auxiliares, ou por

outros módulos auxiliares. O serviços de acesso a rotinas críticas fornece ao aplicativo-base a possibilidade de utilização de rotinas dos sistemas internos aos quais os aplicativos não têm direito de acesso, por não se confiar que esses respeitem a integridade dos dados fornecidos como entrada às mesmas.

- Front-end: responsável por interceptar as requisições HTTP e, em função do *install_token* e do path presente em cada uma (como detalhado adiante), entregá-las aos módulos de aplicação com classe de segurança compatível (o *install_token* é removido do header da requisição, dada sua natureza secreta). Esses módulos podem não ter sido previamente instanciados, sendo responsabilidade do front-end fazê-lo ao receber uma requisição destinada aos mesmos. Para realizar tais funções, o front-end utiliza-se dos dados armazenados em bases de dados administrativas, bem como do código-fonte das aplicações, que é também mantido nessas bases de dados.
- External gateway: fornece acesso à rede para os módulos de aplicação cujos rótulos são da categoria app_i e (app_i, u_j, l) . O acesso a uma URL específica é autorizado somente se essa se encontra em uma lista de endereços autorizados para o aplicativo i , previamente definida pelo seu desenvolvedor e armazenada na base de dados administrativa.
- Internal gateway: fornece o acesso a dados de sistemas internos da instituição para módulos de aplicação. Em função das configurações de privacidade armazenadas nas bases de dados administrativas – sendo essas definidas pelo usuário ou pela própria instituição – e do rótulo de segurança do módulo acessante (que determina se o mesmo tem acesso à rede), regula o acesso a diferentes categorias de dados.

Os módulos de aplicativos executam, naturalmente, os componentes *server-side* dos aplicativos de terceiros, estando assim sujeitos às restrições do mecanismo IFC. Cada aplicativo é associado a três tipos de módulos, rotulados com classes de segurança da política da seção 4.1: app_i , (app_i, u_j, l) e (app_i, u_j, h) . O módulo auxiliar de *front-end* roteia as requisições para os diferentes módulos em função do *install_token* e do path contidos nas mesmas. O path de requisições para módulos de classe app_i possui prefixo “/app_i”, ao passo que requisições a módulos de classificação (app_i, u_j, l) e (app_i, u_j, h) possuem, respectivamente, prefixos “/app_i/l” e “/app_i/h”. A finalidade dos módulos de aplicação é especificada a seguir:

- Módulos de classe app_i : atendem a requisições de todos os usuários e possuem acesso à rede externa. Por esse motivo, os dados institucionais a que os mesmos têm acesso são considerados como de possível divulgação pública nas configurações de privacidade das bases administrativas.
- Módulos de classe (app_i, u_j, l) : atendem a requisições somente do usuário u_j , e possuem também acesso à rede externa. Esses módulos têm a finalidade de fornecer aos aplicativos de terceiros a proteção de seus próprios dados contra ataques de XSS ou SQL-injection, possivelmente conectando a implementação dos mecanismos IFC do servidor institucional a outras implementações nos servidores de terceiros. Entretanto, os dados institucionais a que os mesmos têm acesso são considerados também, igualmente ao caso anterior, como de possível divulgação pública nas configurações de privacidade das bases administrativas.
- Módulos de classe (app_i, u_j, h) : atendem a requisições somente do usuário u_j , e não possuem acesso à rede externa. Por se tratar dos módulos

de maior confinamento, possuem acesso irrestrito aos dados do usuário u_j fornecidos através das APIs institucionais do *internal gateway*.

Muito embora não seja permitido que os módulos rotulados com diferentes classes de segurança comuniquem-se bidirecionalmente, o compartilhamento de dados entre os mesmos pode se dar através do sistema de arquivos, respeitando-se a relação $\sqsubseteq$ da política da seção 4.1. Assim, como se vê na figura 8, módulos de classe (app_i, u_j, h) têm acesso de leitura a arquivos de classe (app_i, u_j, l) , app_i e $globals$; módulos de classe (app_i, u_j, l) têm acesso de leitura a arquivos de classe app_i e $globals$; e módulos de classe app_i têm acesso de leitura a arquivos de classe $globals$.

4.3 Arranjo no Aplicativo-base

O aplicativo-base, como mencionado, utiliza-se do componente de Web-View para executar de modo confinado os aplicativos de terceiros, exercendo sobre os mesmos a política de fluxo de informação cumprida no servidor. Utilizando-se dos serviços auxiliares apresentados na seção 4.2, o aplicativo-base executa também funcionalidades administrativas de autenticação de usuário e configuração de privacidade de aplicativos, além de fornecer e regular aos mesmos o acesso a rotinas críticas (de forma indireta, porém, submetendo os dados submetidos como entrada à inspeção do usuário). O aplicativo-base ainda fornece aos aplicativos dados provenientes do dispositivo (como informações de hardware ou software e leituras de sensores), também sob a discricção das configurações de privacidade especificadas para os mesmos.

Assume-se que cada um dos dispositivos em que o aplicativo-base é instalado associa-se a somente um usuário. Essa hipótese, que é assumida também por inúmeros aplicativos de larga utilização na atualidade, fundamenta-se no modo de utilização predominantemente pessoal dos dispositivos móveis. Essa hipótese permite a simplificação do mecanismo IFC do aplicativo-base, pois esse passa a exercer uma política de segurança que é, de certa forma, uma simplificação da política de confidencialidade descrita na seção 4.1, estruturada somente de partições pertencentes a um único usuário. Tal política pode ser definida, então, como a seguir.

$$SC = \{app_i, (app_i, ns) | 1 \leq i \leq n \wedge ns \in NS\} \quad (4.5)$$

$$\sqsubseteq = \left\{ \left((app_i, app_i), (app_i, (app_i, ns)), ((app_i, l), (app_i, ns)), ((app_i, h), (app_i, h)) \right) \right. \\ \left. \left| 1 \leq i \leq n \wedge ns \in NS \right. \right\} \quad (4.6)$$

$$sc_1 \sqcup sc_2 = \begin{cases} \text{se } sc_1 \sqsubseteq sc_2, & \text{então } sc_2 \\ \text{se } sc_2 \sqsubseteq sc_1, & \text{então } sc_1 \\ \text{indefinida, caso contrário} \end{cases} \quad (4.7)$$

4.3.1 O componente de WebView

Nas plataformas móveis, o componente de WebView pertence ao grupo de componentes de interface gráfica nos frameworks de desenvolvimento das mesmas. Disponível no conjunto de bibliotecas de runtime dos dispositivos, o componente encapsula, como se detalhará, o funcionamento de um motor de renderização completo de Web-browsers, bem como de seus módulos correlacionados, provenientes das implementações originais de Web-browsers tradicionais.

Sendo um componente de interface gráfica, o WebView se associa a uma área retangular da tela do dispositivo, em que seu conteúdo é exibido (páginas HTML, em específico) e os eventos de interação são capturados e enviados ao mesmo (que os interpreta como eventos de interação HTML). Além disso, fornece um conjunto de rotinas com os quais se pode examinar e controlar o processo de carregamento e renderização de páginas, bem como interagir com o conteúdo carregado, permitindo que o código controlador do conjunto de *views* em que o componente se insere dê a esse conjunto um comportamento específico.

A figura 9 ilustra uma arquitetura conceitual dos componentes de WebView, baseada na arquitetura genérica de Web-browsers concebida por Groskurth e Godfrey (2005). Observa-se na figura a presença dos seguintes componentes:

- **WebView Wrapper:** implementação em mais alto nível, aderindo à especificação de componentes de interface nas APIs de plataformas móveis. É responsável tanto por encapsular e coordenar a ação de seus módulos internos, quanto expor o funcionamento dos mesmos através de interfaces de alto nível, definidas num submódulo de monitorização e controle.
- **Motor de renderização:** principal módulo do componente, responsável por interpretar código HTML e CSS, transformá-lo em uma árvore DOM e, posteriormente, numa árvore de desenho, com a qual se constrói a interface gráfica da página exibida (GARSIEL; IRISH, 2011).
- **Browser Engine:** fornece uma interface em mais alto nível para o motor de renderização, utilizada internamente pela camada de Wrapper.
- **Networking:** responsável por funcionalidades de acesso à rede, executando requisições HTTP a comando do motor de renderização e da própria WebView.
- **Interpretador Javascript e XML parser:** utilizado pelo motor de renderização para manejar, naturalmente, Javascript e CSS. No caso do interpretador Javascript, a WebView expõe uma interface externa para que o código controlador da View possa interagir com o conteúdo da página, permitindo tanto que esse código execute Javascript diretamente, quanto que o próprio Javascript da página possa disparar a execução de código desse controlador.

- Display-backend: utilizado pela implementação da WebView e pelo motor de renderização para desenhar a interface gráfica do componente.
- Persistência: responsável por armazenar dados de diferentes camadas do componente, incluindo-se cookies, histórico, pilha de navegação, preferências, certificados, conteúdo do Web-storage e o cache de arquivos.

Figura 9: Arquitetura conceitual do WebView

Com a interface exposta pelo componente, o código controlador da WebView pode, dentre outras funções, comandar o carregamento de uma página, e posteriormente consultar o progresso dessa tarefa; interceptar requisições HTTP, podendo bloqueá-las e sobrescrever o conteúdo das respostas; manipular o histórico, cache, certificados e preferências armazenados pelo componente; alterar seu comportamento, por exemplo, para desabilitar a execução de áudio e vídeo ou interpretação de Javascript; e, como já mencionado, interagir com o código Javascript da página carregada.

4.3.2 IFC dinâmico em WebViews

A maleabilidade fornecida pelo componente de WebView sobre seus módulos internos, conforme descrito na seção anterior, possibilita o exercício de políticas de controle de acesso adicionais àquelas intrinsecamente implementadas no componente, relacionadas ao confinamento do modelo de execução

Web e à política de mesma origem. Em particular, tal maleabilidade é aqui utilizada para limitar o acesso a recursos disponíveis às páginas de aplicativos, como a execução de requisições HTTP ou a leitura de dados provenientes do dispositivos, a partir da política de controle de fluxo descrita na seção 4.3 e das configurações de privacidade do usuário.

Muito embora os trabalhos referenciados na seção 3.2.1, propondo a aplicação de mecanismos IFC em Web-browsers, atuem com rastreo em nível de frame ou variável, o componente de WebView original – ao qual a utilização se restringe nesta proposta – não permite a implementação de mecanismos de mesma granularidade. Os listeners e delegates descritos na seção anterior associam-se a instâncias de WebView, isto é, objetos mantidos no *heap* dos processos dos aplicativos, com contextos de execução relativamente isolados, sendo análogos às tradicionais abas dos Web-browsers. Como esses listeners e delegates não identificam frames solicitantes de um recurso, tampouco se prestam à monitoração e controle da comunicação entre frames, deve-se considerar como sujeitos, no mínimo, as próprias instâncias de WebView.

Em função dessas limitações, o mecanismo IFC aqui proposto associa rótulos de segurança a instâncias de WebView em função dos paths das requisições feitas ao servidor pelas mesmas. Assim, ao fazer requisições a um módulo com path “/app_*i*” do aplicativo *i*, uma instância é associada à classe de segurança *app_i*. A partir de então, como representado na tabela 4, torna-se capaz de fazer requisições a somente esse próprio módulo, a módulos com path “/app_*k*”¹, sendo *k* outro aplicativo, a sistemas externos – desde que, em ambos os casos, esses destinos estejam numa lista determinada para essa classe de segurança –, ou a módulos com path “/app_*i*/” e “/app_*i*/h”, ocasiões em que o rótulo associado à instância é alterado para

¹A requisição nesse caso recebe um *header* extra para indicar origem de uma página de *i*, que pode ser usada por *k* para decidir acerca do compartilhamento de dados com *i*

Nível de segurança	Módulos de aplicativos		Sistemas externos	Dados do dispositivo
	Sem mudança de rótulo	Com mudança de rótulo		
app_i	$\begin{matrix} /app_i \\ /app_k \\ \text{onde } 1 \leq k \leq n \end{matrix}$	$\begin{matrix} /app_i/l \\ /app_i/h \end{matrix}$	URLs de um conjunto previamente definido pelo desenvolvedor	De um conjunto previamente autorizado pelo usuário
(app_i, u_j, l)	$/app_i/l$	$/app_i/h$	URLs de um subconjunto do acima definido	Do mesmo conjunto acima definido
(app_i, u_j, h)	$/app_i/h$	–	–	Irrestrito

Tabela 4: Matriz de controle de acesso da WebView, definida em função da classe de segurança associada à mesma

(app_{a_1}, u_j, l) e (app_{a_1}, u_j, h) , respectivamente. Associada à classe de segurança (app_i, u_j, l) , a instância tem o direito de fazer requisições aos módulos do aplicativo com path “/app__i/l” ou “/app__i/h” (nesse último caso, porém, a classe de segurança da instância é alterada para (app_i, u_j, h)), ou a sistemas externos previamente listados para essa classe de segurança. Com classe de segurança (app_i, u_j, h) , a instância pode realizar requisições somente para o módulo com path “/app__i/h”. Quanto a dados provenientes do dispositivo, concede-se a leitura dos mesmos a instâncias com classe de segurança app_i ou (app_i, u_j, l) somente se o usuário, nas suas configurações de privacidade, classificar tais dados como de acesso potencialmente público. Instâncias com classe (app_i, u_j, h) , por outro lado, têm acesso irrestrito a esses mesmos dados.

Note-se que as restrições descritas são insuficientes, porém, para garantir o isolamento entre diferentes instâncias do componente de WebView, rotuladas com classes de segurança incompatíveis. Uma vez que os módulos de todos os aplicativos são associados a uma mesma origem – característica

que facilita o compartilhamento de dados de diferentes classes de segurança em páginas de um mesmo aplicativo –, possibilita-se o compartilhamento de dados através de mecanismos legítimos como Web-storage e cookies, associados a essa mesma origem e acessíveis por qualquer instância de WebView, independentemente de sua classe de segurança. Além disso, canais de comunicação encobertos existentes no browser, utilizando o cache e o histórico de navegação (JACKSON, 2006), por exemplo, ameaçam a atuação do mecanismo IFC. Tais dificuldades impõem, assim, restrições adicionais para a utilização do WebView como elemento de isolamento: a utilização de uma única instância por instante de tempo e a limpeza do componente ao se encerrar a execução de um aplicativo. Essa limpeza, por sua vez, aproxima a execução de um aplicativo no aplicativo-base à execução de uma seção privada em um Web-browser, após a qual, segundo Aggarwal (2010), deve-se eliminar algumas categorias de dados que compõem um estado visível do browser. No caso específico do aplicativo-base, a limpeza abrange dados acessíveis via Javascript (cookies, Web-storage), cache, o histórico de páginas visitadas e estados de configuração (certificados, base de dados de senhas e inputs de formulários).

4.3.3 Label Creep, layer apartado e seccionamento de aplicativos

A granularidade de rastreo grosseira do mecanismo IFC no aplicativo-base ocasiona um fenômeno de label creep análogo ao apresentado na seção 2.1.2.1, manifestando-se em função de uma instância de WebView poder acumular rótulos de segurança somente mais restritivos, que limitam progressivamente suas capacidades de comunicação. Essa deficiência acarreta em prejuízos de programabilidade ao mecanismo, pois obriga aplicativos a fazer o pré-carregamento de dados a serem utilizados quando estiverem com ca-

pacidade de comunicação limitada – e, para algumas funcionalidades, esse pré-carregamento pode ser inviável, ou mesmo impossível.

Tal efeito negativo é mitigado nesta proposta através de um artifício que amplia as capacidades de comunicação de aplicativos confinados, permitindo a eles acessar módulos do servidor de classes de segurança menos restritivas – sem que a comunicação dependa dos dados manipulados na WebView – ou reiniciar o estado de navegação do aplicativo num módulo de classificação menos restritiva, depois de realizada a limpeza do componente descrita na seção anterior.

O artifício utilizado para tal finalidade baseia-se na sobreposição de um *layer apartado* na interface gráfica do aplicativo, acima da área de desenho da WebView, como ilustrado na figura 10. O conteúdo do layer é especificado pelo aplicativo correntemente executado através da interface entre código Javascript e o aplicativo-base, por meio de uma função denominada **populate**. Essa função, cuja execução se restringe a WebViews com classe de segurança app_i ou (app_i, u_j, l) , recebe como entradas: elementos gráficos a serem exibidos no layer; o posicionamento dos mesmos elementos na tela; e a ação executada quando um desses elementos recebe um evento de toque do usuário. Quando um evento de toque ocorre num desses elementos, o aplicativo-base, que controla diretamente os eventos de interação dos usuários com o layer apartado, executa a ação especificada, que pode ser uma requisição a um dos módulos server-side de classe de segurança inferior à associada à WebView, ou o reinício do estado de navegação do aplicativo com o conteúdo de algum desses módulos. No primeiro caso, a requisição é feita diretamente pelo aplicativo-base, que notifica o aplicativo quando do término da mesma. No segundo caso, interrompe-se a execução corrente do aplicativo e executa-se a limpeza do componente de WebView, seguindo-se do reinício da execu-

ção do aplicativo a partir do conteúdo fornecido por algum de seus módulos, como especificado na chamada anterior à função **populate**.

Figura 10: Disposição da interface, mostrando a área reservada à WebView, sobre a qual o layer apartado é desenhado. No exemplo, 3 botões são exibidos no layer.

4.3.4 Layer de sanitização

A sanitização de entradas de rotinas críticas, executadas pelos serviços auxiliares do servidor (aos quais os aplicativos de terceiro não possuem acesso), se dá de modo análogo ao funcionamento do layer apartado descrito na seção anterior. Nesse caso, o aplicativo se comunica com o aplicativo-base através de uma função Javascript específica, associada à rotina crítica a ser executada, informando os dados a serem fornecidos como entrada à rotina. Esses dados são então exibidos ao usuário em um outro layer, de controle exclusivo do aplicativo-base, em que o usuário autoriza a execução da rotina se concluir que os dados de entrada são válidos. A figura 11 ilustra a disposição na interface do aplicativo-base.

O reinício do estado de navegação descrito sugere que os aplicativos de terceiros possuam uma organização interna seccionada, em que uma seção de páginas se agrupa em torno de uma página que faz requisições somente a módulos de classe de segurança app_i , que realizam o carregamento de todos os dados a serem utilizados pela seção ao processar dados de nível de segurança h . Tal estruturação, em comparação a uma estrutura monolítica de aplicativos – isto é, uma em que há uma única seção, que carrega todos os dados externos na sua inicialização – tem as vantagens não só na redução na quantidade de dados trafegados pelo aplicativo (e, possivelmente, não utilizados) e no tempo execução do mesmo, como também habilita aos aplicativos obter dados de origem mais recente. Um possível critério para o seccionamento de aplicativos pode ser definido de acordo com as funcionalidades executadas pelos mesmos – isto é, diferentes funcionalidades são agrupadas em diferentes seções – e também de acordo com as subdivisões dos dados em que essas funcionalidades operam.

Figura 11: Disposição da interface, mostrando a área reservada à WebView, o layer apartado e o layer de sanitização, exibindo uma mensagem de confirmação

4.4 Implementação

No intuito de comprovar a viabilidade da arquitetura descrita, implementou-se uma prova de conceito abrangendo os componentes essenciais da mesma, isto é, aqueles relacionados ao confinamento de aplicativos de terceiro no aplicativo-base. A implementação no servidor, apesar de fundamental para o funcionamento do aplicativo-base, já foi amplamente estudada nos trabalhos apresentados na seção 3.2.2, versando sobre mecanismos IFC em servidores de aplicações Web. Assim, omitiu-se aqui a discussão acerca de seu funcionamento, tendo sido utilizado um servidor de aplicações Web comum para dar suporte à execução do aplicativo base e dos aplicativos de terceiro. Tal servidor executa os mesmos módulos descritos na seção 4.2 – possuindo a mesma interface que possuiria uma implementação em servidor com IFC –, mas sem o isolamento característico entre os mesmos.

4.4.1 Android

O aplicativo-base foi implementado como um aplicativo nativo da plataforma Android, cuja natureza de código aberto permitiu a inspeção interna do componente de WebView nela embarcado. O Android abrange um sistema operacional baseado em Linux, um sistema de middleware, bibliotecas de runtime, um framework para aplicações de usuário e um subconjunto padrão de aplicações de usuário. Tais elementos se organizam numa arquitetura em camadas (HU, 2012), conforme ilustra a figura 12. Desenvolvida pela associação de empresas Open Handset Alliance, a plataforma permite que fabricantes modifiquem elementos das camadas inferiores de forma a adaptá-la ao funcionamento em seus dispositivos móveis, enquanto que o middleware, as bibliotecas de runtime e o framework - mantidos inalterados - garantem a

compatibilidade desses dispositivos com o conjunto de aplicativos de usuário escritos para a plataforma por terceiros e comercializados numa loja de aplicativos única da mesma.

Aplicações <i>Home, Calendário, Discador, Câmera, Alarme, Calculadora, Relógio...</i>	
Framework <i>ActivityManager, WindowManager, ContentProvider, ViewSystem, NotificationManager, ResourceManager, LocationManager...</i>	
Bibliotecas <i>SurfaceManager, MediaFramework, SQLite, WebKit, OpenGL/ES, AudioManager, SSL...</i>	Runtime Android <i>Core Libraries</i> <i>Máq. Virtual: Dalvik/ART</i>
Camada de Abstração de Hardware (HAL) <i>Gráficos, Áudio, Câmera, Bluetooth, GPS, Rádio, Wifi...</i>	
Kernel Linux <i>Drivers (display, câmera, bluetooth, memória compartilhada, Binder, Wifi, USB), Gerenciador de energia...</i>	

Figura 12: Arquitetura do Android

Os aplicativos desenvolvidos para a plataforma Android são executados pelo middleware, que é uma implementação de máquina virtual Java. Essa camada semi-interpretativa adiciona portabilidade de código à plataforma, pois possibilita que o código de aplicativos seja independente das arquiteturas dos diferentes processadores utilizados nos dispositivos. A linguagem com que aplicativos Android são desenvolvidos é, então, primordialmente Java, o que torna a plataforma compatível com as bibliotecas originais da linguagem. A implementação de mecanismos de JNI (ORACLE INC., 2014) possibilita também a execução de código nativo C dentro do middleware. De fato, inúmeras bibliotecas da camada de bibliotecas de runtime são total ou parcialmente escritas em C, linguagem em que componentes de aplicativos também podem ser escritos.

Em adição às restrições da camada semi-interpretativa do middleware, o

framework Android também impõe restrições à estrutura interna dos aplicativos. Sendo assim, esses devem possuir uma divisão interna descrita em termos de componentes de quatro tipos, a saber:

- *Activity*: componente de interface gráfica, responsável por interagir com o usuário através de mecanismos baseados em eventos. Especifica elementos gráficos a serem exibidos na tela e trata a interação do usuário com estes elementos, possivelmente realizando alterações nesses elementos gráficos.
- *Service*: componentes que executam tarefas de longa duração, e que, portanto, não podem estar associados a um ciclo de vida curto como os de Activities. Desse modo, componentes Service podem ser executados em background, sem que usuários estejam de fato interagindo com a aplicação.
- *Content Provider*: fornecem para outros componentes uma interface de leitura e escrita a uma fonte de dados, seja esta local (um arquivo) ou remota (algum serviço na rede).
- *Broadcast Receiver*: componente responsável por receber e tratar mensagens enviadas por outros componentes.

Todos esses componentes são executados tipicamente em um processo, de thread interpretativa única, associado ao aplicativo (essa configuração é, entretanto, modificável pelos desenvolvedores). Os processos de diferentes aplicativos recebem *uids* distintos do Unix, de modo que os mecanismos de isolamento inter-processos da camada de sistema operacional forneçam também isolamento entre recursos de diferentes aplicativos, como, por exemplo, diretórios do sistema de arquivos. A camada de sistema operacional também

é responsável por controlar diferentes permissões associadas aos aplicativos (por exemplo, acesso à rede) através do mapeamento dos processos a diferentes *gids* Unix. Tais permissões são declaradas num arquivo descritivo associado ao aplicativo denominado manifest, cujo conteúdo é parcialmente exibido ao usuário durante a instalação do aplicativo, momento em que se concedem as permissões ao aplicativo. Os componentes de diferentes aplicativos Android podem se comunicar através de um mecanismo de comunicação inter-processos de alto nível denominado Binder, que é sempre intermediado por um processo privilegiado denominado Reference Monitor. Esse processo verifica a permissão dos componentes envolvidos na comunicação e, estando essa de acordo como conteúdo do manifest de ambos, a mensagem é entregue. Uma descrição mais detalhada dos mecanismos de segurança do Android pode ser encontrada em Enck, Ongtang e McDaniel (2009).

4.4.2 Android WebView

Na plataforma Android, o componente de Webview se encontra implementado na camada de bibliotecas de runtime, e é acessado pelo código dos aplicativos através de wrappers na camada de framework. A partir da versão 4.4 do Android, o código do Webview faz uso de componentes do projeto Chromium, que foram adaptados em função das restrições impostas pelo middleware e framework da plataforma. Tais componentes, que incluem o motor de renderização Blink e o interpretador Javascript V8 e outros, são linkados ao processo de um aplicativo através dos mecanismos JNI do middleware.

Os wrappers da camada do framework Android para acesso aos componentes de Webview fornecem mecanismos para monitorar e controlar tarefas dos módulos apresentados na seção 4.3.1, proporcionando expressividade suficiente para possibilitar a implementação de Web browsers completos, in-

clusive. Além disso, é especificada uma API para a comunicação entre código nativo e o Javascript interpretado na WebView. Algumas das classes que implementam essas funcionalidades são as seguintes:

- *WebView*: classe que representa uma instância de WebView, e que desce de *View*, a classe pai de todos os componentes desenháveis na tela, utilizada por Activities na especificação de suas interfaces. A classe age como vínculo entre os métodos de ciclo de vida e desenho das Views do Android e a implementação dos mesmos em código nativo, pelos componentes de browser. Além disso, fornece métodos para disparar o carregamento de páginas e de comunicação entre o código nativo e o Javascript nela executado.
- *WebSettings*: permite controlar diversas configurações de uma WebView, tais como a habilitação do acesso ao sistema de arquivos, bloqueio do carregamento de tipos de conteúdos (com vídeo, áudio, etc), de Javascript e o acesso à geolocalização, tamanho de fontes, entre outros.
- *WebViewClient*: fornece métodos para monitorar eventos relacionados ao carregamento de páginas, bem como controlar se o carregamento de uma nova página ou a execução de uma requisição HTTP deve ser efetuada.
- *CookieManager*: possibilita o controle dos cookies armazenados pelo componente de WebView, em nome de diversas origens.
- *WebDatabase*: possibilita o gerenciamento de dados de autenticação e de formulários HTML preenchidos pelo usuário.
- *WebStorage*: permite a inspeção e limpeza de dados armazenados pelas páginas através das APIs de persistência fornecidas por Javascript.

- *WebBackForwardList*: permite a manipulação do histórico de navegação entre páginas de uma instância de *WebView*.
- *WebIconDatabase*: fornece acesso à base de dados de ícones associados às páginas visitadas.

4.4.3 Estrutura e Operação do Aplicativo-Base

Na implementação do aplicativo-base em Android, uma classe do tipo *Activity* controla a execução dos aplicativos de terceiros em uma *WebView*, reservando para essa uma área em sua interface. Sobre essa área são desenhados os layers apartado e de sanitização, cujos eventos de interação do usuário são tratados exclusivamente pela *Activity*, como especificado nas seções anteriores. Antes do início da execução de um aplicativo *i*, o aplicativo-base consulta os serviços auxiliares do servidor, carregando a lista de sistemas externos acessíveis pelo aplicativo e as configurações de privacidade para acesso a dados do dispositivo, tal como ajustado pelo usuário. Inicialmente, associa-se à *WebView* uma classe de segurança *app_i*, e dispara-se a execução do aplicativo ao se carregar uma página com path “/app_*i*”. Uma instância de *WebViewClient* é utilizada para interceptar as requisições executadas pelas páginas carregadas, de forma a se atualizar a classe de segurança associada à *WebView*, bem como verificar a adequação das requisições à política de segurança da seção 4.1 e às configurações de acesso a sistemas externos ajustadas pelo desenvolvedor (especificadas nas listas anteriormente carregadas). Além disso, insere-se o *install_token* ao header das requisições, de forma a identificar no servidor dispositivo e usuário originadores da requisição. Adicionalmente, a interface entre código nativo e Javascript fornece acesso de preenchimento ao layer apartado (através da função **populate**), à execução de rotinas críticas e aos dados do dispositivo, regulando tais acessos em

função da mesma política e das configurações de privacidade ajustadas pelo usuário, como mostrado na seção 4.3.2 (na implementação atual, os dados do dispositivo fornecidos aos aplicativos restringem-se a leituras do GPS, por simplicidade). A classe *WebSettings* é utilizada para restringir o acesso a arquivos de áudio e vídeo, cujas requisições de *streaming* não são capturadas pelo *WebViewClient*. Quando da necessidade da execução de limpeza, funcionalidades das classes de *CookieManager*, *WebDatabase*, *WebBackForwardList*, *WebIconDatabase* e *WebView* são utilizadas para eliminar o estado global visível do componente de *WebView*, permitindo o reinício da execução do aplicativo a partir de um módulo de classe de segurança inferior.

4.5 Funcionamento Integrado

Nesta seção, demonstra-se, através da descrição de um aplicativo-exemplo, o funcionamento integrado dos mecanismos IFC do aplicativo-base e do servidor, bem como a maneira com que as arquiteturas de aplicativos de terceiros devem ser seccionadas dadas as restrições impostas por esses mecanismos. Para tanto, imagina-se o cenário de uma loja de aplicativos mantida por uma operadora de cartão de crédito, e um aplicativo de terceiro que utiliza dados privados dos usuários para lhes recomendar restaurantes próximos às suas localizações geográficas, exibindo num mapa a localização do cliente e dos restaurantes recomendados. Para realizar tal tarefa, o aplicativo utiliza dados privados da operadora associados aos usuários, dados de um banco de dados de restaurantes e promoções armazenados e administrados em um sistema externo, bem como, naturalmente, a geolocalização do dispositivo. Além da funcionalidade de recomendação de restaurantes, o aplicativo fornece ainda a possibilidade de fazer check-in em um dado restaurante, visualizar outros usuários nele presentes e realizar pagamentos com um cartão de crédito.

Os dados fornecidos aos aplicativos de terceiro estruturam-se como ilustrado no diagrama de entidade-relacionamento da figura 13. Nesse diagrama, observa-se uma entidade principal denominada “Cliente”, que lista dados de clientes da operadora de cartão de crédito, tais como “Nome”, “Data de nascimento”, “Endereço”, etc. Relacionada à entidade “Cliente”, a entidade “Cartão de crédito” agrega dados dos cartões de crédito – os produtos da operadora, em si – que os clientes possuem, contendo dados como “Limite”, “Saldo”, “Categoria”, “Taxas”, entre outros. Já a entidade “Fatura mensal” contém dados de um mês de transações dos cartões, contendo dados como “Valor”, “Data de vencimento”, “Data de pagamento”, “Valor pago”, entre outros. Por fim, a enti-

dade “Lançamento”, relacionada à entidade “Fatura mensal”, contém dados de transações particulares dos cartões, tais como o “Nome do estabelecimento”, “Categoria do estabelecimento”, “Valor”, “Número de parcelas”, dentre outros.

Figura 13: Diagrama E-R dos dados fornecidos às aplicações de terceiro

A atribuição dos níveis de segurança h ou l a esses dados é arbitrária, dependendo tanto das configurações de privacidade do usuário quanto da política de segurança institucional. Essa atribuição, como já dito, é feita pelo módulo de *Internal Gateway* do servidor, que fornece acesso aos dados de sistemas internos. A figura 14 exibe a tela do aplicativo-base onde o usuário configura suas opções de privacidade, definindo quais de seus dados são de categoria h ou l .

Figura 14: Tela de configurações de privacidade para o aplicativo

Os dados particulares do aplicativo de recomendações, por sua vez, organizam-se segundo o diagrama de entidade-relacionamento da Figura 15. Nele, observa-se a entidade “Estabelecimento”, que possui os atributos

“Nome”, “Localização geográfica”, “Gênero”, entre outros. Relacionada a essa entidade, uma tabela de “Ofertas” agrega dados promocionais dos estabelecimentos, contendo campos como “Descrição”, “Categoria”, “Valor”, entre outros. Tais dados, como já mencionado, são armazenados em sistemas externos ao ambiente de execução da loja de aplicativos, pois imagina-se que a administração de tais dados faz parte do negócio dos desenvolvedores do aplicativo.

Figura 15: Diagrama E-R dos dados do aplicativo de recomendação de restaurantes

4.5.1 Inicialização

A Figura 16 ilustra a tela inicial do aplicativo de recomendações, sendo executado na implementação do aplicativo-base em Android. A área em azul no topo da tela (denominada *Toolbar* na terminologia dos componentes de interface Android) consiste numa área da interface reservada ao aplicativo-base, ao passo que a área logo abaixo é reservada à *WebView*. A Figura 14, por sua vez, exibe uma tela pertencente ao próprio aplicativo-base, acessível pelo botão à direita no topo da tela da Figura 16. Essa tela dá acesso a configurações de privacidade para o aplicativo de recomendações, configurando quais dados do usuário são de nível de segurança *h* ou *l*. Na figura, categorias de dados com *checkbox* selecionados são considerados como de nível de segurança *l*.

Para executar a página exibida na Figura 16, os componentes do aplicativo e do mecanismo executam a sequência de interações exibida na Figura

Figura 16: Tela inicial do aplicativo de recomendações, sendo executado na implementação Android do aplicativo-base

Figura 17: Exemplo de interação de módulos na arquitetura (inicialização)

17 (observe-se que a maioria dos componentes confiáveis no servidor são emitidos, como o é o componente com nível de segurança l). No passo 1, o aplicativo-base carrega as configurações de privacidade do aplicativo e do usuário dos serviços auxiliares, e ajusta o rótulo associado à WebView para app_i (assumindo-se i , agora, como um pseudônimo para o aplicativo de recomendações). Em seguida, no passo 3, dispara-se o carregamento da página inicial do aplicativo na WebView, que executa uma requisição ao módulo de path $/app_i$. Esse módulo, para produzir a resposta, consulta arquivos de classe $globals$, lê e escreve em arquivos de classe app_i e se comunica com

sistemas internos, obtendo o nome do usuário exibido na tela.

4.5.2 Recomendação

A funcionalidade de recomendação de restaurantes ilustra as restrições impostas pelo mecanismo IFC a um aplicativo manipulando dados de segurança h , bem como esse deve ser estruturado para consultar dados corporativos desse nível de segurança e processá-los em conjunto com dados provenientes de sistemas externos.

Essa funcionalidade, acessada pela interface exibida na Figura 18, é alcançada através da tela inicial. Para executá-la, o aplicativo efetua as interações representadas na Figura 19. No passo 1, a página pede ao aplicativo-base a leitura dos dados do GPS e o preenchimento do layer apartado com um conjunto de botões, um no canto superior esquerdo (com a inscrição “fechar”), e outros exibidos lado-a-lado na parte inferior (com as inscrições “Italiana”, “Japonesa”, etc). O aplicativo-base preenche o layer apartado no passo 3 (com a função **populate**) e devolve a leitura do GPS ao aplicativo, que o transmite ao módulo de path “/app_i” do servidor. Esse, então, busca dados de sistemas externos referentes a ofertas de restaurantes de uma determinada cozinha (passo 4) próximos à localização do usuário, em seguida armazenando esses dados e a localização do cliente em arquivo. O aplicativo, então, consulta um serviço de mapas (passo 6) para exibir o mapa dos arredores da localização do usuário. Em seguida, realiza uma requisição ao módulo de path “/app_i/h” do servidor (passo 7), o que causa uma mudança no rótulo de segurança da WebView (passo 8). Para responder tal requisição, o módulo no servidor lê as ofertas anteriormente armazenadas (passo 9) e consulta dados de nível h dos sistemas internos, filtrando tais ofertas em função desses dados e armazenando o resultado em arquivo (passo 11).

Figura 18: Tela da funcionalidade de recomendação

Figura 19: Exemplo de interação de módulos na arquitetura (recomendação)

Note-se que, uma vez que a WebView encontra-se com nível de segurança h (a partir do passo 8), o aplicativo fica restrito a se comunicar somente com seus módulos server-side de nível de segurança também h . Como a capacidade de comunicação desses é também restrita, impede-se que o aplicativo divulgue dados secretos a entidades não autorizadas.

Para fazer nova carga de dados de sistemas externos mesmo ainda com capacidade de comunicação restrita, o aplicativo deve fazer uso do layer apartado do aplicativo-base, que, a partir da interação como o usuário, aciona

requisições aos módulos server-side de nível de segurança l do aplicativo. Assim, como ilustrado na Figura 20, quando o usuário toca em um dos botões do layer apartado que lista uma categoria de cozinha (por exemplo, “Japonesa”), o aplicativo-base faz uma requisição ao módulo de path “/app_i” do aplicativo no servidor, utilizando parâmetros anteriormente especificados pelo próprio aplicativo (através da função populate). Para responder a essa requisição, o módulo então consulta nos sistemas externos ofertas dessa categoria de cozinha (passo 2), armazenando-as em arquivos (passo 3). O aplicativo-base, depois de recebida a resposta do servidor, notifica a página executada na WebView (passo 4), que então faz uma requisição ao módulo de path “/app_i/h”, que para respondê-la, como no caso anterior, lê as ofertas carregadas (passo 6), filtra-os com dados de nível de segurança h (passo 7) e armazena em arquivo os resultados (passo 8).

Figura 20: Exemplo de interação de módulos na arquitetura

O layer apartado do aplicativo-base também pode ser utilizado para fazer com que o aplicativo recupere sua capacidade inicial de comunicação, quando, depois do evento de interação do usuário, o aplicativo-base realiza a limpeza de dados da WebView utilizada para o confinamento do aplicativo. No

exemplo, quando o usuário aciona o botão “fechar” da Figura 20, que é exibido no layer apartado, a operação de limpeza de WebView é disparada, após a qual a página inicial do aplicativo (da Figura 16) é novamente carregada, segundo uma sequência de passos similar à da Figura 17.

4.5.3 Check-In e Pagamento

A funcionalidade de “Check-in e pagamento” ilustra como o mecanismo IFC do aplicativo-base, através do layer de sanitização, protege a integridade dos dados de entrada de um determinado conjunto de operações críticas, que não são diretamente acessíveis pelos aplicativos de terceiro. Essa funcionalidade, acessível a partir da tela inicial do aplicativo e implementada quase totalmente por sistemas externos, inicialmente utiliza-se dos dados de geocalização do aparelho para exibir uma lista de restaurantes próximos a ele (Figura 21). Nessa lista, o usuário clica em um dos restaurantes exibidos, fazendo *check-in* no estabelecimento. Essa ação o leva à tela da Figura 22, onde se exibe uma lista de usuários que também fizeram *check-in* no mesmo estabelecimento.

Figura 21: Telas da funcionalidade de check-in e pagamento

A funcionalidade de pagamento, acessível pela tela à direita na Figura 21, dispara a interação entre módulos ilustrada na Figura 23. No passo 1, a página do aplicativo consulta nos sistemas externos um determinado valor a ser pago, e requisita o pagamento desse valor através da interface entre código nativo e Javascript. O aplicativo-base, então, exibe o layer de sanitização (Figura 22, passo 3) ao usuário, que ao autorizar o pagamento, dispara uma requisição ao módulo de serviços auxiliares (passo 4), que efetua a transação nos sistemas internos (passo 5).

Figura 22: Layer de sanitização exibido na funcionalidade de pagamento

Figura 23: Exemplo de interação de módulos na arquitetura (pagamento)

5 ANÁLISE DA PROPOSTA

A arquitetura anteriormente apresenta é avaliada nesta seção sob três aspectos: segurança, impactos de desempenho e programabilidade. Com isso, posiciona-se a proposta frente aos trabalhos apresentados no capítulo 3.

5.1 Segurança

Em função da potencial existência de aplicativos maliciosos no cenário considerado, exige-se que os mecanismos IFC empregados na arquitetura considere modelos de atacantes centrados nos desenvolvedores de aplicativos. Tais atacantes são naturalmente mais poderosos do que seriam atacantes exclusivamente usuários, tendo-se em vista que os primeiros poderiam explorar, no mínimo, as mesmas vulnerabilidades que os últimos. Por esse motivo, em termos da proteção dos fluxos de informação, é fundamental considerar-se não somente a transmissão de informações através de canais de comunicação explícitos, como também de canais encobertos.

A análise aqui apresentada concentra-se nos aspectos de proteção de confidencialidade. Adicionalmente, restringe-se ao estudo de canais de comunicação nos dispositivos móveis dos usuários, ou entre esses e o servidor, desconsiderando-se a transmissão de informação através de canais de comunicação internos ao servidor. Tal restrição justifica-se pelo fato de que a segurança em servidores de aplicações Web dotados de IFC já ter sido exten-

sivamente estudada em diversos dos trabalhos apresentados na seção 3.2, tendo se demonstrado que alguns deles são capazes de fornecer garantias de não-interferência. A utilização de um desses trabalhos para a implementação do server-side da arquitetura é, assim, considerada suficiente para atingir-se os requisitos de segurança apresentados.

5.1.1 Limitações do Atacante

Consideram-se nesta análise atacantes que almejam burlar a confidencialidade dos dados corporativos aos quais obtém acesso através dos aplicativos que desenvolvem, divulgando esses dados a entidades não autorizadas. As capacidades desses atacantes se restrinjam a: distribuir aos usuários aplicativos executados sob as restrições da arquitetura proposta; assumir o papel, eles próprios, de usuários do sistema; e distribuir aplicativos nativos das plataformas móveis diretamente aos dispositivos dos usuários, através dos meios usuais de distribuição das mesmas. Considera-se, em adição, que esses atacantes possam burlar o funcionamento do mecanismo IFC somente através de suas limitações intrínsecas (por exemplo, através de canais encobertos não rastreados, ou através dos meios de desclassificação), descartando-se atacantes que sejam capazes de comprometer o sandbox exercido pelas plataformas móveis em que o aplicativo-base é executado, ou de comprometer o funcionamento do aplicativo-base e do mecanismo IFC do servidor.

Deve-se ressaltar que o funcionamento adequado do aplicativo-base constitui a premissa de maior risco assumida nessa análise, em razão do mecanismo IFC empregado no mesmo possuir maior TCB. Isso se deve à complexidade do componente de WebView, que, devido à sua implementação da ordem de milhões de linhas de código, aumenta a probabilidade de ocorrência de bugs que comprometam o confinamento – como, de fato, já ocorreu

em versões de Android anteriores à 4.2 (LUO, 2011). Adicionalmente, a implementação desse componente se dá majoritariamente em bibliotecas de código C e C++, que frequentemente possuem, entre outras, vulnerabilidades como buffer e integer overflow, viabilizando a perpetração de ataques de control-hijacking e execução de código arbitrário. Assim, tais vulnerabilidades possibilitariam a um atacante capaz de explorá-las não só suplantar os mecanismos IFC do aplicativo-base, como também executar código arbitrário com a mesma autoridade desse – e, portanto, tendo acesso aos mesmos recursos do aplicativo-base. Assim, tendo acesso ao *install_token* armazenado em disco, o aplicativo poderia efetuar requisições a módulos de quaisquer outros aplicativos com a autoridade do usuário, e divulgar arbitrariamente os dados privados desse, bem como executar chamadas a procedimentos críticos em seu nome.

Muito embora seja possível a presença de vulnerabilidades no componente de WebView, deve-se ressaltar que as mesmas têm grande probabilidade de ser encontradas e corrigidas. Tal fato se deve à importância, no contexto das plataformas móveis, do componente de WebView, que é compartilhado entre todos os aplicativos nativos que dele fazem uso.

Como última observação, note-se que presença de um aplicativo-base comprometido não adiciona riscos de segurança ao dispositivo do usuário, pois os aplicativos executados pelo aplicativo-base terão direito a, no máximo, os mesmos privilégios de segurança que o último possui. Evidentemente, as plataformas móveis são, muitas vezes, vulneráveis a ataques que levam à subversão de seus mecanismos de segurança – por exemplo, StageFright (SOFTWARE ENGINEERING INSTITUTE, 2015). Entretanto, não faz parte deste estudo analisá-las, tendo em vista que elas constituem ameaça à segurança do dispositivo como um todo, e podem ser perpetradas mais facilmente por aplicativos nativos do que por código Javascript executado em WebViews.

5.1.2 Garantias de Não-Interferência

Fornece-se aqui a intuição de que o mecanismo IFC do aplicativo-base é capaz de garantir não-interferência na execução dos aplicativos, controlando os canais de comunicação que poderiam ser utilizados para transmitir informações em desacordo com a política exercida. A rigor, a garantia de não-interferência característica do mecanismo é, segundo as definições apresentadas na seção 2.2, insensível a terminação e temporização. Em adição, o aplicativo-base torna inviável a utilização de canais de comunicação de terminação e temporização entre as páginas de um mesmo aplicativo, reduzindo consideravelmente suas taxas de transferência. Assume-se ser improvável a utilização de canais de comunicação físicos, dada a impossibilidade de leitura, pelos aplicativos, de grandezas físicas além do tempo de processamento. Entretanto, persiste a possibilidade de uso de canais de recurso compartilhado, devido ao isolamento potencialmente imperfeito tanto entre instâncias de WebView (que pode acarretar a transmissão de informações dentro do aplicativo-base) quanto entre aplicativos nativos nas plataformas móveis (que pode levar à transmissão de informações de algum aplicativo, executado pelo aplicativo-base, a outro aplicativo nativo executado no mesmo dispositivo).

Em termos simples, as garantias de não-interferência providas pelo aplicativo-base residem na impossibilidade de execução simultânea de diferentes instâncias de WebView e na inexistência de canais de comunicação explícitos entre elas, mesmo executadas em instantes de tempo diferentes. Como detalhado no capítulo 4, à única instância de WebView mantida em execução por instante de tempo atribui-se rótulo também único, que denota um limite superior das classes de segurança de todos os dados mantidos no componente de WebView e acessíveis à página em execução (como elementos da árvore DOM, variáveis Javascript, cookies, entre outros). A essa página, por sua vez,

é permitida comunicação somente através de requisições aos próprios módulos, se respeitada a relação $\sqsubseteq$ entre os rótulos da WebView e o rótulo da URL acessada. Se autorizada a requisição, o rótulo do componente é atualizado de acordo com a função de combinação de classes, o que impõe um padrão de crescimento monotônico ao rótulo de segurança da WebView e torna sua capacidade de comunicação progressivamente mais limitada.

Sob o aspecto das garantias de segurança, tais restrições possibilitam ao mecanismo fornecer não-interferência, pois o resultado de computações realizadas em uma página com rótulo de segurança r_1 não pode ser comunicado, através de requisições HTTP, para um módulo de rótulo de segurança r_2 , se $r_1 \sqsubseteq r_2$ não for satisfeita. Além disso, havendo uma única instância de WebView em execução, elimina-se a existência de canais de comunicação legítimos ou encobertos internamente ao aplicativo-base, e os possíveis efeitos que esses canais teriam sobre as garantias de não-interferência, como se observa nos canais implícitos inter-processos apresentados no capítulo 2.

Mesmo não havendo mais de uma instância de WebView ao mesmo tempo no processo do aplicativo-base, note-se que o componente de WebView alocado em memória é reaproveitado em instantes de tempos diferentes para a execução de códigos de classes de segurança diferentes. A operação de limpeza do componente de WebView, que restaura a valores padrão o estado global do componente, visível pelas páginas de aplicativos executadas, impossibilita o estabelecimento de canais de comunicação entre módulos de classes de segurança incompatíveis segundo a política exercida. Note-se, porém, que a operação de limpeza não tem mapeados em sua operação a eliminação de canais encobertos provenientes do compartilhamento de recursos do processo (por exemplo, memória alocada ou capacidade de espaço em disco, como detalha o capítulo 2), que podem ser utilizados para transmitir

dados entre duas instâncias de WebView, depois da limpeza do componente. Essa possibilidade é agravada pelo tamanho do componente de WebView em termos do número de linhas de código, que, como dito, aumenta o tamanho do TCB do mecanismo IFC, e, conseqüentemente, o número de variáveis de estado do componente.

5.1.2.1 Controle dos canais de temporização e terminação

A operação de limpeza da WebView, como já detalhado no capítulo 4, só pode ser comandada pelo aplicativo-base. Essa restrição reside na necessidade de se limitar a taxa de transmissão de canais de temporização que, sem ela, poderiam ser utilizados para transmitir dados entre páginas, mesmo que não haja canais de estado de recurso compartilhado. Para ilustrar tal situação, suponha que os aplicativos pudessem disparar no aplicativo-base o processo de limpeza através de uma função Javascript denominada **clean_and_load**, que, depois da limpeza, carrega na WebView o conteúdo encontrado na URL fornecida como argumento. Os trechos de código 7 e 8, respectivamente, ilustram em pseudocódigo os códigos-fonte das páginas “timing_attack_1.htm” e “timing_attack_2.htm” (funções em negrito destacam a consultas ao servidor), do aplicativo malicioso *i*. A página “timing_attack_1.htm”, ao ser carregada, recupera o tempo de início de sua execução (linha 1) e o armazena no módulo de path “/app_i” do aplicativo, no servidor. Em seguida, ela lê um segredo *s*, consultando o módulo “/app_i” do servidor (linha 3), o que atribui um rótulo de segurança de nível de segurança *h* ao WebView do aplicativo-base. A depender do valor de *s*, a execução da página é interrompida por 2 segundos (linhas 4 a 6). Em seguida, a página pede ao aplicativo-base a limpeza da WebView, especificando como argumento “timing_attack_2.htm”, a página a ser carregada em seguida (linha 7). O WebView é então limpo, e a página

“timing_attack_2.htm” é carregada. Essa, então, lê o valor de seu tempo de início de processamento (linha 1), recupera o tempo de início de processamento da página “timing_attack_1.htm” do servidor (linha 2), e os compara (linhas 3 a 7): se o valor é maior do que um certo limiar ¹, sabe-se que s é 1; se não, sabe-se que s é 0.

```

1. t_i := getTime();
2. store_init_time(t_i);
3. secret := retrieve_secret();
4. if (secret = 1)
5.     sleep(1 s);
6. endif
7. clean_and_load('timing_attack_2.htm');

```

Trecho de código 7: timing_attack_1.htm

```

1. t_f := getTime();
2. t_i := retrieve_init_time();
3. if (t_f - t_i > LIMAR)
4.     secret := 1;
5. else
6.     secret := 0;
7. endif

```

Trecho de código 8: timing_attack_2.htm

Esse canal de temporização é inviabilizado ao se restringir o comando da operação de limpeza ao aplicativo-base, que, por sua vez, só o faz ao receber inputs do usuário. Desse modo, para se utilizar tal canal, torna-se necessário ludibriar o usuário a fornecer tais inputs (especificamente, clicar em botões da área nativa de desclassificação, como detalhado no capítulo 4), e de forma repetida, para se transmitir mais de um bit. Não só é improvável que se consiga ludibriar o usuário a fazê-lo, como também seria difícil contabilizar o tempo que esse levaria para inserir tais inputs, bem como a variância desse

¹O valor de limiar pode ser ajustado previamente para eliminar o efeito, por exemplo, do tempo para transmissão na rede e variância do mesmo

tempo, o que impossibilita o estabelecimento do parâmetro limiar (da linha 3) confiável de diferença para se determinar inequivocamente a transmissão de um bit.

Um canal de terminação funcionaria de modo semelhante ao canal de temporização descrito, sendo necessária, dessa vez, a atuação do usuário para disparar a operação de limpeza. Como exemplo, os trechos de códigos 9 e 10 ilustram em pseudocódigo as páginas “`termination_attack_sender.htm`” e “`termination_attack_receiver.htm`”, do aplicativo i , que transmitem um bit secret através de um canal de temporização. A página “`termination_attack_sender.htm`”, antes de recarregar, carrega no layer apartado um botão com o texto “Recarregar”, apontando para a página “`termination_attack_receiver.htm`” (linha 1). Então, como no ataque anterior, a página registra um tempo de início de execução e o armazena no módulo de nível de segurança app_i do servidor (linhas 2 e 3), lê o bit secreto de um módulo de nível de segurança h (na linha 4 aumentando o nível de segurança da WebView também para h), e, sendo o bit secreto igual a 1, entra em loop infinito (linhas 5 a 9), ou sendo 0, exibe uma mensagem na WebView pedindo que o usuário clique no botão recarregar (linha 10). A página “`termination_attack_receiver.htm`”, ao iniciar, recupera o tempo de início armazenado pela página anterior no módulo I (linhas 1 e 2), e o compara com o seu tempo atual de processamento (linhas 3 a 6); sendo a diferença entre eles maior do que certo limiar, sabe-se que o bit secreto é 1, ou 0, caso contrário. Naturalmente, o funcionamento desse método depende do usuário clicar no botão “recarregar”, havendo ocorrência de loop infinito ou não na página “`termination_attack_sender.htm`”; e depende também de haver diferença perceptível entre o tempo que o usuário leva para clicar no botão “recarregar” quando há loop infinito ou não. Note-se que, dadas essas restrições, esse canal de co-

municação possui taxa de transmissão e confiabilidade muito baixas, sendo inutilizável.

```

1. populate('Recarregar')
2. t_i := getTime();
3. store_init_time(t_i);
4. secret := retrieve_secret();
5. if (secret = 1)
6.     while(true);
7.     nop;
8.     endloop
9. endif
10. alert('Ocorreu um erro. Clique em recarregar');

```

Trecho de código 9: timing_attack_1.htm

```

1. t_f := getTime();
2. t_i := retrieve_init_time();
3. if (t_f - t_i > LIMIAI)
4.     secret := 1;
5. else
6.     secret := 0;

```

Trecho de código 10: timing_attack_2.htm

5.1.3 Segurança das operações de desclassificação

Embora não enunciados com tal finalidade, um conjunto de operações previstas na arquitetura proposta envolve, em sua essência, a desclassificação de informação, fato que cria a necessidade de se examinar a segurança das mesmas. Esse conjunto de operações é formado pela comunicação com sistemas externos – executada tanto no aplicativo-base, quanto no servidor –, a comunicação inter-aplicativos, a limpeza e redirecionamento de instâncias de WebView e o acesso a rotinas críticas, executadas no aplicativo-base.

Em relação à comunicação com sistemas externos e inter-aplicativos, verifica-se que a desclassificação envolve menor risco pois ocorre com infor-

mações de classe, no máximo, app_i , considerada de menor importância pela instituição e pelos usuários. Além disso, essas operações envolvem sistemas externos ou aplicativos previamente listados pelos desenvolvedores, não havendo possibilidade de manipular, em tempo de execução, para qual entidade a desclassificação é feita.

Quanto à operação de limpeza e redirecionamento de instâncias de WebView, verifica-se a possibilidade de desclassificar informações de nível de segurança h a módulos server-side de classes app_i ou (app_i, u_j, l) , havendo risco maior na execução da mesma. Muito embora envolva a ação do usuário para ser utilizada – pois o mesmo tem de comandá-la diretamente através dos elementos de interface no layer apartado –, os elementos exibidos nesse layer, e a disposição dos mesmos, pode ser feita de maneira tal a induzir o usuário a revelar informações exibidas no WebView, o que pode constituir num canal de comunicação mais efetivo do que os canais de terminação ou temporização descritos na seção anterior, para transmitir uma quantidade pequena de dados. Tal ataque pode ser denominado de *phishing de desclassificação*, dada sua natureza semelhante à dos ataques de engenharia social.

A execução de rotinas críticas, por fim, pode ser igualmente utilizada para desclassificar informações de nível de segurança h do usuário executor a outros usuários do sistema, através dos próprios sistemas internos da instituição. Para tanto, essas rotinas críticas devem produzir um efeito visível a esses outros usuários do sistema. Uma vez que os dados de entrada das operações críticas são sempre submetidos à inspeção do usuário executor, a utilização desse mecanismo envolve a esteganografia da informação desclassificada, de forma a torná-la invisível ao usuário.

5.2 Desempenho

A despeito da ausência neste trabalho de estudos provendo dados experimentais, simulações ou modelos teóricos, os impactos de desempenho causados pela arquitetura proposta podem ter suas origens relacionadas a algumas das características fundamentais da mesma. Assim, utilizando como métrica preferencial o tempo de resposta do sistema – definido aqui como o tempo transcorrido entre a entrada de um input do usuário no aplicativo-base e o término da resposta a essa entrada –, as referidas características, bem como seus efeitos potenciais, são discutidas qualitativamente como segue:

- Camada interpretativa adicional: no aplicativo-base, os aplicativos de terceiros são restritos a especificar sua interface gráfica utilizando HTML e CSS, e a especificar sua lógica utilizando Javascript. Em relação a aplicativos nativos, que geralmente dispõem de execução nativa ou semi-interpretada e APIs especializadas de desenho gráfico, tais aplicativos encontram-se, portanto, em desvantagem em termos do tempo de resposta: primeiro, pela sobrecarga envolvida na interpretação de código; e, segundo, pela manipulação da interface exclusivamente através da árvore DOM, sujeita à mesma sobrecarga de interpretação. Essas questões, como apontado por Charland e Leroux (2011), são inclusive critérios para a escolha da técnica de implementação de aplicativos, entre as opções de implementação nativa, híbrida ou Web. Usualmente, trechos de execução crítica de aplicativos podem ser implementados de modo nativo, ao passo que o restante não crítico do aplicativo pode permanecer como páginas Web, comunicando-se com a parte nativa através da interface Javascript-nativo da WebView. No entanto, essa escolha não é possível na arquitetura proposta, pois ocasionaria o rompimento

do Sandbox Web ao qual os aplicativos estão submetidos. A interface Javascript-nativo é utilizada somente para leitura de dados de sensores e para o controle do layer apartado. No entanto, como observa Corral, Sillitti e Succi (2012), a utilização dessa interface também é fonte de sobrecargas em relação a um aplicativo nativo.

- Uso de rede: em relação a aplicativos nativos, espera-se que os aplicativos na arquitetura proposta possuam utilização mais frequente da rede, principalmente em função de sua natureza Web, que os faz recuperar frequentemente do servidor código-fonte e recursos utilizados em tempo de execução. Assim, esperam-se diferenças no tempo de resposta em relação a aplicativos nativos, cujos código fonte e recursos se encontram usualmente no dispositivo.
- Operação de limpeza e redirecionamento de WebViews: a realização desta operação ocasiona a remoção de código e recursos de uma página armazenados tanto em memória quanto em arquivos. Assim, torna-se necessário recuperá-los repetidamente do servidor, o que acarreta em novos acessos a rede, e a um novo processo de parsing de HTML e CSS e interpretação Javascript. Assim, um aplicativo na arquitetura proposta está em desvantagem quanto ao tempo de resposta não somente em relação a aplicações nativas, mas também a aplicações híbridas ou Web, que podem fazer uso do cache para, ao menos, evitar um novo acesso à rede.
- Separação de privilégios no servidor: um servidor IFC, como apresentado no capítulo 3.2.2, é projetado de forma a separar os privilégios de códigos executando com diferentes classes de segurança. A depender da técnica de implementação do servidor – principalmente, utilizando-se OS-IFC, técnica que é mais compatível com a arquitetura proposta –,

essa separação implica na impossibilidade de compartilhamento de recursos, que é fundamental para o desempenho do mesmo. Assim, perde-se a possibilidade de compartilhar pools de conexões com bancos de dados, *threads* e *caches* de arquivos estáticos em memória.

5.3 Programabilidade

Em relação a aplicativos móveis implementados de forma híbrida ou nativa, ou até mesmo aplicações Web móveis, o conjunto de funcionalidades implementáveis por aplicativos na arquitetura proposta é limitado, em razão do confinamento por ela imposto e pela política de confidencialidade nela exercida. Além disso, o ferramental disponível aos programadores para a construção desse subconjunto implementável de funcionalidades é também limitado, pelas mesmas razões.

No aplicativo-base, a limitação de funcionalidades é evidente pela restrição de funcionamento dos aplicativos ao sandbox Web, que, como já detalhado, tem o objetivo de restringir os canais de comunicação disponíveis aos aplicativos. Assim, esses não gozam da possibilidade de executar código nativo e, portanto, não possuem acesso a APIs nativas da plataforma móvel. Usualmente, em aplicações móveis híbridas, o acesso a funcionalidades providas por essas APIs é feito através da interface de comunicação Javascript-nativo, mas no caso da arquitetura proposta, esse acesso é restrito ao modo somente leitura de sensores, das câmeras, GPS e microfones. Essas restrições têm diversas consequências sobre o conjunto de funcionalidades implementáveis na arquitetura e o ferramental disponível para fazê-lo, que podem ser descritas – novamente, de maneira qualitativa – como segue:

- Acesso a APIs de interface gráfica e desenho da plataforma: estando

restritos ao ambiente de execução Web, os aplicativos só podem especificar e manipular suas interfaces utilizando HTML, CSS e Javascript. Entretanto, as APIs de interface gráfica das plataformas móveis fornecem rotinas especializadas de manipulação de elementos gráficos, usualmente já de acordo com os padrões gráficos da plataforma, e com desempenho superior ao fornecido pelos mecanismos de renderização de páginas Web (SERRANO; HERNANTES; GALLARDO, 2013). Assim, torna-se mais difícil criar os mesmos padrões utilizando-se somente do feramental Web. Além disso, o acesso a rotinas de baixo nível de manipulação gráfica paralela (recentemente introduzidas nos Web-browsers) não é universalmente disponibilizada em WebViews, o que impossibilita a escrita de aplicações que dependam fundamentalmente desse recurso (como jogos em terceira dimensão, por exemplo).

- I/O, comunicação inter-processos e compartilhamento de dados: devido à sua natureza Web, os aplicativos têm a utilização de I/O restrita ao conjunto de mecanismos intermediados pelo aplicativo-base, perfazendo inputs do usuário (que ocorrem, num dispositivo de toque, através de toques, gestos e o teclado virtual), a saída de vídeo e áudio, requisições HTTP e ao sistema de arquivos (através das APIs Javascript) e à leituras de sensores do dispositivo. Funcionalidades que requeiram maior controle de funções de I/O (por exemplo, bases de dados ou controle de periféricos) são incompatíveis com a arquitetura do aplicativo-base. Além disso, as plataformas móveis (notadamente o Android) especificam mecanismos de alto-nível para que os diversos aplicativos nativos se comuniquem, trocando dados para a realização de tarefas em colaboração. Na arquitetura proposta, porém, esses mecanismos não são disponibilizados, pois dão origem a canais de comunicação que, como no caso

anterior, transgrediriam os limites de confinamento do aplicativo-base e da área de memória da WebView. Porém, é permitido que um aplicativo *a1* faça requisições ao módulo de path “/app_a2” de um aplicativo *a2*, como detalhado no capítulo 4.

- Ciclo de vida: por serem executados sob as restrições de uma WebView, que é um componente das APIs de interface gráfica das plataformas móveis, os aplicativos estão sujeitos ao mesmo ciclo de vida de execução desses componentes – isto é, estão restritos a serem executados quando a interface gráfica que os contém está ativa, interagindo com o usuário. Essa restrição é incompatível com funcionalidades usualmente implementadas por aplicativos nativos ou híbridos, que utilizam de porções de código que executam sem necessariamente a interação do usuário. Esse tipo de componente é útil para executar tarefas longas, como a execução de mídias, sem a necessidade de interação ininterrupta do usuário, ou para receber assincronamente notificações do sistema ou do próprio servidor, sem estar inicialmente sendo executado.
- Configurações de hardware e software do aparelho: é vetada aos aplicativos na arquitetura proposta a visualização ou manipulação de configurações globais de hardware e software do dispositivo, como o controle de conexões wifi, brilho da tela, dispositivos bluetooth ou USB, configurações de interface, como planos de fundo e atalho, entre muitas outras configurações. O motivo para tal restrição é evitar o estabelecimento de canais de compartilhamento de recurso que tornariam possível estabelecer a comunicação entre páginas de nível de segurança incompatíveis.

Essas restrições, apesar de aparentemente severas, ilustram que os aplicativos devem se comportar de forma mais similar a uma aplicação Web do que, a de fato, um aplicativo móvel. Para funcionalidades de visualização e

manipulação de dados institucionais – que são o foco da proposta apresentada neste trabalho – tais restrições acabam por ter um impacto menor, como discutido na seção 4.5. Nessa seção, evidenciou-se ainda que a arquitetura proposta é compatível com ferramentas e processos habituais de desenvolvimento de aplicações Web – isto é, permitindo a utilização das mesmas linguagens, bibliotecas, frameworks e arquiteturas habituais.

No servidor, as restrições de programabilidade são resultado da separação de privilégios imposta pelo mecanismo IFC. Assim, é necessário que o desenvolvedor divida o código do aplicativo em módulos (correspondentes a sujeitos como processos ou threads) com os privilégios adequados, como detalhado no capítulo 3. A restrição mais severa desse modelo é a ausência de módulos que manipulem informações de nível de segurança h de mais de um usuário, sendo essa uma consequência do nível de granularidade de rastreio do mecanismo adotado. Entretanto, a utilização de um mecanismo de granularidade de rastreio mais grosseira aumenta possibilita uma melhor compatibilidade com linguagens, frameworks e aplicações já utilizados.

5.4 Comparação com Trabalhos Relacionados

A proposta apresentada neste trabalho baseia-se em conceitos provenientes do emprego de mecanismos IFC em dispositivos móveis e em navegadores e servidores web. Por esse motivo, a proposta se relaciona a trabalhos das três categorias, sendo π Box, BFlow, COWL, Asbestos e Hails os mais relevantes para comparação com a mesma, dentre os do capítulo 3.

No π Box, que se destina à proteção de dados corporativos em aplicativos móveis, verifica-se o mesmo princípio de conectar aplicativos móveis a servidores corporativos, em que também há o emprego de mecanismos IFC.

Entretanto, π Box atinge esse objetivo através da modificação do sistema operacional dos dispositivos móveis, possibilidade que é excluída do cenário aqui considerado. Verifica-se, quanto à programabilidade fornecida por π Box, que esse possibilita o desenvolvimento de aplicativos nativos e é compatível com aplicativos já desenvolvidos (de determinadas categorias). Sob esses aspectos, a programabilidade da presente proposta é menor (já que nela deve-se desenvolver aplicativos em HTML e Javascript), mas deve-se ressaltar que ela ainda permite aos aplicativos a comunicação com sistemas externos, diferentemente de π Box. Quanto à segurança, pode-se argumentar que, por utilizar uma granularidade de rastreo mais grosseira, π Box deve oferecer melhor isolamento, enquanto que a presente proposta emprega um isolamento potencialmente mais imperfeito. Quanto ao desempenho, verifica-se que, por utilizar uma camada interpretativa adicional, a presente proposta impõe sobre custos maiores do que os impostos por π Box.

Em relação a BFlow e COWL, que se destinam à aplicação de IFC em web browsers, a presente proposta apresenta uma granularidade de rastreo mais grosseira, pois os primeiros atribuem rótulos a frames HTML, enquanto aqui se associam rótulos de segurança ao componente inteiro de WebView. Por esse motivo, pode-se argumentar que a presente proposta apresenta impactos de desempenho mais severos do que os dois trabalhos, mas deve-se lembrar que o layer de sanitização pode compensar essa desvantagem. Entretanto, é difícil tecer argumentos acerca das diferenças dos impactos de desempenhos trazidos entre as diferentes propostas.

Quanto a Asbestos e Hails, que empregam IFC em servidores web, cabe-se destacar somente que esta proposta adiciona a proteção de código do lado do cliente necessária para fornecer a elas melhores garantias de segurança.

6 CONSIDERAÇÕES FINAIS E TRABALHOS FUTUROS

O presente trabalho considera o problema do emprego de mecanismos IFC em dispositivos móveis, sob as restrições de se manter inalterados os sistemas operacionais e de runtime das plataformas móveis; de se permitir o emprego de linguagens, bibliotecas e frameworks de uso habitual no desenvolvimento de aplicativos móveis; e de se fornecer garantias robustas de segurança frente a um cenário de aplicativos possivelmente maliciosos. A arquitetura proposta fornece uma alternativa prática frente a esse cenário, valendo-se do emprego de tecnologias Web e da integração de um middleware equipado com IFC no cliente, a um servidor onde módulos de aplicativos, de classificação de segurança distintos, são mantidos também sob o controle de mecanismos IFC. Com isso, garante-se a compatibilidade com linguagens, bibliotecas e frameworks de uso habitual (isto é, HTML, CSS, Javascript e tecnologias relacionadas), bem como se determina alterações somente em nível de aplicação no lado do cliente – já que o *sandbox* Web é largamente disponível nos frameworks de plataformas móveis. Adicionalmente, a utilização desse sandbox consagrado possibilita a obtenção de garantias de segurança robustas.

O aplicativo-exemplo apresentado no capítulo 4 mostra que a arquitetura proposta ainda permite o desenvolvimento de aplicações de funcionalidades atrativas, muito embora forneça acesso limitado às APIs nativas dos dispositi-

tivos. A razão para isso é que essa restrição, embora incompatível com funcionalidades relacionadas à utilização e gerenciamento diretos do hardware ou sistema operacional, não se configura como impeditivo para funcionalidades cujos cernes sejam visualização e manipulação de dados – em especial, os dados corporativos fornecidos através dos módulos *server-side*. A arquitetura ainda permite que aplicativos consultem sob demanda sistemas externos ao seu ambiente de execução, possibilitando a implementação de funcionalidades que agregam a visualização de dados de origens diversas (como se observa nas populares páginas Web com *mash-up*), ou que requerem o processamento em serviços externos. Como consequência da existência de módulos que manipulam dados de diversos usuários, a arquitetura ainda permite a implementação de funcionalidades de compartilhamento, colaboração e redes sociais. Note-se que, com essas duas últimas características, introduzir-se-ia na arquitetura a possibilidade de vazamento de dados entre usuários ou para sistemas externos, o que se evita limitando-as à manipulação de dados públicos. Porém, a distinção entre dados públicos e privados é configurável, podendo-se inclusive deixar a cargo dos próprios usuários a definição de quais de seus dados devem ser considerados de uma categoria ou de outra.

Em adição ao modelo de execução Web dos aplicativos, a arquitetura proposta ainda especifica um meio de comunicação entre o aplicativo-base e os aplicativos que possibilita aos mesmos facilitar a navegação entre suas páginas face às restrições impostas pelo uso do mecanismo IFC. Assim, a disposição de interface gráfica no aplicativo-base é particular, dividindo uma área para a exibição do conteúdo HTML do aplicativo e outra área nativa, que controla a navegação entre páginas do aplicativo (sob comandos do usuário) e é parcialmente especificada por esse. A atuação da área nativa sobre o conteúdo carregado na área de HTML determina que, caso os dados persistidos

no componente executando a página sejam incompatíveis com outra página a ser carregada, efetue-se a limpeza desse componente. Tal artifício pode ser generalizado como a divisão da interface gráfica em partições que exibem conteúdo de códigos não-confiáveis e outra área que exibe código confiável, que permite o controle da execução dos anteriores. Até onde se sabe, essa técnica é inédita na área de estudo de IFC, portanto consistindo em uma das contribuições mais originais deste trabalho.

6.1 Publicações

Com a pesquisa bibliográfica utilizada para a elaboração deste trabalho, publicou-se o artigo *A Survey on Information-Flow Control Mechanisms in Web Applications* na conferência *The 2015 International Conference on High Performance Computing & Simulation* (HPCS 2015). Planeja-se a publicação de um artigo adicional em periódico especializado, descrevendo detalhadamente o funcionamento da arquitetura proposta.

6.2 Trabalhos Futuros

A arquitetura proposta neste trabalho fomenta a realização de trabalhos futuros, tanto com o propósito de aperfeiçoá-la e adicionar-lhe funcionalidades, como também de utilizar as formulações nela propostas em novos cenários e aplicações.

No intuito de aperfeiçoar a arquitetura, destacam-se as seguintes ideias:

- Especificação de APIs para fornecer aos aplicativos acesso a mais recursos nativos de hardware ou do sistema operacional: neste trabalho esses acessos se limitam à leitura de sensores e ao GPS. Assim, pode-se,

por exemplo, fornecer um meio de acesso à câmera do dispositivo, ou de envio de notificações assíncronas ao aplicativo-base, ou de acesso ao sistema de arquivos. Em ambos os casos, é necessário o estudo dos mecanismos de ambas as funcionalidades, para que esses não originem canais de estado de recurso compartilhado.

- Introdução de páginas off-line: ao se introduzir a possibilidade de páginas offline nos aplicativos, permitir-se-ia que os usuários utilizassem aplicativos sem conexão de rede. No entanto, essa funcionalidade obviamente exigiria o armazenamento offline das páginas, o que, de certa forma, requer o espelhamento no aplicativo-base de algumas funcionalidades do mecanismo IFC *server-side*.
- Execução de código Javascript sem interface: ao depender de um componente de interface, o código dos aplicativos só pode ser executado quando conectado a uma interface em execução. No entanto, pode-se estender o *sandbox* Javascript para confinar código de aplicativos que rodem sem essa restrição, permitindo que os mesmos executem em background, como, por exemplo, para executar operações assíncronas. As consequências de segurança de tal funcionalidade, porém, devem ser detalhadamente estudadas.
- Divisão dos componentes de *WebView* por classe de segurança: a operação de limpeza, como detalhado no capítulo 4, ocasiona prejuízos de desempenho aos aplicativos, dada a limpeza de caches e do conteúdo em memória, que deve ser recarregado e reinterpretado. Ao se isolar as instâncias do componente de *WebView* por diferentes classes de segurança, um número delas poderia ser mantido em memória, um em execução por vez. Assim, em vez de executar a operação de limpeza, chavear-se-ia a instância em execução. Tal ideia exige o estudo pro-

fundo do componente de *WebView* para garantir o isolamento efetivo dentre os aplicativos executados.

- Separação inter-processo dos componentes de *WebView* e do aplicativo-base: o código dos aplicativos é executado no mesmo processo do aplicativo-base e, portanto, o isolamento entre códigos de diferentes classes de segurança é arriscado pelo compartilhamento de recursos desse processo. Ao se efetuar a divisão entre um processo do aplicativo-base e outro para execução dos aplicativos (inspirando-se na arquitetura do projeto Chromium (BARTH, 2008), alcançar-se-ia melhor isolamento entre os mesmos. Essa é uma possível alternativa, inclusive, para a implementação do item anterior.

Para a utilização de ideais da arquitetura proposta em outros cenários, destaca-se a divisão de interface gráfica proposta no aplicativo-base, que é inédita na área de estudo de IFC e parece promissora para utilização com mecanismos que genericamente devem executar código não-confiável com interface gráfica. Em especial, a utilização em navegadores Web suscita a discussão de um novo modelo de política de controle de acesso Web, baseado não somente na política de mesma origem, mas também em políticas de controle de fluxos, implementadas com uma granularidade de rastreo em nível de página.

REFERÊNCIAS

- AGGARWAL, G. et al. An analysis of private browsing modes in modern browsers. In: *Proceedings of the 19th USENIX Conference on Security*. Berkeley, CA, USA: USENIX Association, 2010. (USENIX Security'10), p. 6–6. ISBN 888-7-6666-5555-4.
- ARZT, S. et al. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *SIGPLAN Not.*, ACM, New York, NY, USA, v. 49, n. 6, p. 259–269, jun. 2014. ISSN 0362-1340.
- ASKAROV, A.; CHONG, S. Learning is change in knowledge: Knowledge-based security for dynamic policies. In: *Computer Security Foundations Symposium (CSF), 2012 IEEE 25th*. [S.l.: s.n.], 2012. p. 308–322. ISSN 1940-1434.
- ASKAROV, A. et al. Termination-insensitive noninterference leaks more than just a bit. In: *Proceedings of the 13th European Symposium on Research in Computer Security: Computer Security*. Berlin, Heidelberg: Springer-Verlag, 2008. (ESORICS '08), p. 333–348. ISBN 978-3-540-88312-8.
- ASKAROV, A.; SABELFELD, A. Gradual release: Unifying declassification, encryption and key release policies. In: *Security and Privacy, 2007. SP '07. IEEE Symposium on*. [S.l.: s.n.], 2007. p. 207–221. ISSN 1081-6011.
- BARTH, A. et al. *The security architecture of the Chromium browser*. 2008.
- BELL, D. E.; LAPADULA, L. J. Secure computer systems: Mathematical foundations. *Data Base*, National Technical Information Service, v. 1, n. MTR-2547 Vol. I, p. 513–523, 1973.
- BIBA, K. J. Integrity Considerations for Secure Computer Systems. *Proceedings of the 4th Annual Symposium on Computer Architecture*, ACM, v. 5, n. 7, p. 66, 1977.
- BIRGISSON, A.; RUSSO, A.; SABELFELD, A. Unifying facets of information integrity. In: *Information Systems Security*. [S.l.]: Springer Berlin Heidelberg, 2010, (Lecture Notes in Computer Science, v. 6503). p. 48–65. ISBN 978-3-642-17713-2.
- BISHOP, M. *Computer Security: Art and Science*. [S.l.]: Addison-Wesley, 2003. ISBN 9780201440997.

BORTZ, A.; BONEH, D. Exposing private information by timing web applications. *Proceedings of the 16th international conference on World Wide Web - WWW '07*, ACM Press, New York, New York, USA, p. 621, 2007.

BOUDOL, G.; CASTELLANI, I. Noninterference for concurrent programs and thread systems. *Theoretical Computer Science*, v. 281, p. 109–130, 2002. ISSN 0304-3975. Selected Papers in honour of Maurice Nivat.

BRUMLEY, D.; BONEH, D. Remote timing attacks are practical. *Computer Networks*, v. 48, n. 5, p. 701–716, ago. 2005. ISSN 13891286.

CARLIER, V. et al. Electromagnetic side channels of an fpga implementation of aes. 2004.

CHARLAND, A.; LEROUX, B. Mobile application development: Web vs. native. *Commun. ACM*, ACM, New York, NY, USA, v. 54, n. 5, p. 49–53, maio 2011. ISSN 0001-0782.

CHLIPALA, A. Ur/web: A simple model for programming the web. In: *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. New York, NY, USA: ACM, 2015. (POPL '15), p. 153–165. ISBN 978-1-4503-3300-9.

CHONG, S. et al. Secure web applications via automatic partitioning. In: *ACM. SIGOPS Operating Systems Review*. [S.l.], 2007. v. 41, p. 31–44.

CLARKSON, M. R.; MYERS, A. C.; SCHNEIDER, F. B. Quantifying information flow with beliefs. *J. Comput. Secur.*, IOS Press, Amsterdam, The Netherlands, The Netherlands, v. 17, n. 5, p. 655–701, out. 2009. ISSN 0926-227X.

CORRAL, L.; SILLITTI, A.; SUCCI, G. Mobile multiplatform development: An experiment for performance analysis. *Procedia Computer Science*, v. 10, p. 736 – 743, 2012. ISSN 1877-0509. {ANT} 2012 and MobiWIS 2012.

DE GROEF, W. et al. Flowfox: A web browser with flexible and precise information flow control. In: *Proceedings of the 2012 ACM Conference on Computer and Communications Security*. New York, NY, USA: ACM, 2012. (CCS '12), p. 748–759. ISBN 978-1-4503-1651-4.

DE RYCK, P. et al. Security of web mashups: A survey. In: _____. *Information Security Technology for Applications: 15th Nordic Conference on Secure IT Systems, NordSec 2010, Espoo, Finland, October 27-29, 2010, Revised Selected Papers*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012. p. 223–238. ISBN 978-3-642-27937-9.

DENNING, D. E. A lattice model of secure information flow. *Commun. ACM*, ACM, New York, NY, USA, v. 19, n. 5, p. 236–243, maio 1976. ISSN 0001-0782.

DEVRIESE, D.; PIESENS, F. Noninterference through secure multi-execution. In: *Security and Privacy (SP), 2010 IEEE Symposium on*. [S.l.: s.n.], 2010. p. 109–124. ISSN 1081-6011.

EFSTATHOPOULOS, P. et al. Labels and event processes in the asbestos operating system. In: ACM. *SIGOPS Operating Systems Review*. [S.l.], 2005. v. 39, p. 17–30.

ENCK, W. et al. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In: *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*. Berkeley, CA, USA: USENIX Association, 2010. (OSDI'10), p. 1–6.

ENCK, W.; ONGTANG, M.; MCDANIEL, P. Understanding android security. *Security Privacy, IEEE*, v. 7, n. 1, p. 50–57, Jan 2009. ISSN 1540-7993.

GARSIEL, T.; IRISH, P. *How Browsers Work*. 2011. Acessado em: 10/08/2015. Disponível em: <<http://www.html5rocks.com/en/tutorials/internals/howbrowserswork/>>.

GENKIN, D.; SHAMIR, A.; TROMER, E. Rsa key extraction via low-bandwidth acoustic cryptanalysis. Springer Berlin Heidelberg, v. 8616, p. 444–461, 2014.

GIFFIN, D. B. et al. Hails: Protecting data privacy in untrusted web applications. In: *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*. Berkeley, CA, USA: USENIX Association, 2012. (OSDI'12), p. 47–60. ISBN 978-1-931971-96-6.

GOGUEN, J. A.; MESEGUER, J. Security policies and security models. In: . Los Alamitos, CA, USA: IEEE Computer Society, 1982. v. 0, p. 11. ISSN 1540-7993.

GOOGLE INC. *Google Web Toolkit Project*. 2006. Acessado em: 08/03/2015. Disponível em: <<http://www.gwtproject.org>>.

GORRIERI, R.; MARTINELLI, F.; MATTEUCCI, I. Towards information flow properties for distributed systems. *Electronic Notes in Theoretical Computer Science*, Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, v. 236, p. 65–84, abr. 2009. ISSN 1571-0661.

GROSSKURTH, A.; GODFREY, M. A reference architecture for web browsers. In: *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on*. [S.l.: s.n.], 2005. p. 661–664. ISSN 1063-6773.

HEDIN, D. et al. Jsflow: Tracking information flow in javascript and its apis. In: *Proceedings of the 29th Annual ACM Symposium on Applied Computing*. New York, NY, USA: ACM, 2014. (SAC '14), p. 1663–1671. ISBN 978-1-4503-2469-4.

HEDIN, D.; SABELFELD, A. A perspective on information-flow control. In: *Software Safety and Security - Tools for Analysis and Verification*. [S.l.]: IOS Press, 2012, (NATO Science for Peace and Security Series - D: Information and Communication Security, v. 33). p. 319–347. ISBN 978-1-61499-027-7.

HOSEK, P. et al. Safeweb: A middleware for securing ruby-based web applications. In: *INTERNATIONAL FEDERATION FOR INFORMATION PROCESSING. Proceedings of the 12th International Middleware Conference*. [S.l.], 2011. p. 480–499.

HU, W. et al. The build dependency perspective of android's concrete architecture. In: *Mining Software Repositories (MSR), 2012 9th IEEE Working Conference on*. [S.l.: s.n.], 2012. p. 128–131. ISSN 2160-1852.

IBTIMES. *Android malware discovered on Google Play has infected millions of users with spyware*. 2016. Acessado em: 01/06/2016. Disponível em: <<http://www.ibtimes.co.uk/android-malware-discovered-google-play-store-1553341>>.

IDC INC. *Smartphone OS Market Share 2015, 2014, 2013, and 2012*. 2015. Acessado em: 12/11/2015. Disponível em: <<http://www.idc.com/prodserv/smartphone-os-market-share.jsp>>.

JACKSON, C. et al. Protecting browser state from web privacy attacks. In: *Proceedings of the 15th International Conference on World Wide Web*. New York, NY, USA: ACM, 2006. (WWW '06), p. 737–744. ISBN 1-59593-323-9.

KASHYAP, V.; WIEDERMANN, B.; HARDEKOPF, B. Timing- and Termination-Sensitive Secure Information Flow: Exploring a New Approach. In: *Proceedings of the IEEE Symposium on Security and Privacy*. [S.l.]: IEEE, 2011. p. 413–428. ISBN 978-1-4577-0147-4.

KOCHER, P.; JAFFE, J.; JUN, B. Differential power analysis. Springer Berlin Heidelberg, v. 1666, p. 388–397, 1999.

KROHN, M.; TROMER, E. Noninterference for a practical difc-based operating system. In: *Security and Privacy, 2009 30th IEEE Symposium on*. [S.l.: s.n.], 2009. p. 61–76. ISSN 1081-6011.

KROHN, M. et al. Information flow control for standard OS abstractions. In: *ACM. SIGOPS Operating Systems Review*. [S.l.], 2007. v. 41, p. 321–334.

LAMPSON, B. Perspectives on protection and security. In: *SOSP History Day 2015*. New York, NY, USA: ACM, 2015. (SOSP '15), p. 4:1–4:21. ISBN 978-1-4503-4017-5.

LAMPSON, B. W. A note on the confinement problem. *Commun. ACM*, ACM, New York, NY, USA, v. 16, n. 10, p. 613–615, out. 1973. ISSN 0001-0782.

LAMPSON, B. W. Protection. *SIGOPS Operating Systems Review*, ACM, New York, NY, USA, v. 8, n. 1, p. 18–24, jan. 1974. ISSN 0163-5980.

LEE, S. et al. π box: A platform for privacy-preserving apps. In: *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*. Berkeley, CA, USA: USENIX Association, 2013. (nsdi'13), p. 501–514.

LUO, T. et al. Attacks on webview in the android system. In: *Proceedings of the 27th Annual Computer Security Applications Conference*. New York, NY, USA: ACM, 2011. (ACSAC '11), p. 343–352. ISBN 978-1-4503-0672-0.

MIGLIAVACCA, M. et al. Distributed middleware enforcement of event flow security policy. In: *Proceedings of the ACM/IFIP/USENIX 11th International Conference on Middleware*. Berlin, Heidelberg: Springer-Verlag, 2010. (Middleware '10), p. 334–354. ISBN 978-3-642-16954-0.

MYERS, A. C. Jflow: Practical mostly-static information flow control. In: *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. New York, NY, USA: ACM, 1999. (POPL '99), p. 228–241. ISBN 1-58113-095-3.

MYERS, A. C.; LISKOV, B. A decentralized model for information flow control. *SIGOPS Operating Systems Review*, ACM, New York, NY, USA, v. 31, n. 5, p. 129–142, out. 1997. ISSN 0163-5980.

ORACLE INC. *The Java Virtual Machine Specification*. 2013. Acessado em: 19/12/2015. Disponível em: <<https://docs.oracle.com/javase/specs/jvms/se7/html/>>.

ORACLE INC. *Java Native Interface*. 2014. Acessado em: 10/08/2015. Disponível em: <<http://docs.oracle.com/javase/7/docs/technotes/guides/jni/>>.

ORACLE INC. *Java Platform, Enterprise Edition (Java EE) Specification*. 2015. Acessado em: 19/12/2015. Disponível em: <<http://www.oracle.com/technetwork/java/javaee/tech/index.html>>.

RICE, H. G. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, v. 74, p. 358–366, 1953.

RUSSO, A. et al. Advances in computer science - asian 2006. secure software and related issues: 11th asian computing science conference, tokyo, japan, december 6-8, 2006, revised selected papers. In: _____. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007. cap. Closing Internal Timing Channels by Transformation, p. 120–135. ISBN 978-3-540-77505-8.

SABELFELD, A.; MYERS, A. C. Language-based information-flow security. *IEEE J. Sel. A. Commun.*, IEEE Press, Piscataway, NJ, USA, v. 21, n. 1, p. 5–19, set. 2006. ISSN 0733-8716.

SABELFELD, A.; RUSSO, A. From dynamic to static and back: Riding the roller coaster of information-flow control research. In: *Perspectives of*

Systems Informatics. [S.I.]: Springer Berlin Heidelberg, 2010, (Lecture Notes in Computer Science, v. 5947). p. 352–365. ISBN 978-3-642-11485-4.

SABELFELD, A.; SANDS, D. Probabilistic noninterference for multi-threaded programs. In: *Proceedings of 13th IEEE Computer Security Foundations Workshop*. [S.I.: s.n.], 2000. p. 200–214. ISSN 1063-6900.

SCHAEFER, M. et al. Program confinement in kvm/370. In: *Proceedings of the 1977 Annual Conference*. New York, NY, USA: ACM, 1977. (ACM '77), p. 404–410. ISBN 978-1-4503-3921-6.

SCHMIDT, D. C. et al. *Pattern-Oriented Software Architecture, Patterns for Concurrent and Networked Objects*. [S.I.]: John Wiley & Sons, 2013.

SERRANO, N.; HERNANTES, J.; GALLARDO, G. Mobile web apps. *Software, IEEE*, v. 30, n. 5, p. 22–27, Sept 2013. ISSN 0740-7459.

SMITH, G.; VOLPANO, D. Secure information flow in a multi-threaded imperative language. In: *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. New York, NY, USA: ACM, 1998. (POPL '98), p. 355–364. ISBN 0-89791-979-3.

SOFTWARE ENGINEERING INSTITUTE. *Vulnerability Note 924951: Android Stagefright contains multiple vulnerabilities*. 2015. Acessado em: 22/09/2015. Disponível em: <<https://www.kb.cert.org/vuls/id/924951>>.

SPETT, K. Cross-site scripting. *SPI Labs*, v. 1, p. 1–20, 2005. Disponível em: <<http://people.cs.ksu.edu/hankley/d764/Topics/SPIcross-sitescripting.pdf>>.

STATISTA. *Number of apps available in leading app stores*. 2015. Acessado em: 12/11/2015. Disponível em: <<http://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores>>.

STEFAN, D. et al. Addressing covert termination and timing channels in concurrent information flow systems. In: *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*. New York, NY, USA: ACM, 2012. (ICFP '12), p. 201–214. ISBN 978-1-4503-1054-3.

STEFAN, D. et al. Disjunction category labels. In: *Proceedings of the 16th Nordic Conference on Information Security Technology for Applications*. Berlin, Heidelberg: Springer-Verlag, 2012. (NordSec'11), p. 223–239. ISBN 978-3-642-29614-7.

STEFAN, D. et al. Flexible dynamic information flow control in haskell. In: *Proceedings of the 4th ACM Symposium on Haskell*. New York, NY, USA: ACM, 2011. (Haskell '11), p. 95–106. ISBN 978-1-4503-0860-1.

STEFAN, D. et al. Protecting users by confining javascript with cowl. In: *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. Broomfield, CO: USENIX Association, 2014. p. 131–146. ISBN 978-1-931971-16-4.

TRUSTWAVE. *Application Vulnerability Trends Report*. 2014. Disponível em: <<https://www2.trustwave.com/test-the-rest-vulnerability-report.html>>.

VOLPANO, D.; IRVINE, C.; SMITH, G. A sound type system for secure flow analysis. *J. Comput. Secur.*, IOS Press, Amsterdam, The Netherlands, The Netherlands, v. 4, n. 2-3, p. 167–187, jan. 1996. ISSN 0926-227X.

VOLPANO, D. M. Safety versus secrecy. In: *Proceedings of the 6th International Symposium on Static Analysis*. London, UK, UK: Springer-Verlag, 1999. (SAS '99), p. 303–311. ISBN 3-540-66459-9.

W3C. *The XMLHttpRequest Object*. 2006. Acessado em: 05/08/2015. Disponível em: <<http://www.w3.org/TR/2006/WD-XMLHttpRequest-20060405/>>.

WHITE HAT SECURITY. *Website Security Statistics Report*. 2013. Disponível em: <<https://www.whitehatsec.com/resource/stats.html>>.

XU, Y.; WITCHEL, E. Maxoid: Transparently confining mobile applications with custom views of state. In: *Proceedings of the Tenth European Conference on Computer Systems*. New York, NY, USA: ACM, 2015. (EuroSys '15), p. 26:1–26:16. ISBN 978-1-4503-3238-5.

YIP, A. et al. Privacy-preserving browser-side scripting with bflow. In: *Proceedings of the 4th ACM European Conference on Computer Systems*. New York, NY, USA: ACM, 2009. (EuroSys '09), p. 233–246. ISBN 978-1-60558-482-9.

YIP, A. et al. Improving application security with data flow assertions. In: *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*. New York, NY, USA: ACM, 2009. (SOSP '09), p. 291–304. ISBN 978-1-60558-752-3.

YOSHIHAMA, S. et al. Dynamic information flow control architecture for web applications. In: *Computer Security - ESORICS 2007*. [S.l.]: Springer Berlin Heidelberg, 2007, (Lecture Notes in Computer Science, v. 4734). p. 267–282. ISBN 978-3-540-74834-2.

ZDANCEWIC, S.; MYERS, A. C. Observational determinism for concurrent program security. In: *In Proc. 16th IEEE Computer Security Foundations Workshop*. [S.l.: s.n.], 2003. p. 29–43.

ZELDOVICH, N. et al. Making information flow explicit in histar. In: USENIX ASSOCIATION. *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*. [S.l.], 2006. p. 263–278.