CENTRO FEDERAL DE EDUCAÇÃO TECNOLÓGICA
DE MINAS GERAIS - CAMPUS TIMÓTEO
Engenharia de Computação

Leonardo Filipe Rodrigues Ribeiro

IMPLEMENTAÇÃO DE INTERFACE EM UMA FERRAMENTA PARA ANÁLISE DE FLUXO DE INFORMAÇÃO

Belo Horizonte
Agosto de 2014

Leonardo Filipe Rodrigues Ribeiro

IMPLEMENTAÇÃO DE INTERFACE EM UMA FERRAMENTA PARA ANÁLISE DE FLUXO DE INFORMAÇÃO

Monografia apresentada ao Curso de Engenharia de Computação do Centro Federal de Educação Tecnológica de Minas Gerais, como requisito parcial para obtenção do título de bacharel em Engenharia de Computação.

Orientador: Bruno Rodrigues Silva

Belo Horizonte

Agosto de 2014

Leonardo Filipe Rodrigues Ribeiro

IMPLEMENTAÇÃO DE INTERFACE EM UMA FERRAMENTA PARA ANÁLISE DE FLUXO DE INFORMAÇÃO

Monografia apresentada ao Curso de Engenharia de Computação do Centro Federal de Educação Tecnológica de Minas Gerais, como requisito parcial para obtenção do título de bacharel em Engenharia de Computação.

Bruno Rodrigues Silva
Orientador - CEFET/MG

Deisymar Botega Tavares
CEFET/MG

Belo Horizonte
Agosto de 2014

Dedico esta monografia a meus pais pela fé e confiança demonstradas.
Aos meus amigos pelo apoio incondicional.
Aos professores pelo companheirismo e pela disposição de sempre ensinar.
Enfim, a todos que de alguma forma tornaram este caminho mais fácil de ser percorrido.

Agradecimentos

Agradeço a Deus pela oportunidade de realizar este trabalho. A meus pais, pelo incentivo e colaboração, principalmente nos momentos de dificuldade. À Nathália Araújo, pelo carinho e por me fazer uma pessoa melhor. Ao meu orientador Bruno Silva por estar sempre disposto a ajudar. Agradeço também aos meus colegas de trabalho e amigos pelo apoio nas horas difíceis e nas dificuldades e principalmente por me incentivar nesta caminhada, tornando-a mais rica e agradável.

"... E mais importante, tenha a coragem de seguir seu coração e sua intuição.
Eles de alguma forma já sabem o que você realmente quer se tornar.
Tudo mais é secundário."
(Steve Jobs)

Resumo

Vazamento de informações sigilosas e fluxo contaminado são dois importantes problemas em computação que merecem atenção por parte dos desenvolvedores. O primeiro diz respeito a quando uma informação sigilosa alcança canais públicos de saída através de fluxos de dados e de controle. Já o segundo problema acontece quanto existe um caminho em tais fluxos a partir de uma função de entrada pública até uma função sensível do programa, permitindo ataques do tipo *SQL Injection*, por exemplo. Como uma solução para o primeiro problema, existe uma ferramenta chamada Flow Tracking que efetua uma análise estática de código. Entretanto ela não é facilmente adaptada para tratar outros tipos de vulnerabilidades, tratando apenas o vazamento de endereços de memória. Sendo assim, neste trabalho de monografia esta ferramenta foi alterada aumentando seu domínio de análise. Agora é possível analisar outras vulnerabilidades, como o fluxo contaminado.

Palavras-chaves: LLVM, Fluxo de informação, Flow Tracking, Compiladores.

Abstract

Leak sensitive information and contaminated flow are two important issues in computing that deserve attention from developers. The first relates to when a confidential information reaches the public channels through data and control flows. The second problem happens when there is a path in such flows that starts in a public input to a sensitive function of the program allowing attacks such as SQL Injection. As a solution to the first problem, there is a tool called Flow Tracking which performs a static analysis of code. However, it is not easily adapted to treat other types of leaks. In this paper, that tool was modified by increasing its power of analysis. It is now possible to analyze other vulnerabilities, such as contaminated flow, the second above cited issue.

Key-words: LLVM, Information flow, Flow Tracking, Compilers.

Lista de ilustrações

Figura 1 – Exemplo de um fluxo explícito em C. 20
Figura 2 – Exemplo de um fluxo implícito em C. 20
Figura 3 – Exemplo de um programa que manipula informação sigilosa. 21
Figura 4 – Resultado da aplicação do sistema de tipos de Hunt e Sands no exemplo da figura 3. 23
Figura 5 – Fluxo de compilação do LLVM. 24
Figura 6 – Exemplo da figura 4 convertido para o formato SSA. 27
Figura 7 – Grafo de fluxo de informação criado para o programa visto na figura 4. O tipo de variável é mostrado ao lado do vértice que a representa. . . . 28
Figura 8 – Conteúdo do shellscript *dotgen.bin*. 33
Figura 9 – Exemplo de saída da execução do Flow Tracking. 34
Figura 10 – Programa com vazamento de informação por fluxo explícito. 34
Figura 11 – Saída do Flow Tracking para o programa da figura 10. 35
Figura 12 – Grafo de dependência para o programa da figura 10, considerando as chamadas em *assembly* LLVM. 35
Figura 13 – Grafo de dependência para o programa da figura 10, considerando as linhas do código-fonte. 36
Figura 14 – Programa com vazamento de informação por fluxo implícito. 37
Figura 15 – Saída do Flow Tracking para o programa da figura 14. 37
Figura 16 – Um dos grafos de dependência para o programa da figura 14, considerando as chamadas em *assembly* LLVM. 38
Figura 17 – Um dos grafos de dependência para o programa da figura 13, considerando as linhas do código-fonte. 38
Figura 18 – Estatísticas relacionadas aos grafos de dependência construídos para cada *benchmark*. 39
Figura 19 – Número de avisos de vazamento de informação. O gráfico está usando escala logarítmica. 39
Figura 20 – Exemplo da estrutura de um arquivo XML que pode ser lido pelo Flow Tracking modificado. 41
Figura 21 – Saída do Flow Tracking para o código 5.5. 46
Figura 22 – Grafo de dependência gerado para o código 5.5, considerando as chamadas em *assembly* LLVM. 46
Figura 23 – Grafo de dependência gerado para o código 5.5, considerando as linhas do código-fonte. 47

Lista de Códigos

3.1 Programa *teste.c*. 25
3.2 Código em linguagem *assembly* LLVM do programa *teste.c* 25
3.3 Atribuição de valores a variáveis. 26
3.4 Atribuição de valores a variáveis utilizando o SSA. 26
5.1 Estrutura da classe *parserXML*. 41
5.2 Código que utilizada a api do LLVM para receber parâmetros. 43
5.3 Parte do código Flow Tracking modificado que utilizada a classe *parserXML*. 43
5.4 Arquivo *teste1.xml*. 44
5.5 Programa vulnerável ao ataque *SQL Injection*. 45

Lista de abreviaturas e siglas

LLVM	Low Level Virtual Machine
XML	Extensible Markup Language
SQL	Structured Query Language

Sumário

1 INTRODUÇÃO 13
1.1 Definição do Problema 14
1.2 Motivação 14
1.3 Objetivo 15

2 ESTADO DA ARTE 17

3 FUNDAMENTOS TEÓRICOS 19
3.1 Fluxos Explícitos e Fluxos Implícitos 19
3.2 Fluxo contaminado 20
3.3 Vazamento de informação sigilosa 21
3.4 Formas de análise do fluxo da informação 22
3.4.1 O Sistema de Tipos de Hunt e Sands 22
3.4.2 Monitores puramente dinâmicos são inconsistentes 23
3.5 Flow Tracking 24
3.5.1 LLVM 24
3.5.1.1 Exemplo de uma representação intermediária de código de um programa 25
3.5.2 Formato de Atribuição Estática Única - SSA 26
3.5.3 A implementação concreta do sistema de tipagem 27
3.5.4 A necessidade de variáveis com estados invariantes 28
3.5.5 Grafo de fluxo 28

4 METODOLOGIA 30

5 RESULTADOS 32
5.1 Utilizando o Flow Tracking 32
5.2 Detectando vazamento de informação com o Flow Tracking 34
5.2.1 Fluxos explícitos 34
5.2.2 Fluxos implícitos 36
5.3 Análises feitas com o Flow Tracking 37
5.4 Modificação do algoritmo do Flow Tracking 39
5.4.1 Formato do arquivo XML 40
5.4.2 TinyXML 41
5.4.3 Flow Tracking modificado 41
5.4.4 Resultado de testes com Flow Tracking modificado 44
5.5 Considerações finais 47

6 CONCLUSÃO 48

Conclusão 48

7 PROPOSTAS DE CONTINUIDADE 49

Referências 50

1 INTRODUÇÃO

Atualmente, aplicações têm manipulado informações cada vez mais valiosas e sigilosas. As organizações utilizam sistemas de informação para dar suporte ao negócio e passam a depender da confidencialidade, disponibilidade e integridade das informações contidas nestes sistemas. A confidencialidade busca garantir que todo acesso à informação seja restrito somente àqueles usuários autorizados para tal acesso. A integridade demanda que a informação seja protegida contra alterações indevidas. A disponibilidade requer que a informação e os recursos do sistema estejam acessíveis a todo momento em que forem solicitados (BACE, 2000).

A informação é um ativo estratégico e também é um recurso de vital importância nas organizações. Por isso devem ser criados mecanismos com a finalidade de garantir a segurança desta informação. “A segurança da informação de uma empresa garante, em muitos casos, a continuidade de negócio, incrementa a estabilidade e permite que as pessoas e os bens estejam seguros de ameaças e perigos” (PHOENIX, 2014).

A área de segurança computacional possui diversos temas, como segurança em redes de computadores ou segurança em sistemas operacionais. Neste trabalho focaremos na segurança em programas. Em uma definição prática de segurança, um programa é considerado seguro quando se comporta de maneira esperada por seus usuários (GARFINKEL, 1996).

Estes sistemas, eventualmente, contêm vulnerabilidades que podem ser exploradas por meio de ataques. Ataques à programas vem de códigos maliciosos, geralmente utilizados por adversários. Adversários são pessoas ou sistemas que atacam a aplicação. Eles buscam tomar o controle de operações críticas do sistema ou atingir itens essenciais de informação explorando vulnerabilidades (SILVA; PEREIRA; OLIVEIRA, 2013). Uma vez exploradas por adversários, estas vulnerabilidades colocam organizações e usuários individuais sobre risco (GARFINKEL, 1996).

Dentre as técnicas de análise de programas, levando em conta a segurança, existe o rastreamento de fluxo de informação. Por informação entende-se qualquer dado ou conhecimento que permite a um adversário comprometer o correto funcionamento de um programa (SILVA; PEREIRA; OLIVEIRA, 2013). Os fluxos de informação são relações de dependência entre as partes de um programa, que podem ser variáveis ou funções por exemplo.

1.1 Definição do Problema

Se o adversário alimenta a aplicação sob ataque com dados maliciosos para tomar o controle de operações críticas, diz-se que esse programa é vulnerável a *ataques de fluxos contaminados*. Informação pode também trafegar na direção contrária. Se o programa permite que um adversário tome conhecimento sobre dados sigilosos a partir da leitura de suas funções de saída, então diz-se que esse código possui um *vazamento de informação* (SILVA; PEREIRA; OLIVEIRA, 2013).

Estuda-se a análise de fluxo de informação com base na necessidade de solução dos seguintes problemas:

1. **Vazamento de informações por canais públicos:** Existem informações sigilosas que não devem sair do programa por canais públicos como, por exemplos as funções *printf* e *putc*. Endereços de variáveis são exemplos deste tipo de informação. Este vazamento pode ocorrer explicitamente, através de transferência de dados entre variáveis, ou implicitamente, através de fluxo de controle do programa.

2. **Encontrar entradas que causam problemas de segurança no programa:** Um adversário, que possui informações do código-fonte do programa e acesso as variáveis de entrada, poderá inserir dados que explorem alguma vulnerabilidade presente do programa. A análise de fluxo de dados pode ser usada para prevenir estes ataques. Ataques do tipo *SQL Injection* (TRIPP et al., 2009) exploram estas vulnerabilidades.

1.2 Motivação

Através da análise do fluxo da informação é possível entender o comportamento do programa e detectar possíveis vulnerabilidades que podem ser exploradas. Por isso programação segura é uma área importante nestes dias em que é muito difundido o desenvolvimento de serviços acessados por diferentes usuários. A programação descuidada é a causa maior de erros nos programas e, conseqüentemente, de falhas de segurança em sistemas computacionais (LEVANDOSKI, 2011).

Em programas desenvolvidos em linguagem C, por exemplo, os programadores manipulam diretamente a memória e podem escrever códigos que são inseguros (SILVA; PEREIRA; OLIVEIRA, 2013). Atacantes podem aproveitar estas falhas para comprometer um sistema. Uma fonte comum de falha de segurança em aplicativos é a falta de validação correta de dados recebidos de usuários. Esta fraqueza leva a quase todas as principais vulnerabilidades em aplicações como *Interpreter Injection*, ataques de *Locale/Unicode*, ataques de sistemas de arquivos e *Buffer Overflow* (LEVANDOSKI, 2011).

Por esses motivos, tem surgido nos últimos anos um forte interesse em desenvolver técnicas para garantir que aplicações sejam, por construção (em tempo de desenvolvimento), seguras no que diz respeito aos dados que estas manipulam. Se o adversário alimenta a aplicação sob ataque com dados maliciosos para tomar o controle de operações críticas, diz-se que esse programa é vulnerável a ataques de fluxos contaminados. Se o programa permite que um adversário tome conhecimento sobre dados sigilosos a partir da leitura de suas funções de saída, então diz-se que esse código possui um vazamento de informação (SILVA; PEREIRA; OLIVEIRA, 2013).

Um dos benefícios em estudar técnicas para melhorar a segurança de aplicações é buscar desenvolver ferramentas que implementam estas técnicas provendo suporte para o desenvolvimento de programas mais seguros.

Uma ferramenta de detecção de vazamento de informação em programas será um dos objetos de estudo neste trabalho. Esta ferramenta contém algumas limitações. Este trabalho pretende aumentar o poder de análise desta ferramenta contribuindo para uma análise mais apurada de detecção de vulnerabilidades em sistemas.

1.3 Objetivo

Flow Tracking (SILVA; PEREIRA; OLIVEIRA, 2013) é uma ferramenta que analisa código fonte de programas feitos em C/C++ e reporta se estes programas contém vazamento de informação, mais precisamente vazamentos de endereços de variáveis por canais públicos. Esta ferramenta verifica o programa em tempo de compilação. Ela foi implementada na forma de uma analisador estático para o compilador LLVM.

O principal objetivo deste trabalho é estudar a ferramenta Flow Tracking e modificá-la de forma que seja possível a análise de outros tipos de vulnerabilidades além do vazamento de endereços. Prevê-se que será possível aumentar a abrangência de análise da ferramenta sendo possível detectar vulnerabilidades de fluxo contaminado. Um fluxo contaminado ocorre se existe um caminho através do qual uma informação, controlada pelo adversário, pode fluir e chegar até uma operação sensível no programa.

A ferramenta será analisada e testada utilizando alguns exemplos e situações. Hoje ela utilizada como fontes de vazamentos de endereços funções da biblioteca libc que alocam espaço em memória como *malloc*, *calloc* e *realloc*. E utiliza "saídas" funções que escrevem dados em canais públicos, como *printf*, *putc* e *fputc*. A modificação proposta na ferramenta tem o intuito de permitir customizar quais funções poderão ser fontes ou saídas dos fluxos de informação.

Tendo em vista este cenário, serão estudados os tipos de fluxos de informação em programas. Estes fluxos, que podem ser explícitos e implícitos (HUNT; SANDS, 2006),

permitem através da sua análise detectar vulnerabilidades no programa.

Para estes objetivos serem alcançados é necessário o estudo sistemático do compilador LLVM, que é base para o desenvolvimento da ferramenta, bem como os fundamentos de análise de fluxos de informação em programas.

2 ESTADO DA ARTE

Diversos trabalhos abordam os problemas da detecção de vazamento de informação. Alguns pesquisadores buscam descobrir vazamentos de informação via técnicas de análise estática. Outros preferem a análise dinâmica. E existem os grupos que combinam as duas técnicas. Essas técnicas, em geral, não são equivalentes. Algumas abordagens são mais conservadoras e reportam que programas seguros são vulneráveis. Outras são mais liberais, e deixam de reportar vulnerabilidades reais (SILVA; PEREIRA; OLIVEIRA, 2013).

As primeiras análises estáticas propostas para detectar vazamento de informação levavam em consideração somente dependências de dados (DENNING; DENNING, 1977). Essas análises sofrem de falsos negativos, deixando de reportar vulnerabilidades devido a fluxos implícitos de informação. Monitores puramente dinâmicos são ainda mais permissivos que esse tipo de análise estática. Esses monitores, como os propostos por Sabelfeld e Russo (RUSSO; SABELFELD, 2010), são isentos de falsos negativos: todo aviso é uma vulnerabilidade, de acordo com um modelo semântico da linguagem de programação usada. No entanto monitores puramente dinâmicos são incapazes de identificar todo vazamento de informação.

A análise estática proposta por Hunt e Sands (HUNT; SANDS, 2006) descobre vazamentos tanto devido a fluxos implícitos quanto explícitos. Contudo, neste trabalho se observa que, em situações práticas, a sua taxa de falsos positivos é muito alta. Em outras palavras, esse tipo de análise tende a indicar vulnerabilidades em programas que, em realidade, são seguros. O Flow Tracking utiliza uma implementação concreta deste sistema de tipagem que detecta fluxos implícitos de informação.

Russo e Sabelfeld propuseram, em 2010, um tipo de monitor dinâmico que reporta as mesmas vulnerabilidades que o sistema de tipos de Hunt e Sands (RUSSO; SABELFELD, 2010). Esse tipo de monitor é de difícil implementação, e nunca chegou a ser testado em programas reais.

Em 2011, Quadros e Pereira (QUADROS; PEREIRA, 2011) fizeram uma análise estática para detectar vazamentos de endereços com a motivação de se evitar ataques de *buffer overflow*. Eles citam que os sistemas operacionais modernos possuem proteções para ataques de *buffer overflow* como o mecanismo de proteção *ASLR - Address Space Layout Randomization*. Este mecanismo consiste em carregar os módulos binários correspondentes ao programa executável em diferentes endereços de memória cada vez que o programa é executado. Assim o atacante não terá um ponto fixo na memória para executar ataques como o *ROP - return-oriented programming* (BUCHANAN et al., 2008) já que a cada execução será atribuído um endereço diferente. Porém, mesmo um sistema protegido

por *ASLR* pode estar vulnerável caso alguns de seus programas permita que endereços alcancem funções públicas de saída.

De acordo com Silva (SILVA; PEREIRA; OLIVEIRA, 2013) fluxo contaminado é um problema complementar ao vazamento de informações. Neste problema, é importante saber se existe um caminho através do qual a informação pode fluir de uma entrada pública, que um adversário pode controlar, para uma operação sensível no programa. O fluxo contaminado foi formalizado por Orbaek e Palsberg (ORBÆK; PALSBERG, 1997), como uma instância de *type-checking*. Eles escreveram um sistema de tipos para linguagem $\lambda - calculus$ e provaram que se um programa passa na verificação de tipos, então ele não é vulnerável à ataques de fluxo contaminado. Dez anos mais tarde, Jovanovic et Al. (JOVANOVIC; KRUEGEL; KIRDA, 2006) implementaram um algoritmo que soluciona tal problema para linguagem PHP 4.0. Este algoritmo era uma análise de fluxo de dados baseada no sistema de tipos de Orbaek e Palsberg (ORBÆK; PALSBERG, 1997). Rimsa et al. (RIMSA et al., 2012) melhoram a complexidade das soluções anteriores para o problema de fluxo contaminado. Para isso, eles propõem uma solução sobre o número de variáveis do programa fonte. A ideia chave por trás dessa abordagem é a conversão do programa para uma representação intermediária chamada *e-SSA (Extend Static Single Assignment)* introduzida por Bodik et al. (BODIK; GUPTA; SARKAR, 2000). Esta representação torna possível resolver o problema de fluxo contaminado através da associação de restrições diretamente à variáveis do programa ao invés de associá-las à variáveis em cada ponto do programa.

Programas escritos em linguagem C e C++ são alvos de ataques por fluxo contaminado. Pois, estas linguagens permitem a execução de operações maliciosas que podem explorar fluxos contaminados, como técnicas que permitem o *buffer overflow*. *Buffer overflows* podem ser detectáveis através de técnicas de instrumentação de código ou modificações no código fonte. Em Santos (SANTOS; PEREIRA; OLIVEIRA, 2013) é apresentada uma ferramenta de auxílio ao programador, que gera anotações no arquivo fonte de programas C e C++, evidenciando possíveis problemas de *buffer overflow*.

3 FUNDAMENTOS TEÓRICOS

Diferentes partes de um sistema de computação em execução podem auxiliar no rastreamento de fluxo de informação: o sistema operacional, a arquitetura e o compilador. Neste trabalho serão tratadas soluções baseadas no compilador. Silva (SILVA; PEREIRA; OLIVEIRA, 2013) cita as seguintes categorias de classificação destas soluções:

1. **Análise Estática:** Somente considera o código do programa e geralmente não faz testes com dados de entrada. Esta análise fornece informações definitivas sobre o comportamento do programa e pode ser facilmente incorporada na implementação de um compilador. Porém, tende a ser uma análise conservadora em suas respostas, gerando falsos positivos. Sua execução pode ser lenta.

2. **Análise Dinâmica:** Necessita que o programa seja executado e é capaz de fornecer respostas precisas sobre o traço de programa executado. Por outro lado, as respostas são influenciadas por dados de entrada, desta forma não permite obter conclusões sobre o comportamento do programa de forma geral. Sua execução pode ser rápida.

3. **Análises Híbridas:** Esta solução combina ambas abordagens para melhor verificação e validação, pois a análise estática reduz parte do *overhead* da instrumentação e pode ser usada para geração de testes para a análise dinâmica.

A ferramenta Flow Tracking utiliza a análise estática em suas buscas por vazamentos de endereços de memória através de canais públicos. Esta análise é feita rastreando a informação que trafega no programa por meio de fluxos explícios e implícitos (SILVA; PEREIRA; OLIVEIRA, 2013).

3.1 Fluxos Explícitos e Fluxos Implícitos

A informação trafega nos programas via **relações de dependência**. Essas relações existem em duas formas: dados e controle. Se o programa contém uma instrução como v = *f(v1,v2,...,vn)*, então há uma dependência de dados de cada $vi, 1 \leq i \leq n$ para v. A dependência de dados, também chamada de *fluxo explícito*, deve-se à movimentação de bits de uma posição de memória para outra. Se um programa possui uma instrução como u = *f(...,v,...)*, então existe um fluxo explícito de dados de v para u. O exemplo de código da figura 1 retorna um endereço de memória para o usuário, através de um fluxo explícito.

No início do programa, foi criada uma variável i do tipo ponteiro para inteiro. A função *malloc* aloca um bloco de bytes consecutivos na memória do computador e

Figura 1 – Exemplo de um fluxo explícito em C.

```
#include <stdio.h>
#include <stdlib.h>
int main (int argc, char **argv) {
        int *i, *j, *k;
        i = (int *) malloc (8);
        j = i;
        j++;
        k = j;
        printf ("%p", k);
}
```

Fonte: (Flow Tracking software, 2013)

devolve o endereço desse bloco (FEOFILOFF, 2014). O número de bytes é especificado no argumento da função. Assim, na próxima linha a função *malloc* aloca 8 bytes na memoria e devolve o endereço do primeiro bloco para a variável *i*. Nas próximas linhas o endereço é copiado e incrementado, e no fim um endereço de memoria é impresso na tela.

Por outro lado, a dependência de controle ocorre quando uma instrução é capaz de determinar se outra será executada ou não. Por exemplo, se o valor de uma variável v depende de um teste condicional, então o programa contém uma dependência de controle entre v e p, o predicado que controla aquele teste. A sequência mostrada na figura 2 determina uma dependência de controle.

Figura 2 – Exemplo de um fluxo implícito em C.

```
p = (a > b);
if(p)
      v = 0;
else
      v = 1;
```

Fonte: o autor.

Dependências de controle criam os chamados fluxos implícitos de informação. Nesse caso, v depende de p, ou, posto doutro modo, informação flui implicitamente de p para v. Enquanto o rastreamento de dependências de dado é simples, e tem sido feito com sucesso em diversas análises de fluxo contaminado (RIMSA et al., 2012; TRIPP et al., 2009; JOVANOVIC; KRUEGEL; KIRDA, 2006), fluxos de controle ainda são um desafio. (SILVA; PEREIRA; OLIVEIRA, 2013)

3.2 Fluxo contaminado

A análise de fluxo contaminado é um tipo de análise de fluxo de informação. Assim, ela procura saber se existem caminhos através do qual uma operação pública, que pode

ser executada por um adversário, consegue fluir até chegar a uma operação sensível no programa (SILVA; PEREIRA; OLIVEIRA, 2013). Para evitar esta vulnerabilidade é necessário fazer um tratamento dos dados recebidos evitando entradas maliciosas.

Um adversário poderá atacar sistemas que possuem estas vulnerabilidades utilizando ataques de fluxo contaminado. Este tipo de ataque é bastante comum em sistemas web. O atacamente geralmente insere dados maliciosos em entradas públicas, como formulários HTML por exemplo, de forma a expor dados escondidos, ou sobrescrever dados valiosos, ou ainda executar comandos de sistema. *SQL Injection*, *cross-site scripting*, upload de arquivo malicioso, execução de comandos não desejáveis e ataques ao sistema de arquivos são exemplos de ataques de fluxo contaminado (SCOTT; SHARP, 2003).

Programas *standalone* escritos em linguagen C e C++ também são alvos de ataques por fluxo contaminado. Estas linguagens permitem o uso de técnicas que podem executar operações maliciosas. Por exemplo, o *buffer overflow* ocorre quando uma quantidade grande de dados é escrita em um buffer, ultrapassando os limites do buffer e causando sobreescrita de valores nas posições de memória adjacentes a este limite.

Geralmente o adversário objetiva sobreescrever dados sensíveis ou redirecionar o fluxo de controle para outro local na memória. Para isto ele deve escolher cuidadosamente quais entradas públicas ele deve usar para explorar estas vulnerabilidades.

3.3 Vazamento de informação sigilosa

O vazamento de informação ocorre quando uma informação, que deveria estar protegida, chega as mãos de um atacante por meio de um dos canais públicos disponíveis. A análise de vazamento de informação sigilosa é também um tipo de análise de fluxo de informação, mas se dá no sentido contrário do que acontece nos fluxos contaminados.

No código mostrado na figura 3, existe uma variável sigilosa *secret*. Esta variável por algum motivo recebe um valor que o adversário necessita. Se o adversário estiver de posse do código-fonte do programa e obtiver a resposta da execução do algoritmo, o mesmo poderá inferir o valor de *secret* após a execução do programa.

Figura 3 – Exemplo de um programa que manipula informação sigilosa.

```
secret := ?
teste := “branco”
p := 0
if (secret != 1) then
   p := 1
if (p == 0) then
   teste := “vermelho”
print teste
```

Fonte: o autor.

Nesse caso, muito embora o valor de *secret* não seja explicitamente copiado para nenhuma variável impressa pela função *printf*, fluxos implícitos revelam informação. Observando o valor da variável *teste* o adversário pode inferir que o valor de p é zero, desde que seja impresso o valor "vermelho". Se p é zero, o adversário enxerga que a variável *secret* foi carregada com o valor 1. Este conhecimento pode ser importante para uma ataque ao sistema e deve ser evitado.

Outro exemplo típico de informação sigilosa que não deve chegar aos usuários são os endereços de memória internos do programa. Isto é, os endereços de variáveis locais, globais ou dinâmicas. Quando um programa permite que algum endereço chegue a alguma função pública, tem-se uma vulnerabilidade de vazamento de endereços (SILVA; PEREIRA; OLIVEIRA, 2013). Informações sigilosas como estas podem ser usadas como auxílio em algum ataque ao sistema operacional.

3.4 Formas de análise do fluxo da informação

Conforme já mencionado, existem duas formas de análise de fluxo de informação: análise estática e análise dinâmica. A análise estática pode ser realizada por um compilador pois ela necessita apenas do código-fonte do programa, assim não é necessária a execução do programa. Já a análise dinâmica necessida da execução do programa além de um conjunto de dados de entrada e pode ser implementada através de instrumentação de código.

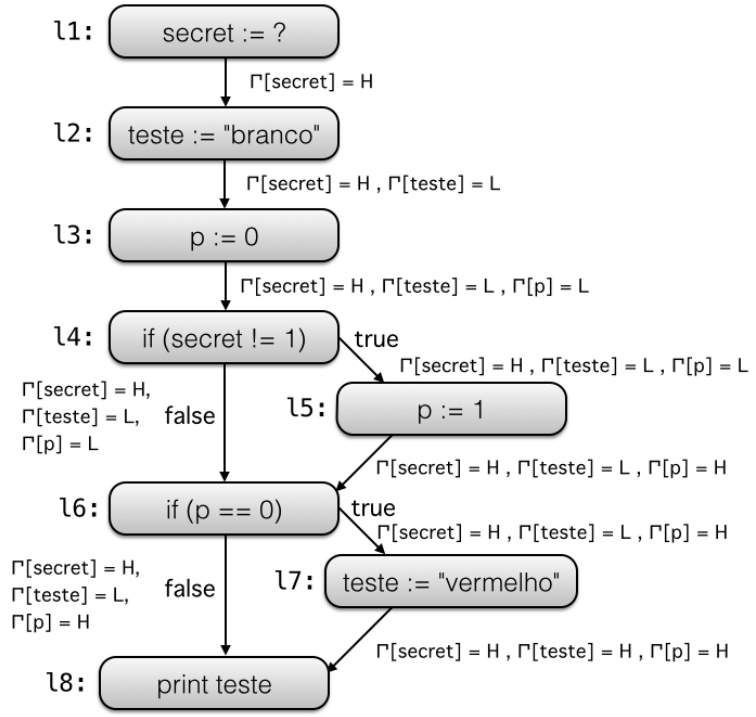
3.4.1 O Sistema de Tipos de Hunt e Sands

Existem diversas análises estáticas de fluxo de informação. Dentre elas encontra-se o sistema de tipos de Hunt e Sands (HUNT; SANDS, 2006).

Em 2006, Hunt e Sands projetaram um sistema de tipos que determina, estaticamente, se um programa contém vulnerabilidades de vazamento de informação (HUNT; SANDS, 2006). Este sistema de tipos é expressivo o suficiente para detectar vazamentos por fluxos implícitos de informação. Em termos simples, esse sistema associa a cada valor usado no programa um tipo, normalmente H para dados de alta segurança, e L para dados de baixa. Se informação de tipo H pode ser usada em alguma função que um atacante consegue observar, então o programa não passa na verificação de tipos.

As regras presentes nesse sistema de tipos foram adaptadas pelo programa Flow Tracking.

Figura 4 – Resultado da aplicação do sistema de tipos de Hunt e Sands no exemplo da figura 3.



Fonte: o autor.

3.4.2 Monitores puramente dinâmicos são inconsistentes

Análises dinâmicas podem ser implementadas por monitores. Um monitor M acoplado a um programa P é um meta-programa que registra todas as mudanças de estado causadas por P. Análises dinâmicas são mais precisas que abordagens estáticas, porém, sofrem de uma séria desvantagem: a inconsistência. Diz-se que uma análise é inconsistente quando ela permite a ocorrência de falsos negativos. Um falso negativo acontece quando o analisador dinâmico falha em abortar um programa que deixa escapar informação para o adversário. Um monitor é chamado puramente dinâmico quando ele pode somente registrar mudanças de estado que aconteceram durante a execução do programa. Existe um importante resultado em teoria da informação que diz que qualquer monitor puramente dinâmico é inerentemente falho devido à presença de fluxos implícitos de informação (RUSSO & SABELFELD, 2010).

Russo e Sabelfeld mostraram, em 2010, que monitores puramente dinâmicos não são poderosos o suficiente para capturar fluxos implícitos. Existem, por outro lado, análises estáticas, como o sistema de tipos proposto por Hunt e Sands (HUNT; SANDS, 2006), que podem fazê-lo.

3.5 Flow Tracking

O programa Flow Tracking desenvolvido por Silva (SILVA; PEREIRA; OLIVEIRA, 2013) é uma implementação do sistema de tipos de Hunt e Sands na forma de um analisador estático para o compilador LLVM. Esta implementação é usada para detectar vazamento de endereços em programas. Ela possui licença pública e está disponível para *download*.

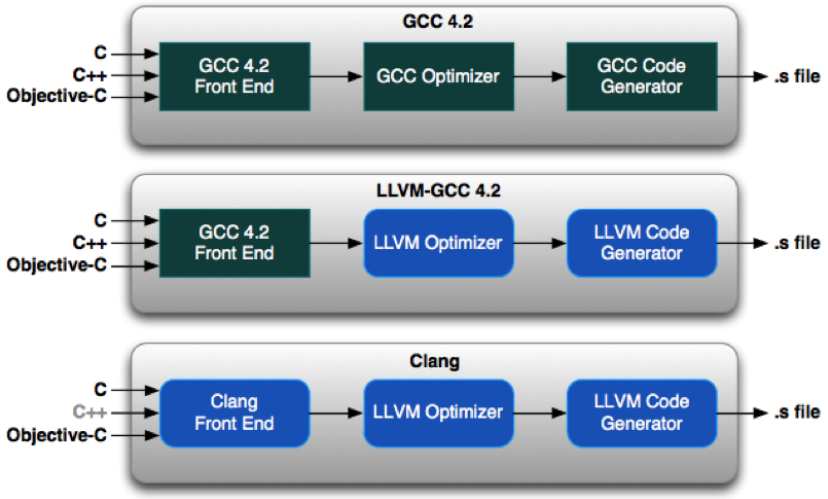
3.5.1 LLVM

Low Level Virtual Machine (LLVM) é um compilador de qualidade industrial, mantido pela empresa Apple Inc, e usado extensivamente na academia, como plataforma de pesquisa. Ele é escrito em C++ e sua arquitetura permitiu a expansão para outras linguagens posteriormente, incluindo Objective-C, Fortran, Ada, Haskell, bytecode Java, Python, Ruby, entre outras.

O LLVM provê diversas bibliotecas que podem ser usadas em diversos momentos da compilação. Com ele é possível otimizar a geração de código, por exemplo. O LLVM é compatível com o GCC (GNU Compiler Collection).

Sua arquitetura é independente da linguagem, conjunto de instruções ou sistema operacional. Cada instrução é definida em uma forma padronizada, permitindo a análise de dependência da árvore de execução do código. A infraestrutura fornece alguns tipos básicos, como ponteiros e estruturas.

Figura 5 – Fluxo de compilação do LLVM.



Fonte: (www.llvm.org, 2014)

Um dos fatores que diferencia o LLVM de outros sistemas é a representação intermediária (IR) de código. Essa deve ser suficientemente baixo nível para permitir a otimização nas fases iniciais da compilação e, suficientemente alto nivel para suportar as

otimizações agressivas feitas nos tempos de ligação e pós-ligação. (MELO; FURTADO, 2008)

A representação intermediária (IR) de código, também chamada de linguagem *assembly* LLVM, é uma representação baseada no Formato de Atribuição Estática Única (SSA) que facilita a análise estática de código. Esta linguagem *assembly* provê operações de baixo nível, flexibilidade e capacidade de representar linguagens de alto nível com clareza. É a IR usada em todas as fases de compilação do LLVM.

3.5.1.1 Exemplo de uma representação intermediária de código de um programa

Abaixo se encontra um programa simples escrito na linguagem C. Neste programa foi criada uma variável inteira *i* inicializada com o valor 0. Na linha imediatamente abaixo essa variável é incrementada em 1.

Código 3.1 – Programa *teste.c*.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main() {
5     int i = 0;
6     i = i + 1;
7     return 0;
8 }
```

Através das ferramentas presentes no compilador LLVM foi gerada a representação do programa acima na linguagem *assembly* LLVM. Essa linguagem possui diversas instruções como por exemplo *alloca*, *store*, *add* e *load*. Na primeira linha tem-se a definição da função *main*, que retorna um inteiro de 32 bits. Nas linhas 2 e 3 são alocados espaços em memória para armazenar inteiros utilizando a instrução *alloca*. Nas linhas 4 e 5 são atribuídos valores a estes espaços de memória utilizando a instrução *store*. Na linha 6 o valor de *i* foi recuperado. Na linha 7 ele foi incrementado em 1 utilizando a função *add*. Por fim, na linha 8 o valor é gravado novamente na memória utilizando a instrução *store*.

Código 3.2 – Código em linguagem *assembly* LLVM do programa *teste.c*

```
1 define i32 @main() #0 {
2   %1 = alloca i32, align 4
3   %i = alloca i32, align 4
4   store i32 0, i32* %1
5   store i32 0, i32* %i, align 4
6   %2 = load i32* %i, align 4
7   %3 = add nsw i32 %2, 1
```

```
8   store i32 %3, i32* %i, align 4
9   ret i32 0
10 }
```

A linguagem *assembly* LLVM é muito bem documentada e poderosa. Através da manipulação de seus *bitcodes* é possível fazer análise de códigos e otimizações.

3.5.2 Formato de Atribuição Estática Única - SSA

SSA é uma técnica de otimização para compiladores, que beneficia muitos algoritmos de otimização. Ela foi concebida na década de 80 nos laboratório de pesquisa da IBM, com a finalidade de simplificar o fluxo de controle e tornar mais fácil a adoção de técnicas que são beneficiadas pela declaracão única de uma variável (SPAKI, 2009).

Nessa representação cada variável **tem apenas uma definição**. Isto permite uma melhor análise do código, tornando mais prático e eficiente aplicar otimizações. Com a SSA é mais fácil criar um grafo de fluxo de controle ou controle de dependência.

O exemplo abaixo mostra atribuição de valores à variáveis:

Código 3.3 – Atribuição de valores a variáveis.

```
X = 5;
X = 10;
Y = X;
```

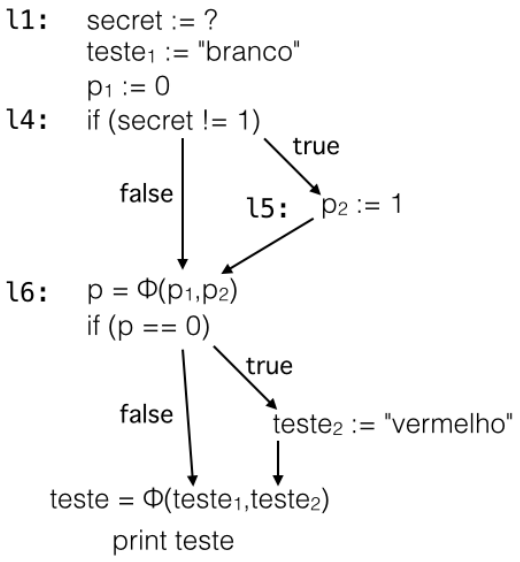
Como é possível ver, a primeira linha é dispensável pois o valor de X sempre é sobreescrito. Para um compilador identificar esta otimização teria que criar um grafo de fluxo e analisá-lo. O que o SSA faz é individualizar as variáveis e seu uso, fazendo com que a identificação do fluxo e tarefas de otimização sejam mais simples. No caso abaixo, um algoritmo de "eliminação de código morto" (SPAKI, 2009) se beneficiaria ao identificar a inatividade de X_1 :

Código 3.4 – Atribuição de valores a variáveis utilizando o SSA.

```
X1 = 5;
X2 = 10;
Y = X2;
```

A figura 6 mostra o exemplo da 4 convertido para a representação SSA. Variáveis com várias definições foram renomeadas. Também existem funções ϕ unindo definições em um único nome. Estas funções funcionam como multiplexadores. Por exemplo, se o fluxo de execução chega em l_6 vindo de l_5 então a função $\phi(p_1, p_2)$ copia a variável p_2 para p.

Figura 6 – Exemplo da figura 4 convertido para o formato SSA.



Fonte: o autor.

O Flow Tracking usa a representação SSA através do compilador LLVM para analisar o código.

3.5.3 A implementação concreta do sistema de tipagem

No programa Flow Tracking, foi utilizada uma implementação que transforma o sistema de tipagem de Hunt e Sands em um problema de busca em grafos. Assim, o grafo $G = (V, D \cup C, O)$ de dependências de um programa é definido seguindo as regras:

- Existe um vértice $n_v \in V$ para cada variável v presente no programa;
- Existe um vértice $n_o \in O$ para cada ocorrência de uma função print no programa;
- Tem-se uma aresta $(n_u, n_v) \in D$ para cada dependência de dados entre duas variáveis, u e v;
- Tem-se uma aresta $(n_v, n_o) \in D$ se v é uma variável usada em uma função print de o;
- Tem-se uma aresta $(n_p, n_v) \in C$ entre p (o predicado que controla um desvio condicional), e toda variável v definida na região de influência deste desvio;
- Tem-se uma aresta $(n_p, n_o) \in C$ entre p (o predicado que controla um desvio condicional) e cada função print de o usada na região de influência de p.

Essa definição de grafo de dependências demanda a noção da região de influência de um predicado. A região de influência de um predicado p, que controla um desvio condicional

posicionado em um rótulo l_p do programa é um conjunto de rótulos. Esses rótulos incluem l_p, o seu pós-dominador l_d, e todos os demais rótulos entre esses dois nós. Diz-se que um rótulo l_d pós-domina um rótulo l_p se qualquer caminho no programa, entre l_p e sua última instrução, passa por l_d.

3.5.4 A necessidade de variáveis com estados invariantes

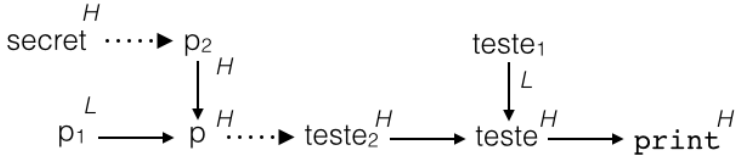
É necessário fazer que o grafo de dependências seja uma representação adequada do problema de vazamento de informação. Para tanto, precisa-se garantir que o estado de uma variável, isto é, seu tipo, seja invariante em qualquer parte do texto do programa. Essa propriedade não existe, em geral. Por exemplo, no programa visto na figura 4, tem-se que $\Gamma[teste] = L$ entre $l2$ e $l3$. Porém, $\Gamma[teste] = H$ entre $l7$ e $l8$. Fenômeno similar ocorre com a variável p, pois ela é definida com o tipo L em $l4$, e com o tipo H em $l6$. Casualmente, existe uma representação de código, popular entre compiladores modernos, que permite lidar com essa dificuldade.

Para se obter estados invariantes para cada nome de variável definido no programa, pode-se utilizar a representação intermediária de código chamada *Formato de Atribuição Estática Única*, ou SSA. Essa forma de representar programas, inventada no final da década de 80 (CYTRON et al., 1989), é hoje chave para dezenas de otimizações de código. Esse formato possui a propriedade de que cada nome de variável é definido em somente um ponto no texto do programa.

3.5.5 Grafo de fluxo

A figura 7 mostra o grafo de fluxo criado para o programa da figura 4. Arestas representando fluxos explícitos são sólidas, enquanto arestas representando fluxos implícitos são tracejadas. Em cada variável é mostrado seu tipo, L ou H. O tipo da variável é H se ela for alcançável a partir de *secret*. Desta forma, o grafo descreve um programa vulnerável se existe um caminho entre a variável *secret* e alguma função de saída como o *print*.

Figura 7 – Grafo de fluxo de informação criado para o programa visto na figura 4. O tipo de variável é mostrado ao lado do vértice que a representa.



Fonte: o autor.

O Flow Tracking cria grafos de fluxos explícitos e implícitos do programa que está sendo analisado. Ele considera "sources" como funções que alocam espaço em memória como *malloc* e *calloc*. E considera "saídas" como quaisquer funções que escrevem dados

em canais públicos como *printf* por exemplo. Assim, a ferramenta analisa os grafos de fluxos do programa verificando para cada "source", se existe um caminho possível para se chegar a alguma "saída".

Um caminho é uma sequência de vértices no grafo de fluxos. Existindo um possível caminho de um "source" até uma "saída", se configura um vazamento de informação. O Flow Tracking reporta todos os caminhos possíveis de vazamento de informação. A ferramenta faz esta análise utilizando a busca em profundidade.

4 METODOLOGIA

O desenvolvimento do projeto envolve um estudo profundo das ferramentas e tecnologias utilizadas. Também envolve uma base teórica de grafos, análise de algoritmos, fluxo de informação e otimização de programas. O trabalho então foi dividido em tópicos separados por nível de dificuldade e assunto. Ao decorrer das etapas foi adiquirido mais embasamento que permitiu contemplar os objetivos definidos. Abaixo, algumas atividades importantes para a conclusão do trabalho são listadas:

1. Estudo de trabalhos relacionados com o tema proposto. Assim, procura-se a inserção no assunto buscando entender as soluções propostas pela academia para os problemas pesquisados.

2. Entendimento do problema de vazamento de informações em programas por canais públicos. Para tal, é utilizada a análise estática.

3. Estudo dos fluxos implícitos e explícitos de informação. Para isto, é necessário obter uma representação intermediária do programa, conhecida como grafo de dependências, onde são mostradas as dependências de dados e controle entre variáveis do programa.

4. Entendimento do compilador LLVM, além de suas ferramentas e API. Estudo dos conceitos da área de otimização de código em compiladores como o Formato de Atribuição Estática Única - SSA, por exemplo.

5. Estudo da ferramenta Flow Tracking bem como a execução de experimentos com diversas entradas analisando os artefatos gerados.

6. **Modificação da ferramenta Flow Tracking**, permitindo customizar quais "sources" e "saídas" serão analisados pela ferramenta. Para isto, é proposto a utilização de um arquivo XML *(eXtensible Markup Language)* de entrada para definição destes parâmetros. A ferramenta modificada então interpretará este arquivo, e passará as informações necessárias para o Flow Tracking fazer seu trabalho.

7. Execução de experimentos. É importante testar a ferramenta Flow Tracking modificada com diferentes parâmetros relevantes. A execução de experimentos é necessária para comprovar ou refutar as metodologias utilizadas. Esta fase é fundamental pois neste momento são feitos testes com problemas relevantes e os resultados são gerados. Deve-se validar os conhecimentos adquiridos e o resultado apresentado.

8. Análise dos resultados obtidos. Etapa importante para possíveis trabalhos futuros. Os resultados obtidos definem se o projeto alcançou os objetivos propostos. Deve-se

analisar se os resultados foram proveitosos, satisfatórios e se atendem as espectativas dos envolvidos.

5 RESULTADOS

A implementação do Flow Tracking utilizada neste trabalho foi construída na forma de um analisador estático para o compilador LLVM, versão 3.3, disponível em Março de 2014. Todos os testes foram feitos em uma máquina Intel Core i5, com 4 GB de memória RAM, e clock de 2.1 GHz. O sistema operacional usado foi Mac OS X, versão 10.9.2. A ferramenta analisa código-fonte de programas escritos em C e C++ e informa se existem possíveis vazamentos de endereço.

O objetivo deste capítulo, é mostrar todos os passos realizados para o desenvolvimento do trabalho. É mostrado um estudo detalhado do Flow Tracking bem como a modificação da ferramenta onde foram adicionadas algumas funcionalidades.

O Flow Tracking considera como possíveis fontes de vazamento de endereço funções de libc que alocam espaço em memória, tais como *malloc*, *calloc* e *realloc*. E considera "saída" quaisquer funções que escrevem dados em canais públicos como *printf*, *putc* e *fputc*.

5.1 Utilizando o Flow Tracking

Para os testes é necessário baixar o código-fonte do Flow Tracking e compilá-lo. A ferramenta é um *pass* para o compilador LLVM. Por isso deve-se instalar o compilador no ambiente de trabalho. No site do projeto (www.llvm.org) é possível baixar a versão para o sistema operacional desejado. Um *pass* é uma importante parte da ferramenta LLVM, porque é onde as partes mais interessantes do LLVM são feitas. Os *passes* são implementados para realizar transformações, otimizações e analisar programas que são compilados pelo LLVM. Com eles é possível manipular facilmente o código do programa através de métodos da API do LLVM, podendo com isso modificar, auditar e otimizar o código que está sendo compilado.

Após a instalação do compilador LLVM é necessário compilar o *pass* Flow Tracking. Detalhes sobre como compilar o *pass* são descritos na página do projeto disponível em <https://code.google.com/p/ecosoc/wiki/flowtracking>.

Para o Flow Tracking analisar o programa *teste1.c* no diretório corrente deve-se executar o shell script que invoca o *pass*:

```
dotgen.bin teste1.c
```

O conteúdo do shell scrit *dotgen.bin* é mostrado na figura 8. Na linha 10, utilizando a ferramenta *Clang* que faz parte do ecossistema LLVM, com o parâmetro *-emit-llvm* é gerada uma representação intermediária (IR) do código-fonte na linguagem *assembly*

LLVM. Essa linguagem permite ao LLVM prover uma interface para transformações e análises de programas em tempo de compilação. O *Clang* é o *frontend* do compilador LLVM. Ele usa o LLVM como seu *backend*.

Figura 8 – Conteúdo do shellscript *dotgen.bin*.

```
1 #!/bin/bash
2
3 file_name=$1
4 base_name=$(basename $1 .c)
5 btcd_name="$base_name.bc"
6 dot_name="$base_name.dot"
7 opt_name="$base_name.rbc"
8 pdf_name="$base_name.pdf"
9
10 clang -emit-llvm -c -g $1 -o $btcd_name
11
12 #clang -emit-llvm -c $1 -o $btcd_name
13
14 opt -instnamer -mem2reg $btcd_name > $opt_name
15
16 opt -disable-output -load PADriver.so -load AliasSets.so -load DepGraph.so -load flowTracking.so -flowTracking $opt_name
17
18 mv /tmp/fullGraph.dot $dot_name
19
20 dot -Tpdf $dot_name -o $pdf_name
21
22 opt -disable-output -dot-cfg $opt_name
```

Fonte: (Flow Tracking Software, 2013)

Após gerado a representação intermediária em *assembly* LLVM do código-fonte, é utilizada a ferramenta *opt* do LLVM para gerar, através deste arquivo, um outro arquivo em um formato mais otimizado que possa ser lido pelo *pass* Flow Tracking. Após esse comando, que está na linha 14, o arquivo irá utilizar o Formato de Atribuição Estática Única (SSA), que é necessário para a análise proposta.

Na linha 16, o *pass* Flow Tracking é chamado através do comando *opt*. Após a execução, é mostrado se existem caminhos para vazamento de informação. Se existirem, serão criados arquivos *.dot* contendo os caminhos existentes, representados por grafos de dependência, no qual pode ocorrer vazamento de informação.

As últimas linhas do arquivo *dotgen.bin* são para a manipulação dos arquivos *.dot* gerados. Para visualizá-los, pode-se utilizar ferramentas de visualização de grafos, como a *Graphviz*, utilizada neste trabalho.

Na figura 9 é possível ver um exemplo onde foram encontrados dois “sources” e uma saída. Também é mostrado que existem dois caminhos possíveis dos “sources” até a saída.

Figura 9 – Exemplo de saída da execução do Flow Tracking.

```
MacBookAir-Leo:monografia leonardoribeiro$ ./dotgen-mac.bin programas/teste8.c
******** Flow Tracking Summary ***********

Sources of address 2
Sinks of address (public channels) 1
Number of address leak paths 2

Writing 'cfg.main.dot'...
MacBookAir-Leo:monografia leonardoribeiro$
```

Fonte: o autor.

5.2 Detectando vazamento de informação com o Flow Tracking

5.2.1 Fluxos explícitos

Nos fluxos explícitos, também chamados de dependências de dados, as informações fluem via movimentação de bits de uma posição da memória para outra. O programa em C mostrado na figura 10 manipula porções de memória. Ocorre fluxo explícito de informação através das variáveis. O Flow Tracking foi utilizado para analisá-lo.

Na linha 13 a função *malloc* aloca 16 bytes de memória e devolve o endereço do primeiro bloco para a variável x, que é um ponteiro para inteiro. Na linha 14, a variável y, que também é um ponteiro para inteiro, recebe o endereço do próximo bloco de memória. Na linha 15, o endereço de y é incrementando mais uma vez e na linha 16 a variável z aponta para este mesmo endereço. Por fim, na linha 17, este endereço é impresso na tela.

Figura 10 – Programa com vazamento de informação por fluxo explícito.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 /*
5 * Fluxo explícito
6 * O endereço retornado pela função malloc pode chegar até a função printf
7 * por meio de fluxo explícito.
8 */
9
10 int main (int argc, char **argv) {
11         int *x, *y, *z;
12
13         x = (int *) malloc (16);
14         y = x+1;
15         y++;
16         z = y;
17         printf ("%p", z);
18 }
```

Fonte: o autor.

Existe um vazamento de informação por fluxo explícito neste programa que começa na linha 13, quando a variável x recebe um endereço. Através deste fluxo, informações de endereço de variáveis na memória trafegam até chegar em um canal público, no caso a função *printf* da linha 17. Neste exemplo, ocorre um vazamento de informação sigilosa,

que deveria estar protegida. Informações sigilosas como estas podem ser usadas como auxílio em um ataque ao sistema computacional.

A figura 11 mostra o resultado da execução do Flow Tracking sobre o programa da figura 10.

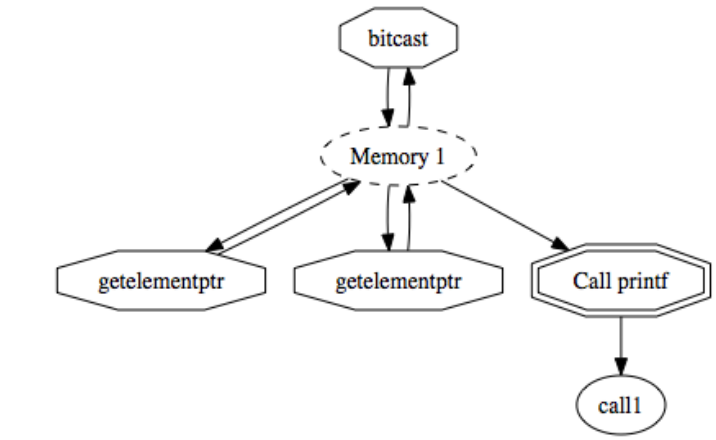
Figura 11 – Saída do Flow Tracking para o programa da figura 10.

```
MacBookAir-Leo:monografia leonardoribeiro$ ./dotgen-mac.bin programas/exemplo1.c
********* Flow Tracking Summary ************

Sources of address 1
Sinks of address (public channels) 1
Number of address leak paths 1

Writing 'cfg.main.dot'...
MacBookAir-Leo:monografia leonardoribeiro$
```

Fonte: o autor.

Como mostrado na figura 11, foi detectado o vazamanto de informação. A ferramenta reportou que existe um "source" e um canal público no algoritmo analisado. E também reporta que existe um caminho entre os dois. O Flow Tracking também gera os grafos de dependência. A figura 12 mostra o grafo gerado por esta execução. É possível ver que existe um caminho entre um endereço de memória e um canal público, no caso a função *printf*.

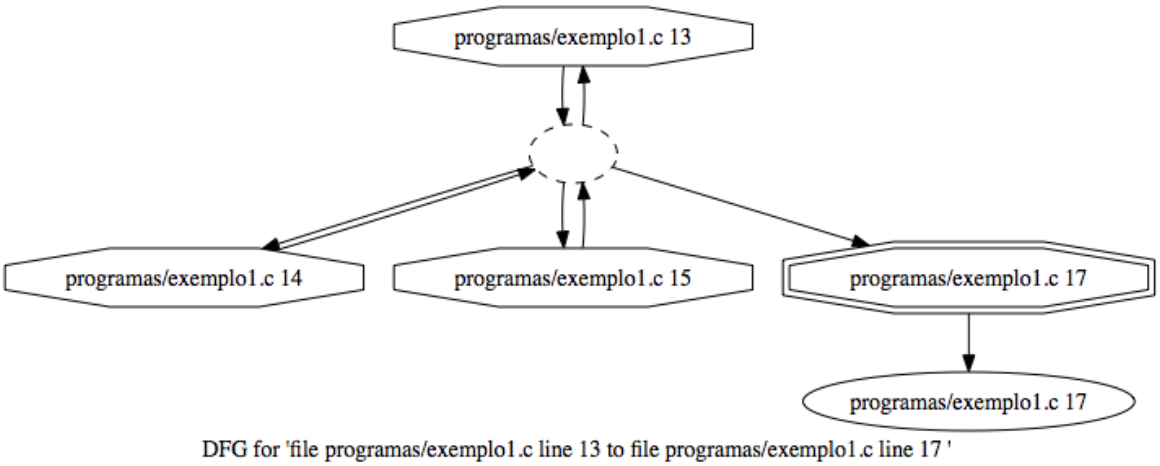
Figura 12 – Grafo de dependência para o programa da figura 10, considerando as chamadas em *assembly* LLVM.



Fonte: o autor.

A figura 13 mostra o grafo de dependência entre as linhas do programa. O grafo mostra que a partir da linha 13, onde um endereço de memória é alocado, é possível chegar até o *printf* da linha 17 através de uma porção da memória. As linhas 14 e 15 também manipulam esta porção de memória que é representada pela elipse tracejada no meio da figura.

Figura 13 – Grafo de dependência para o programa da figura 10, considerando as linhas do código-fonte.



Fonte: o autor.

5.2.2 Fluxos implícitos

Os fluxos implícitos ocorrem por meio das dependências de controle. Dependências de controle ocorrem por meio de estruturas condicionais. Uma informação flui implicitamente de um predicado p, que controle um teste condicional, para toda variável atribuída no escopo dessa condição.

Na linha 17 da figura 14 existe um teste condicional com a variável do tipo ponteiro para inteiro k. O teste verifica se o endereço da variável é maior que zero. De posse do código-fonte deste programa e de saídas resultantes, um atacante pode ter informações acerca do valor dos endereços de memória das variáveis, devido a este fluxo implícito de informação. O Flow Tracking consegue detectar este vazamento implícito de informação. O resultado da execução do Flow Tracking para a análise deste programa se encontra na figura 15. É possível ver que o *pass* reporta duas fontes de endereço e duas saídas. Ele considera a função *malloc* da linha 12 como uma fonte, e também considera o teste condicional com o ponteiro para inteiro k da linha 17. Para as saídas, são considerados os dois *printf's* do programa. Como a partir das duas fontes de endereço é possível chegar as duas saídas públicas, o Flow Tracking retorna 4 caminhos possíveis de vazamento de endereços.

A figura 16 mostra um dos grafos gerados por esta execução. É possível ver que existe um caminho entre um endereço de memória e um dos dois canais públicos através de fluxos implícitos. Setas com linha tracejada representam fluxo implícitos.

A figura 17 mostra um dos grafos de dependência gerados destacando as dependências entre as linhas do programa. A partir da linha 17, onde o endereço da variável k é usado no teste condicional, é possível chegar até o *printf* da linha 18. A elipse tracejada

Figura 14 – Programa com vazamento de informação por fluxo implícito.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 /*
5  * Fluxo implícito
6  * O endereço retornado pela função malloc pode ser inferido através da função printf por meio
7  * de fluxo implícito, analisando o predicado (int)k > 0
8  */
9
10 int main(int argc, char **argv) {
11     int *i, *j, *k, cond;
12     i = (int *) malloc(8);
13     j = i;
14     j++;
15     k = j;
16
17     if ((int) k > 0)
18         printf("1");
19     else
20         printf ("2");
21     return 0;
22 }
```

Fonte: o autor.

Figura 15 – Saída do Flow Tracking para o programa da figura 14.

```
MacBookAir-Leo:monografia leonardoribeiro$ ./dotgen-mac.bin exemplo2.c
********* Flow Tracking Summary ************

Sources of address 2
Sinks of address (public channels) 2
Number of address leak paths 4

Writing 'cfg.main.dot'...
MacBookAir-Leo:monografia leonardoribeiro$
```

Fonte: o autor.

no meio da figura representa uma porção de memória. A seta tracejada da linha 17 para a linha 18 significa o fluxo implícito do predicado do teste condicional para seu escopo.

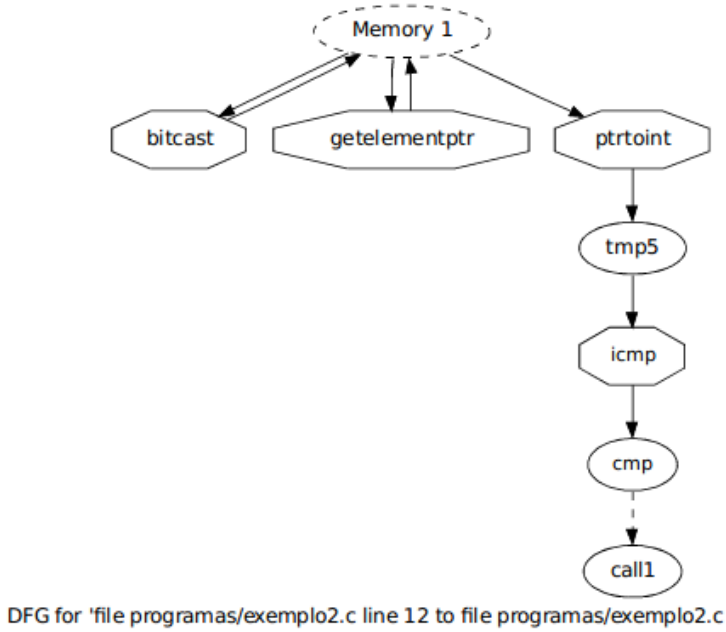
5.3 Análises feitas com o Flow Tracking

Silva executou o Flow Tracking para analisar os *benchmarks* presentes em SPEC CPU 2006 (SILVA; PEREIRA; OLIVEIRA, 2013). Os números gerados dão uma ideia sobre a escala das buscas feitas pra detectar vazamento de informação. Os grafos que levam em consideração somente fluxos explícitos, isto é, dependência de dados, são muito espalhados tendo cerca de 1.4 arestas para cada vértice. Por outro lado, os grafos que levam em consideração fluxos implícitos de informações são mais densos, tendo em média 40 arestas para cada vértice.

Pode-se observar na figura 18 que existem poucas fontes, isto é, operações que geram endereços, nos programas. As funções de saída aparecem normalmente em maior número. Tanto fontes quanto saídas correspondem a apenas 1% dos vértices dos grafos de fluxo dos programas.

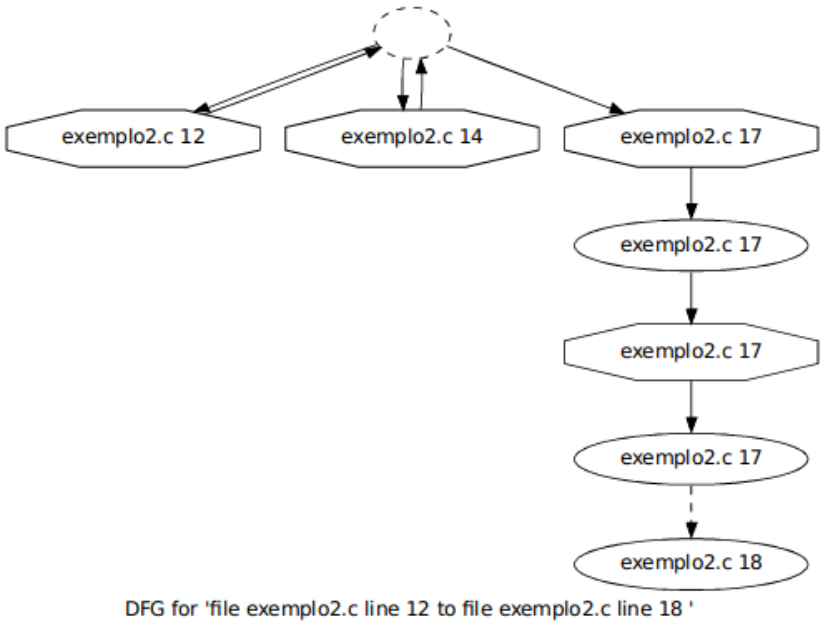
A figura 19 mostra a quantidade de avisos reportados, considerando ou não os

Figura 16 – Um dos grafos de dependência para o programa da figura 14, considerando as chamadas em *assembly* LLVM.



Fonte: o autor.

Figura 17 – Um dos grafos de dependência para o programa da figura 13, considerando as linhas do código-fonte.



Fonte: o autor.

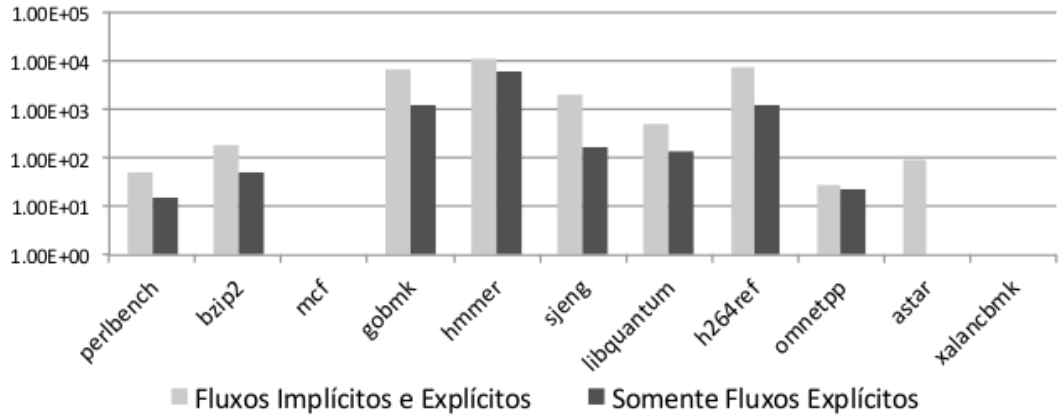
fluxos implícitos de informação. É possível perceber que análises que não consideram fluxos implícitos podem deixar de reportar vulnerabilidades reais.

Figura 18 – Estatísticas relacionadas aos grafos de dependência construídos para cada *benchmark*.

benchmark	Instruções	Vértices	Arestas de Dados	Arestas de Controle	Total de Arestas	Fontes	Saídas
mcf	2.556	$3,8K$	$6K$	$10,1K$	$16,1K$	4	26
libquantum	6.435	$9,8K$	$15,1K$	$27,8K$	$42,9K$	19	50
astar	8.646	$13,2K$	$20,5K$	$4,5K$	$6,5K$	22	27
omnetpp	91.146	$156,4K$	$239,2K$	$3,4M$	$3,6M$	3	65
hmmer	67.528	$107,4K$	$150K$	$580,4K$	$730,4K$	45	451
xalancbmk	588.502	$990,2K$	$1,4M$	$2,1M$	$3,5M$	0	14
sjeng	30.012	$43,8K$	$62,6K$	$1,3M$	$1,4M$	10	224
bzip2	17.324	$27,7K$	$41,5K$	$436,3K$	$477,9K$	5	84
perlbench	283.594	$434,9K$	$607,5K$	$15,5M$	$16,1M$	8	10
gobmk	144.267	$227,6K$	$332,6K$	$2,1M$	$2,4M$	22	471
h264ref	143.804	$234,7K$	$340K$	$1,1M$	$1,4M$	169	220

Fonte: (SILVA; PEREIRA; OLIVEIRA, 2013)

Figura 19 – Número de avisos de vazamento de informação. O gráfico está usando escala logarítmica.



Fonte: (SILVA; PEREIRA; OLIVEIRA, 2013)

5.4 Modificação do algoritmo do Flow Tracking

O *pass* Flow Tracking está implementado para detectar vazamento de endereços. Isto quer dizer que ele considera como fontes as funções de libc que alocam espaço em memória. Alguns exemplos destas funções são *malloc*, *calloc* e *realloc*. E considera como saída funções que escrevem dados em canais públicos, como *printf*, *putc* e *fputc*.

O Flow Tracking original foi modificado para utilizar um arquivo XML *(eXtensible Markup Language)* onde é possível definir quais as fontes e saídas se deseja rastrear. Desta forma, é possível analisar uma gama maior de problemas utilizando dependência de dados e de controle. Por exemplo, um desenvolvedor codificou um programa que manipula diversos arquivos sigilosos em disco utilizando a função *fopen*, mas ele deseja que o usuário que executar o programa não saiba qualquer informação relativa a estas operações. Assim,

pode-se definir como fonte a função *fopen*, e definir como saída a função *printf*. Desta forma, se existisse algum fluxo que saísse de *fopen* e chegasse a *printf*, ele seria reportado pela ferramenta.

Para esta modificação, foi necessário alterar as regras definidas na seção 3.5.3. Estas regras são usadas pelo Flow Tracking para a criação dos grafos de dependência. Assim o grafo de $G = (V, D \cup C, O)$ de dependências de um programa é definido, para o Flow Tracking modificado, seguindo as regras:

- Existe um vértice $n_v \in V$ para cada variável v presente no programa que seja usada por uma função f que manipula a memória e que foi escolhida como fonte;
- Existe um vértice $n_o \in O$ para cada ocorrência de uma função de saída escolhida;
- Tem-se uma aresta $(n_u, n_v) \in D$ para cada dependência de dados entre duas variáveis, u e v, presentes nos caminhos entre as fontes e as saídas;
- Tem-se uma aresta $(n_v, n_o) \in D$ se v é uma variável usada em uma função escolhida que utiliza o;
- Tem-se uma aresta $(n_p, n_v) \in C$ entre p (o predicado que controla um desvio condicional), e toda variável v definida na região de influência deste desvio;
- Tem-se uma aresta $(n_p, n_o) \in C$ entre p (o predicado que controla um desvio condicional) e cada função de saída escolhida que utiliza o usada na região de influência de p.

Com essa modificação, é possível ter flexibilidade para analisar programas utilizando quaisquer funções como fonte e quaisquer funções como saída.

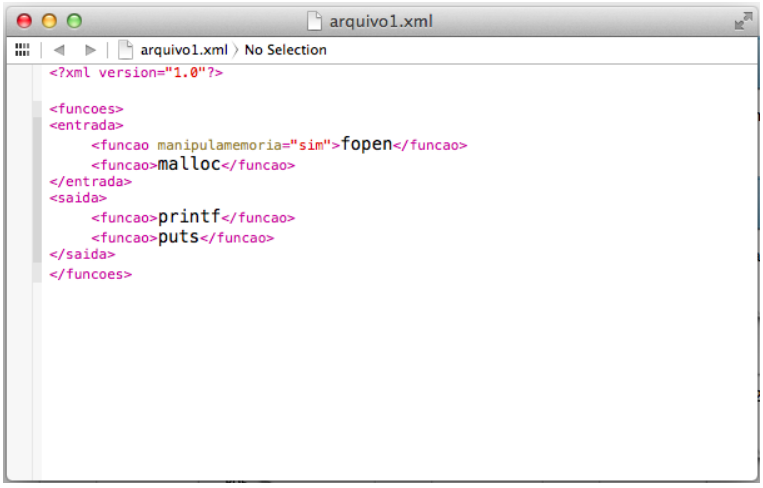
Para analisar um arquivo XML é necessário a utilização de um *parser*. Um *parser*, nada mais é que um código que analisa uma sequência de strings. Neste caso, ele percorre um arquivo XML e analisa sua estrutura.

5.4.1 Formato do arquivo XML

Foi proposta uma estrutura de arquivo XML simples. A tag *<funcoes>* foi definida para conter todas as funções que farão parte da análise. Dentro desta tag, encontram-se tags *<fontes>*, que são utilizadas para informar todas as funções que são "sources" na análise do Flow Tracking. Dentro da tag *<funcoes>* também encontram-se tags *<saidas>* que contêm as funções que são saídas para a análise do Flow Tracking. As funções do programa que se deseja analisar devem ser declaradas no arquivo XML, entre as tags *<funcao></funcao>*. Desta forma, é possível informar ao Flow Tracking quaisquer funções, tanto como fontes quanto como saídas, para a análise de fluxo de informação.

Quando uma função definida como entrada manipula a memória através dos seus parâmetros, deve-se informar isto ao Flow Tracking adicionando um parâmetro na tag *<funcao>* de uma entrada. Este parâmetro deve ser setado da seguinte forma: *<funcao manipulamemoria="sim" >*. A figura 20 mostra um exemplo de um arquivo XML que pode ser lido pelo *parser* desenvolvido.

Figura 20 – Exemplo da estrutura de um arquivo XML que pode ser lido pelo Flow Tracking modificado.



Fonte: o autor.

5.4.2 TinyXML

O TinyXML é um *parser* escrito em C++ que pode ser facilmente integrado a diversos programas, inclusive aos *passes* do LLVM. Ele lê um arquivo XML e cria objetos que representam os elementos presentes neste arquivo. A modificação do Flow Tracking utilizou o TinyXML para manipular o arquivo XML. Uma classe, que usa o TinyXML, foi criada para tratar os dados do arquivo que é passado por parâmetro para o Flow Tracking modificado. O TinyXML pode ser baixado em <http://sourceforge.net/projects/tinyxml/ >.

5.4.3 Flow Tracking modificado

A classe *parserXML* foi desenvolvida para ler o arquivo XML passado por parâmetro na hora da execução do Flow Tracking.

Código 5.1 – Estrutura da classe *parserXML*.

```
1 #ifndef PARSERXML_H_
2 #define PARSERXML_H_
3
4 #include <iostream>
5 #include "tinyxml2.h"
```

```
6  #include <vector>
7
8  namespace parsersXML {
9  class parserXML {
10 public:
11   class funcaoFonte {
12   public:
13     std::string getNome();
14     void setNome(std::string nome);
15     bool usaMemoria();
16     void setUsaMemoria(bool u);
17     funcaoFonte();
18   private:
19     std::string nomeFuncao;
20     bool variavelUsaMemoria;
21
22   };
23
24   bool setFile(std::string fileName);
25   std::vector<funcaoFonte> getFontes();
26   std::vector<std::string> getSaidas();
27   parserXML(std::string fileName);
28   parserXML();
29   virtual ~parserXML();
30
31 private:
32   std::vector<funcaoFonte> funcoes_fonte;
33   std::vector<std::string> funcoes_saida;
34   tinyxml2::XMLDocument doc;
35
36 };
37 }
38 #endif /* PARSERXML_H_ */
```

A classe é simples e possui três métodos importantes. O primeiro, *parserXML::setFile*, é usado para setar o caminho no qual o arquivo XML se encontra. Os dois outros métodos chamados *parserXML::getFontes()* e *parserXML::getSaidas()* retornam o nome das funções que são fontes e saídas informadas no arquivo XML, respectivamente. O retorno das funções de entrada é do tipo *vector<parsersXML::parserXML::funcaoFonte>* e o retorno das

funções de saída é do tipo *vector<string>*. A classe *parserXML::funcaoFonte* contém dois atributos. O primeiro, *nomeFuncao*, contém o nome da função definida para a entrada. O segundo, *variavelUsaMemoria*, armazena a informação que indica se essa função manipula a memória ou não.

O Flow Tracking original foi modificado retirando a parte estática que informava quais funções de libc deveriam ser essas fontes e saídas. Assim, os dois vetores contendo as fontes e saídas são percorridos e os valores recuperados são passados à parte do algoritmo que analisa os fluxos de dados e de controle.

Através desta lógica, a interface do Flow Tracking com o usuário foi automatizada, podendo o mesmo informar no momento da execução, qualquer função que desejar em sua análise.

O arquivo *dotgen.bin* também foi modificado. Agora deve-se passar o arquivo XML como parâmetro para o *pass* Flow Tracking. No exemplo a seguir, que é uma linha que faz parte do arquivo *dotgen.bin* modificado, utilizando o identificador *-xmlfile* foi passado como parâmetro o arquivo *dados-entrada.xml*.

```
opt -disable-output -load PADriver.so -load AliasSets.so
-load flowTracking.so -load ./parserXML.so
-flowTracking teste.bc -xmlfile dados-entrada.xml
```

Através da linha abaixo, no código do Flow Tracking (flowTracking.cpp), é possível referenciar o arquivo passado por parâmetro através da variável *InputFilename*:

Código 5.2 – Código que utilizada a api do LLVM para receber parâmetros.

```
static cl::opt<string> InputFilename("xmlfile",
 cl::Required, cl::desc(
 "Informe um arquivo XML valido."),
cl::value_desc("fileXML"));
```

O pedaço de código seguinte mostra a interação do Flow Tracking modificado com a classe parserXML criada. O conteúdo do arquivo é carregado na variável *xmlparser*, e logo após, os vetores para armazenar as funções que são fontes e saídas são carregados com os valores recuperados do arquivo XML.

Código 5.3 – Parte do código Flow Tracking modificado que utilizada a classe *parserXML*.

```
1 parsersXML::parserXML *xmlparser = new parsersXML::parserXML
     ();
2   if (!xmlparser->setFile(InputFilename)) {
3     errs() << "Nao foi possivel carregar o arquivo XML.";
4     return false;
```

```
5   }
6   std::vector<parsersXML::parserXML::funcaoFonte>
       funcoes_fontes;
7   std::vector<std::string> funcoes_saida;
8   funcoes_fontes = xmlparser->getFontes();
9   funcoes_saida = xmlparser->getSaidas();
```

5.4.4 Resultado de testes com Flow Tracking modificado

Um adversário, com conhecimento sobre o comportamento do programa e tendo o controle das variáveis de entrada, pode inserir dados que explorem alguma vulnerabilidade. Isto pode ser evitado utilizando a análise de fluxo contaminado. Esta análise busca encontrar quais partes do programa são dependentes de entradas que um atacante pode manipular. Análises deste tipo são utilizadas para encontrar vulnerabilidades a ataques do tipo *SQL Injection* (TRIPP et al., 2009). O *SQL Injection* é um ataque no qual código malicioso é passado para um servidor de banco de dados SQL, que o executa. O ataque pode resultar em acesso não autorizado a dados confidenciais, ou destruição de dados críticos.

O Flow Tracking modificado é capaz de analisar ataques de fluxo contaminado, como o *SQL Injection*. Para verificar essa possibilidade, foi definido o arquivo XML abaixo para o Flow Tracking:

Código 5.4 – Arquivo *teste1.xml*.

```
1  <?xml version="1.0"?>
2  <funcoes>
3          <fontes>
4                  <funcao manipulamemoria="sim">scanf</funcao>
5          </fontes>
6
7          <saidas>
8                  <funcao>mysql_query</funcao>
9          </saidas>
10 </funcoes>
```

Será passado como parâmetro para o Flow Tracking modificado o arquivo *teste1.xml*. Neste arquivo foi definido que a função *scanf* é a fonte dos fluxos de dados e controle. Já a função *mysql_query* foi definida como saída. Na linguagem C, a função *scanf* é utilizada para receber dados informados por usuários. Já a função *mysql_query* é utilizada para executar uma *query* em um servidor de banco de dados *MySQL*. Se existir um fluxo que

saia da função *scanf*, que é uma fonte de entrada de dados externos ao programa, e chegue a função *mysql_query*, o Flow Tracking modificado deve alertar o programador que este é um fluxo contaminado que pode ser utilizado para ataques do tipo *SQL Injection*. Um fluxo contaminado desta tipo pode ser encontrado no código 5.5.

Código 5.5 – Programa vulnerável ao ataque *SQL Injection*.

```
1  #include </usr/include/mysql/mysql.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <string.h>
5  int main() {
6     MYSQL *conn;
7     MYSQL_RES *res;
8     MYSQL_ROW row;
9     char *server = "localhost";
10    char *user = "root";
11    char *password = "PASSWORD";
12    char *database = "mysql";
13    conn = mysql_init(NULL);
14
15    /* Connect to database */
16
17    /* send SQL query */
18    char statement[512];
19    char *my_str = (char*)malloc(12 * sizeof(char));
20    scanf("%s",my_str);
21
22    strcpy(statement,"INSERT␣INTO␣table␣VALUES␣('");
23    strcat(statement, my_str);
24    strcat(statement, "')");
25    mysql_query(conn, statement);
26
27    /* close connection */
28    mysql_free_result(res);
29    mysql_close(conn);
30    return 0;
31 }
```

O código 5.5 mostra um programa que se conecta a uma base de dados MySQL utilizando funções presentes na biblioteca *Connector/C* provida pela empresa Oracle.

Este programa executa um comando de *insert* no banco de dados utilizando a função *mysql_connect*. Através da função *scanf*, o usuário passa um parâmetro que é usado no *insert*. Foram omitidas partes do código do programa, como a conexão com a base de dados por exemplo, para simplificar a análise. Utilizando o Flow Tracking modificado para analisar o código, tem-se a saída da figura 21. Foi encontrado um caminho entre a fonte e a saída.

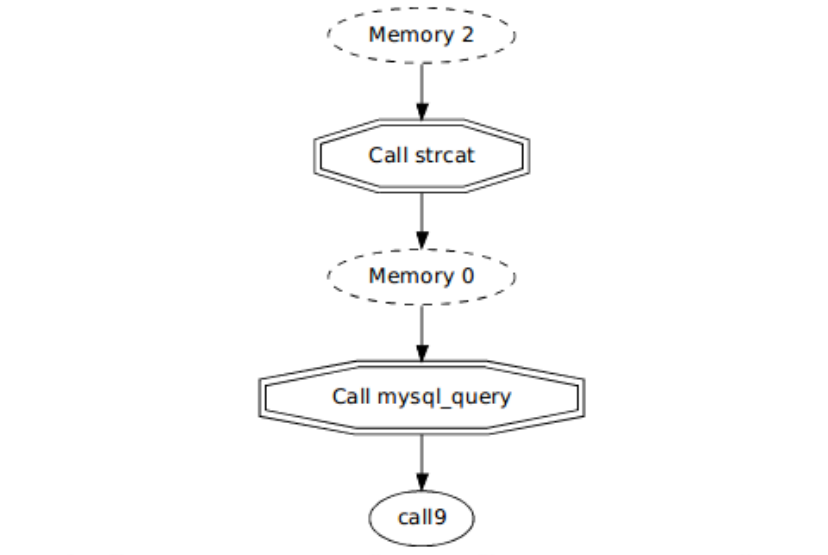
Figura 21 – Saída do Flow Tracking para o código 5.5.

```
MacBookAir-Leo:monografia leonardoribeiro$ ./dotgen-mac.bin programas/teste11c.c
********* Flow Tracking Summary ************

Sources: 1
Public channels: 1
Number of address leak paths 1

Writing 'cfg.main.dot'...
MacBookAir-Leo:monografia leonardoribeiro$
```

Fonte: o autor.

A figura 22 mostra o grafo de dependência gerado que explicita o fluxo contaminado presente no código 5.5. Analisando o grafo é possível ver que através da posição de memória manipulada pela função *scanf*, chamada de *Memory 2*, pode-se chegar até a função *mysql_query*. Este fluxo contaminado pode ser explorado através de um ataque *SQL Injection*. O Flow Tracking modificado detectou e mostrou esse fluxo contaminado.

Figura 22 – Grafo de dependência gerado para o código 5.5, considerando as chamadas em *assembly* LLVM.

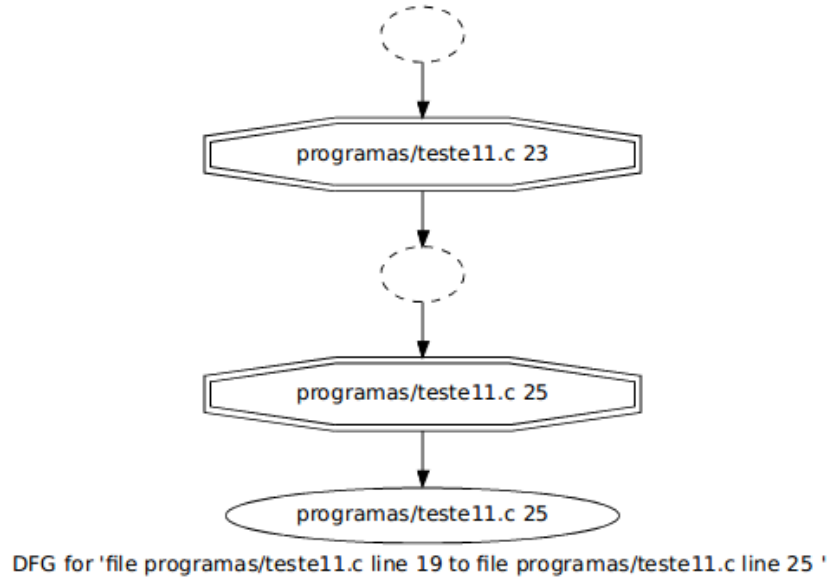


Fonte: o autor.

A figura 23 mostra o grafo de dependência do fluxo contaminado levando em consideração as linhas do programa. A partir da linha 23, onde a função *strcat* usa a

variável que foi carregada pela função *scanf*, existe um fluxo de dados que chega até a função *mysql_query*, na linha 25, através da variável *statement*.

Figura 23 – Grafo de dependência gerado para o código 5.5, considerando as linhas do código-fonte.



Fonte: o autor.

5.5 Considerações finais

O Flow Tracking modificado, implementado neste trabalho, foi capaz de detectar fluxos contaminados através da funcionalidade que diz se a função fonte manipula memória. Foi possível detectar vulnerabilidades que podem ser exploradas por ataques como SQL Injection, execução de comandos não desejáveis e ataques ao sistema de arquivos. Também tornou a análise de programas mais rápida pois permite uma parametrização das funções fontes e saídas que se deseja analisar. Estudar uma implementação de tipos sensível ao fluxo do programa permitiu entender melhor como evitar vulnerabilidades comumente presente nos programas manipulados.

O processo de análise dos fluxos de informações em programas, no momento da compilação, pode ser lento e complicado. Deve-se buscar melhorar esta analise utilizando-se de algoritmos e técnicas mais eficazes.

Este trabalho pode ser considerado um passo na melhoria da segurança de dados em programas de computador.

6 CONCLUSÃO

Esse trabalho apresentou a análise de fluxos explícitos e implícitos em programas. Foram detalhados também os problemas de vazamento de informação. Foi visto que é possível detectar tais vulnerabilidades em códigos a nível de compilador. O Flow Tracking, que implementa o Sistema de Tipos de Hunt e Sands (HUNT; SANDS, 2006) no compilador LLVM, foi apresentado. Esta implementação reduziu aquele sistema de tipos a um problema de busca em grafos. Foram mostradas análises feitas em alguns *benchmarks* utilizando o Flow Tracking.

Após o estudo do Flow Tracking foi proposta uma modificação para melhorar a análise de programas. Esta melhoria permite ao usuário informar no momento da análise quais fontes e saídas serão consideradas no programa analisado. Para tal modificação foi criado um *parser* que, integrado ao Flow Tracking, busca as informações da análise em um arquivo XML.

A principal vantagem de um sistema de detecção de fluxo de informação, como o descrito neste trabalho, é sua capacidade de reportar que o programa é livre de vazamento de informação. Assim, é possível saber que esse código não deixa conhecimento escapar para um adversário de forma explícita ou implícita.

Esse tipo de análise, através de uma ferramenta como o Flow Tracking, é um incentivo a melhoria da segurança em aplicações, pois permite ao programador auditar códigos evitando vazamento de informação.

7 PROPOSTAS DE CONTINUIDADE

Os testes realizados comprovam que as ideias apresentadas neste trabalho têm efetividade no combate de situações de vazamento de informação e de inserção de dados maliciosos que podem comprometer programas. Foi possível analisar alguns programas com falhas de segurança que foram reportadas adequadamente. O problema de vazamentos de endereço é pouco estudado pela academia mas é bem conhecido por profissionais de segurança e hobistas.

A análise mostrada neste trabalho tende a indicar vazamento de endereços em programas que na realidade são seguros. Ou seja, tem-se uma taxa falsos positivos alta. Existem propostas de monitores dinâmicos (RUSSO; SABELFELD, 2010) que reportam as mesmas vulnerabilidades que o sistema de tipos de Hunt e Sands (HUNT; SANDS, 2006), mas estas propostas não chegaram ainda a serem testadas em monitores reais.

Como propostas de continuidade deste projeto pode-se citar:

- A criação de uma busca invertida nos grafos de dependência. No qual, a partir das saídas, se verifique se é possível chegar às fontes. Esta questão é importante pois pode tratar da análise de valores recuperáveis. Uma vez que se sabe os valores de saída de certas variáveis, é interessante saber qual o traço executado que gerou tais valores. Por exemplo, é possível saber o valor que cada variável neste traço assume de forma a culminar nos valores de saída observados.
- Ampliação da estrutura do arquivo XML, permitindo passar parâmetros para o Flow Tracking para detalhar a análise. Por exemplo, configurar o *pass* para analisar instruções *assembly* do LLVM (*opcodes*) ao invés de funções da linguagem C.
- Reduzir o tempo de busca nos grafos dependência, implementando alguma técnica de processamento paralelo, por exemplo.

Referências

BACE, R. In: *Intrusion Detection*. [S.l.]: Macmillan Technical Publishing, 2000. Citado na página 13.

BODIK, R.; GUPTA, R.; SARKAR, V. ABCD: eliminating array bounds checks on demand. In: *PLDI*. [S.l.]: ACM, 2000. p. 321–333. Citado na página 18.

BUCHANAN, E. et al. When good instructions go bad: generalizing return-oriented programming to RISC. In: *CCS*. [S.l.]: ACM, 2008. p. 27–38. Citado na página 17.

CYTRON, R. et al. Efficiently computing static single assignment form and the control dependence graph.). ACM Trans, 1989. Citado na página 28.

DENNING, D. E.; DENNING, P. J. Certification of programs for secure information flow. *Commun. ACM*, ACM, v. 20, p. 504–513, 1977. Citado na página 17.

FEOFILOFF, P. *Alocação dinâmica de memória*. 2014. Disponível em: <http://www.ime.usp.br/~pf/algoritmos/aulas/aloca.html>. Acesso em: 30 apr. 2014. Citado na página 20.

GARFINKEL, G. S. e S. In: *Practical Unix and Internet Security*. [S.l.]: O Reilly and Associates, 1996. Citado na página 13.

HUNT, S.; SANDS, D. On flow-sensitive security types. In: *POPL*. [S.l.]: ACM, 2006. p. 79–90. Citado 6 vezes nas páginas 15, 17, 22, 23, 48 e 49.

JOVANOVIC, N.; KRUEGEL, C.; KIRDA, E. Pixy: A static analysis tool for detecting web application vulnerabilities (short paper). In: *Symposium on Security and Privacy*. [S.l.]: IEEE, 2006. p. 258–263. Citado 2 vezes nas páginas 18 e 20.

LEVANDOSKI, F. Análise de vulnerabilidades de um código fonte escrito em linguagem c. Universidade do Vale do Rios dos Sinos, 2011. Citado na página 14.

MELO, C. E.; FURTADO, O. J. V. Geração de código para a máquina virtual llvm a partir de programas escritos na linguagem de programação java (tradutor java - llvm). In: *Trabalho de Conclusão de Curso apresentado como requisito parcial para obtenção do grau de bacharel em Sistemas de Informação*. [S.l.: s.n.], 2008. Citado na página 25.

ORBÆK, P.; PALSBERG, J. Trust in the λ-calculus. *J. Funct. Program.*, Cambridge University Press, New York, NY, USA, v. 7, n. 6, p. 557–591, nov. 1997. Citado na página 18.

PHOENIX, B. *Boas práticas de segurança*. 2014. Disponível em: <www.bluephoenix.pt>. Acesso em: 15 mai. 2010. Citado na página 13.

QUADROS, G. S.; PEREIRA, F. M. Q. ao. Static detection of address leaks. In: *SBSeg*. [S.l.: s.n.], 2011. p. 23–37. Citado na página 17.

RIMSA, A. A. et al. Efficient static checker for tainted variable attacks. *Science of Computer Programming*, n. 2, p. 2–24, 2012. Citado 2 vezes nas páginas 18 e 20.

RUSSO, A.; SABELFELD, A. Dynamic vs. static flow-sensitive security analysis. In: *CSF*. [S.l.]: IEEE Computer Society, 2010. p. 186–199. Citado 2 vezes nas páginas 17 e 49.

SANTOS, H. N.; PEREIRA, F. M. Q.; OLIVEIRA, L. B. Verificacao estática de acessos a arranjos em c. In: *Anais do XIV Simpósio Brasileiro em Seguranca da Informacao e de Sistemas Computacionais*. [S.l.: s.n.], 2013. p. 198–211. Citado na página 18.

SCOTT, D.; SHARP, R. Specifying and enforcing application-level web security policies. *Trans. on Knowl. and Data Eng.*, IEEE, v. 15, p. 771–783, 2003. Citado na página 21.

SILVA, B. R.; PEREIRA, F. M. Q.; OLIVEIRA, L. B. Uma representação intermediária para a detecção de vazamentos implícitos de informação. In: . [S.l.]: SBSEG, 2013. Citado 12 vezes nas páginas 13, 14, 15, 17, 18, 19, 20, 21, 22, 24, 37 e 39.

SPAKI, E. Apresentando a ssa (static single assignment). 2009. Disponível em: <http://pt.scribd.com/doc/22736240/do-a-SSA>. Acesso em: 30 apr. 2014. Citado na página 26.

TRIPP, O. et al. TAJ: Effective taint analysis of web applications. In: *PLDI*. [S.l.]: ACM, 2009. p. 87–97. Citado 3 vezes nas páginas 14, 20 e 44.