Prevenção de Ataques em Sistemas Distribuídos via Análise de Intervalos

Vitor Mendes Paisante, Luiz Felipe Zafra Saggioro, Raphael Ernani Rodrigues, Leonardo Barbosa Oliveira, Fernando Magno Quintão Pereira

[1]Departamento de Ciência da Computação – UFMG
Av. Antônio Carlos, 6627 – 31.270-010 – Belo Horizonte – MG – Brazil

{paisante,luizfzsaggioro,raphael,leob,fernando}@dcc.ufmg.br

Abstract. *The range analysis of integer variables determines the lowest and highest bounds that each variable assumes throughout the execution of a program. This technique is vital to detect a plethora of software vulnerabilities but the literature does not describe any principled way to apply range analysis on distributed systems. This negligence is unfortunate, as networks are the most common targets of software attacks. The goal of this paper is to set right this omission. Capitalizing on a recent algorithm to infer communication protocols, we have designed, implemented and tested a range analysis for distributed systems. Our contribution, a holistic view of the system, is more precise than analyzing each system module independently. In this paper we support this statement through a number of examples, and experiments performed on top of the SPEC CPU 2006 benchmarks. A prototype of our tool, implemented on the LLVM compiler, is available for scrutiny.*

Resumo. *A análise de largura de variáveis determina o maior e menor valores que cada variável inteira de um programa pode assumir durante a sua execução. Tal técnica é de suma importância para detectar vulnerabilidades em programas mas, até o momento, não existe abordagem que aplique essa análise em sistemas distribuídos. Tal omissão é séria, uma vez que esse tipo de sistema é alvo comum de ataques de software. O objetivo deste artigo é preencher tal lacuna. Valendo-nos de um algoritmo recente para inferir protocolos de comunicação, nós projetamos, implementamos e testamos uma análise de largura de variáveis para sistemas distribuídos. Nosso algoritmo, ao prover uma visão holística do sistema distribuído, é mais preciso que analisar cada parte daquele sistema separadamente. Demonstramos tal fato via uma série de exemplos e experimentos realizados sobre os programas presentes em SPEC CPU 2006. Um protótipo de nossa ferramenta, implementado sobre o compilador LLVM, está disponível para escrutínio.*

1. Introdução

A análise de largura de intervalos [Cousot and Cousot 1977], é uma das técnicas mais importantes que compiladores usam para encontrar vulnerabilidades em programas. Essa análise determina, para cada variável inteira usada em um programa, quais são o menor e o maior valores que ela pode assumir. Tal informação permite ao compilador detectar a possibilidade de ocorrência de dois fenômenos que comprometem a segurança de programas. O primeiro deles é o acesso fora de limites de arranjos. Esse evento ocorre quando

uma variável inteira i indexa um endereço inválido a partir de um ponteiro base a. Erros assim são comuns em linguagens fracamente tipadas, como C, uma vez que a expressão $a[i]$ não assegura que i seja menor que o maior endereço dereferenciável a partir de a. O segundo fenômeno que a análise de largura de variáveis descobre estaticamente são os estouros de inteiros. Uma operação como $j = i + 1$, em linguagens como C, C++ ou Java, pode retornar um valor j menor que i, se i for o maior inteiro representável. Por razões que discutiremos na seção 4, essa semântica pode levar a vulnerabilidades de software.

A análise de largura de variáveis existe há quase 40 anos. Desde a sua concepção original, em 1977 [Cousot and Cousot 1977], vários desafios relacionados à implementação dessa análise foram superados, tanto em termos de precisão [Gawlitza et al. 2009, Su and Wagner 2005], quanto em termos de eficiência [Logozzo and Fahndrich 2008]. Recentemente, por exemplo, cientistas demonstraram como propagar informações de largura de variáveis em estruturas de dados [Oh et al. 2011], eliminando um dos últimos entraves ao projeto de análises de grande precisão. Entretanto, pesquisadores ainda não haviam abordado a análise de largura de variáveis em programas distribuídos. O presente artigo trata desta abordagem.

O grande empecilho à análise de sistemas distribuídos devia-se a um fato simples: até pouco tempo atrás não havia método confiável para determinar, estaticamente, quais operações de envio e recepção de mensagens se comunicam. Em outras palavras, uma vez que operações como `receive`, que coleta mensagens da rede, eram consideradas inseguras, pouco se podia assumir quanto aos valores coletados, já visto que eles podem provir de quaisquer fontes. Entretanto, esse problema foi superado por Teixeira *et al.* [Teixeira et al. 2014] neste ano de 2014. Teixeira *et al.* desenvolveram um algoritmo que infere canais de comunicação entre programas que integram um sistema distribuído. Valendo-nos de tal método, nós projetamos, implementamos e testamos um algoritmo que propaga informações de largura de variáveis entre nós que se comunicam em uma rede.

Nossa solução para o problema da análise de largura de variáveis em sistemas distribuídos consiste em cinco passos. (i) Nós determinamos quais dados representam as mensagens que um programa manipula. (ii) Nós aplicamos a análise de largura de variáveis nesses dados, para determinar o *layout* das mensagens do programa. (iii) Usando as técnicas de Teixeira *et al.*, nós determinamos quais os canais de comunicação existem entre os programas distribuídos. (iv) Nós emparelhamos as mensagens trocadas por esse canal, para determinar quais dados estão fluindo de um programa para o outro. (v) Nós executamos a análise de largura de variáveis uma segunda vez, agora sobre todo o sistema distribuído, obtendo resultados finais. *Dentre esses cinco passos, somente (iii) não é uma contribuição original deste artigo.* Enfatizamos que os passos (i) e (ii) descobrem não somente o *layout* de mensagens, mas o *layout* de arranjos em geral. O fato de podermos entender, de forma automática, como campos estão dispostos em mensagens é uma consequência dessa generalidade. Nossa ferramenta foi implementada sobre o compilador LLVM [Lattner and Adve 2004], e está disponível publicamente.

2. Contextualização

A fim de ilustrar a importância do problema abordado neste artigo, usaremos os dois programas vistos na figura 1. O código mostrado na parte (a) da figura lê uma quantidade `N` de caracteres da entrada padrão, e os envia através de uma conexão de rede, para o programa

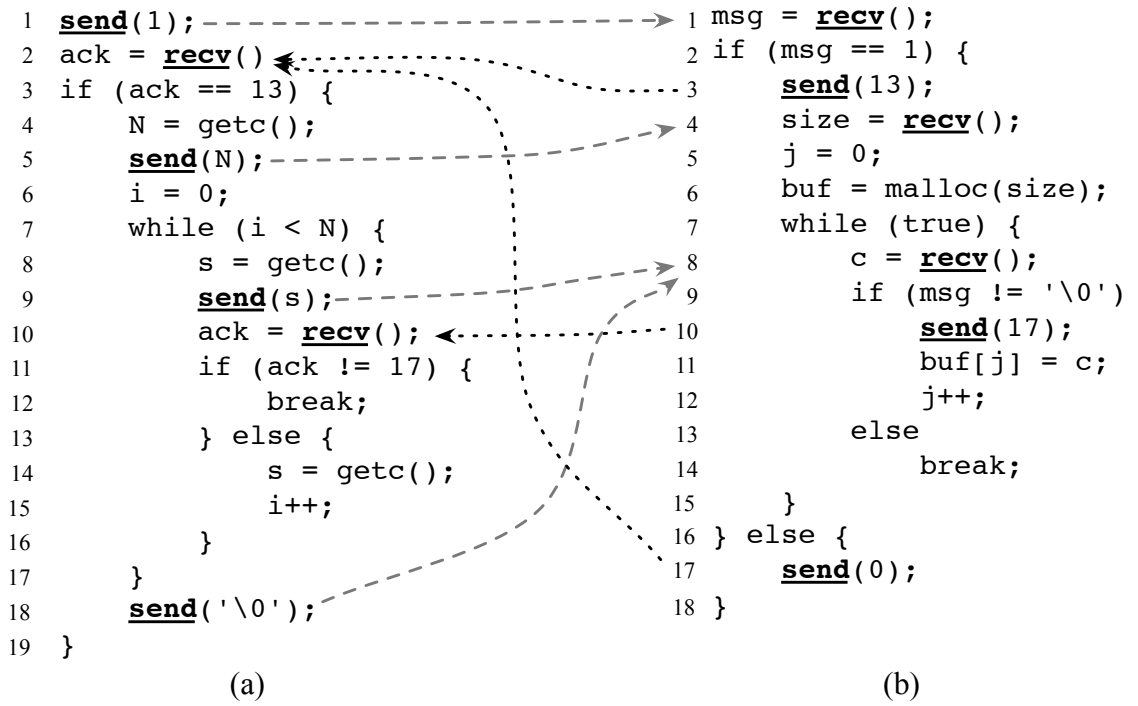

Figura 1. Canais implícitos de comunicação em programa distribuído.

na figura 1 (b). O programa (b) recebe esses caracteres e os armazena em um arranjo de tamanho `N`. Caso fosse possível escrever mais que `N` caracteres no arranjo usado no programa (b), então observaríamos a ocorrência de um fenômeno chamado *buffer overflow*. Esse é um dos principais meios que atacantes usam para comprometer o funcionamento de programas. O nosso programa exemplo, contudo, é seguro: o programa (a) nunca transmite para o programa (b) mais que `N` *bytes* de dados. Entretanto, a análise individual do programa (b) não nos permite inferir tal fato: na ausência de maiores informações, um analisador estático deve, conservadoramente, assumir que o laço na linha 7 daquele programa pode executar mais que `N` iterações.

Cada operação de `send` que se comunica com uma instrução `recv` cria um canal de comunicação *implícito*. A definição de todos os canais implícitos em um sistema distribuído é um problema indecidível. Existem, contudo, algoritmos que apontam a possibilidade de existência de tais canais de forma relativamente precisa. Um deles foi recentemente proposto por Teixeira *et al.*. Os possíveis canais de comunicação implícitos que esse algoritmo encontra aparecem indicados pelas setas tracejadas na figura 1. Analisando esses canais, pode-se concluir que as variáveis `N` e `size` possuem o mesmo valor.

3. Arcabouço de Análises Estáticas

Nossa análise distribuída de largura de variáveis segue a sequência de passos mostrada na figura 2. A análise de segmentação infere os diferentes campos que constituem cada mensagem trocada em um sistema distribuído. A análise local de intervalos nos permite encontrar qual informação está armazenada em cada campo de uma mensagem. A propagação de informações nos diz quais dados fluem de um programa para outro, efetivamente habilitando a última fase de nossa abordagem: a análise global de intervalos.

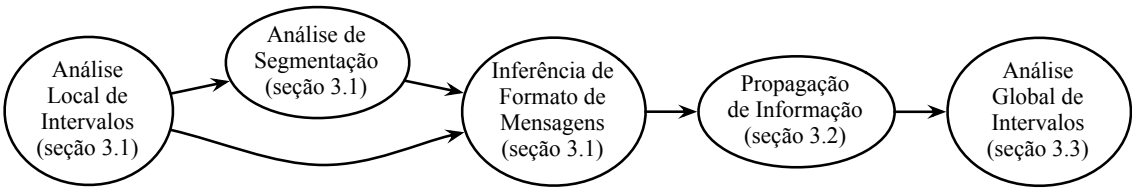

Figura 2. Análises estáticas que compõem o nosso arcabouço de inferência de largura de variáveis em sistemas distribuídos. Todas essas etapas são contribuições deste artigo.

Cada uma dessas análises é discutida nas próximas seções deste artigo.

3.1. Inferência Automática de Formato de Mensagens

O objetivo deste trabalho é portar uma análise de largura de variáveis tradicional para um sistema distribuído. Para alcançar esse objetivo, o primeiro desafio que precisamos vencer é como entender o formato das mensagens trocadas via rede. Normalmente, uma mensagem é implementada como um arranjo. Algumas células desses arranjos são agrupadas em *campos*. Diferentes programas encadeiam esses campos de diferentes maneiras. Campos de mensagens podem conter, por exemplo, seu *opcode*[1], o identificador do remetente, um contador de tempo (costumeiramente conhecido como *time-stamp*), e dados. A figura 3 (a) ilustra um programa que cria um dentre dois tipos diferentes de mensagens, e envia a mensagem criada através de uma operação **send**.

Análise Local de Intervalos. Para inferir como informações são passadas entre nós que se comunicam via rede, utilisamos a mesma análise de largura de variáveis que queremos portar para o mundo distribuído. Porém, dessa vez nós executamos tal análise *localmente*, isto é, de forma individual para cada programa que faz parte do sistema distribuído. A análise de largura de variáveis local nos dá, para cada variável inteira, uma função R, definida da seguinte maneira:

$$R(v) = [l, u], \;\; \{l, u\} \subset \mathbb{Z} \cup \{-\infty, +\infty\}, l \leq u$$

Como existem várias implementações de análises de intervalos descritas na literatura [Cousot and Cousot 1977, Gawlitza et al. 2009, Rodrigues et al. 2013, Su and Wagner 2005], nós omitiremos os detalhes do algoritmo que infere a função R automaticamente. Ao leitor interessado, recomendamos o trabalho de Rodrigues *et al.* [Rodrigues et al. 2013], que descreve uma implementação eficiente de tal análise. Assumiremos, portanto, a existência de uma técnica para construir a função R, que nos informa, para cada variável v, uma estimativa do menor e do maior valores que v assume durante a execução de um programa.

Análise de Segmentação. De posse da função R, passamos à segunda fase de nossa técnica. Nessa etapa, nosso objetivo é inferir para cada ponteiro p uma *tabela de segmentos* que o descreva. A tabela de segmentos associada a um ponteiro p é uma lista de intervalos que podem ser usados para indexar partes de p. Continuando com o nosso exemplo, a figura 3 (b) mostra os intervalos que podem ser usados para indexar o arranjo apontado pelo ponteiro a. Os diferentes ponteiros usados para carregar dados em

[1]O *opcode* de uma mensagem é um valor que descreve o tipo daquela mensagem, e permite ao seu receptor escolher a forma de tratamento mais adequado para ela.

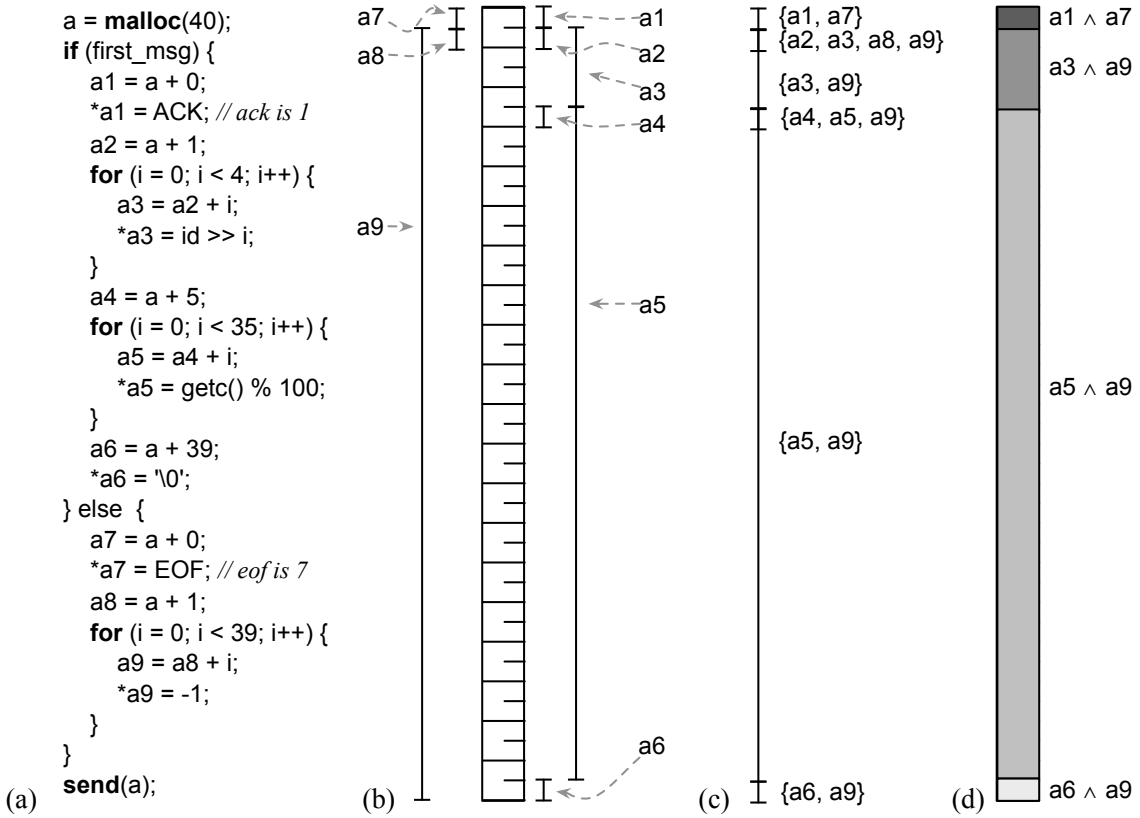

Figura 3. (a) Exemplo de programa que cria e envia mensagens. (b) Os intervalos dos vários ponteiros que podem indexar blocos na mensagem. (c) Outra visão dos intervalos de ponteiros, agrupados por área comum de indexação. (d) A tabela de segmentos da mensagem, formada pelos diferentes ponteiros usados para efetivamente armazenar dados nela via operações de carregamento.

$$
\begin{aligned}
a = \texttt{malloc}(v) \quad &\Rightarrow \quad T(a) = [0, u], \text{ sendo } R(v) = [l, u] \\
a' = a + c \quad &\Rightarrow \quad \begin{cases} R(a') = [\min(l + c, l'), \max(u + c, u')], \\ \qquad\qquad \text{sendo } R(a) = [l, u], R(a') = [l', u'] \\ T(a) = T(a') \end{cases} \\
*a = x \quad &\Rightarrow \quad \text{join}(T(a), R(a))
\end{aligned}
$$

Figura 4. Equações usadas para encontrar os segmentos que constituem cada arranjo de um programa.

um arranjo determinam a sua tabela de segmentos. Assim, a tabela de segmentos para o ponteiro a de nosso exemplo pode ser vista na figura 3 (d). A tabela de segmentos T associada a um arranjo a é construída de acordo com as equações mostradas na figura 4.

A função `merge`, usada para tratar operações de carregamento, e.g., $*a = x$, é definida na figura 5. Nessa implementação, nós usamos a sintaxe de ML, uma linguagem de programação funcional. Tabelas são representadas como listas de tu-


```
fun assertContiguous [] = true
  | assertContiguous [_] = true
  | assertContiguous ((l1, u1, _) :: (l2, u2, p) :: r) =
     u1 = l2 andalso assertContiguous ((l2, u2, p) :: r)

fun merge [] (l, u, p) = [(l, u, [p])]
  | merge ((ll, uu, pp)::r) (l, u, p) =
    if l > uu
    then (ll, uu, pp) :: merge r (l, u, p)
    else if l = ll
       then if u < uu
            then (ll, u, p::pp) :: (u, uu, pp) :: r
            else (ll, uu, p::pp) :: merge r (uu, u, p)
       else if u < uu
            then (ll, l, pp) :: (l, u, p::pp) :: (u, uu, pp) :: r
            else (ll, l, pp) :: (l, uu, p::pp) :: merge r (uu, u, p)


fun join t (l, u, p) = if assertContiguous t
                       then merge t (l, u, p)
                       else nil
```

```
join [(0, 39, [a])]
     (0, 1, a1)
     = [ (0,1,[a1,a]),(1,39,[a]) ]

join [ (0,1,[a1,a]),(1,39,[a]) ]
     (1, 5, a3)
     = [ (0,1,[a1,a]),(1,5,[a3,a]),
         (5,39,[a]) ]

join [ (0,1,[a1,a]),(1,5,[a3,a]),(5,39,[a]) ]
     (1, 5, a9)
     = [ (0,1,[a1,a]),(1,5,[a9,a3,a]),
         (5,39,[a]) ]

join [ (0,1,[a1,a]),(1,5,[a9,a3,a]),
     (5,39,[a]) ]
     (5, 38, a5)
     = [ (0,1,[a1,a]),(1,5,[a9,a3,a]),
         (5,38,[a5,a]),(38,39,[a]) ]
```

Figura 5. Algoritmo que faz o emparelhamento de tabelas de ponteiros. À direita do algoritmo mostramos algumas chamadas da função `join` para os ponteiros vistos na figura 3 (a).

plas. Cada tupla possui três elementos, e.g., (l, u, pp). Os inteiros l e u representam o início e o final do segmento. A lista pp guarda todos os ponteiros que são usados para indexar aquele segmento. O operador ::, em ML, denota a concatenação de listas. Assim, a tabela vista na figura 3 (c) é representada pela seguinte notação: $[(0, 1, [a_1, a_7]), (1, 5, [a_3, a_9]), (5, 38, [a_5, a_9]), (38, 39, [a_6, a_9])]$. Note que os segmentos determinam uma classe de equivalência sobre a tabela. Em outras palavras, cada célula de um arranjo pertence a um segmento e a interseção de dois segmentos diferentes sempre é vazia. Essas propriedades implicam em *contigüidade*, isso é, o intervalo final de um segmento é o intervalo inicial de seu vizinho. Nós salientamos essa propriedade via a função `assertContiguous`, a qual pode ser vista na figura 5. A função `join` recebe uma tabela t e um intervalo para ser inserido em t. Verificada a contigüidade de t, `join` modifica t via uma invocação de `merge`, para que ela passe a conter o novo segmento. As diversas possibilidades de modificação são vistas na parte direita da figura 5.

Quão precisa é nossa análise de segmentação? Neste artigo, estamos interessados em descobrir o *layout* de arranjos usados como mensagens em sistemas distribuídos. Por outro lado, a nossa análise de segmentação é mais geral: ela descobre segmentos em **qualquer** arranjo usado em um programa. Assim, podemos mensurar a precisão da análise contando o número de segmentos descobertos por arranjo: quanto mais segmentos descobrirmos por arranjo, mais precisa é nossa análise de segmentação. A figura 6 mostra esse número para os arranjos nos programas de SPEC CPU 2006. Essa tabela inclui todos os arranjos encontrados naquele *benchmark*. Essa abrangência nos dá uma idéia muito melhor da precisão de nossa análise, que se restringíssemos esse experimento somente aos arranjos trocados como mensagens em sistemas distribuídos, pois esses seriam poucos.

A figura 6 conta o número de tabelas de segmentos, em vez do número de arranjos, pois a mesma tabela pode ser formada por arranjos diferentes. Isso acontece quanto a análise de ponteiros de LLVM não consegue dizer se dois ponteiros podem ou não refe-

Benchmark	#Tabelas	#médio de segmentos	Maior tabela	%Arranjos
433.milc	631	3	34	85.00%
444.namd	617	4	44	72.00%
447.dealII	9,843	2	109	75.00%
450.soplex	2,784	2	55	89.00%
470.lbm	24	16	152	99.00%
401.bzip2	493	2	92	93.00%
429.mcf	103	3	19	68.00%
456.hmmer	4,872	2	64	86.00%
458.sjeng	356	2	14	96.00%
462.libquantum	343	2	6	99.00%
464.h264ref	15,436	2	45	97.00%
471.omnetpp	2,518	2	50	58.00%
473.astar	333	3	22	95.00%
483.xalancbmk	25,194	2	62	82.00%
Total/Média/Max/Média	4,539	3.4	152	85.29%

Figura 6. Precisão de nossa análise de segmentação.

renciar o mesmo endereço base. Pela figura 6, vemos que, em média, cada tabela possui 3.4 segmentos. A maior tabela que observamos, presente em `lbm`, possui 152 segmentos. Em outras palavras, essa tabela representa um arranjo indexado por 152 variáveis contendo intervalos de valores diferentes. A figura 6 contém uma coluna *%Arranjos*, que descreve a porcentagem de arranjos que pudemos analisar por *benchmark*. Somente analisamos arranjos cujo ponteiro base é conhecido. Ou seja, precisamos encontrar, no código do programa, o ponto de criação do arranjo. Arranjos alocados por funções externas[2], por exemplo, não podem ser analisados.

Inferência de Intervalos em Campos de Mensagens. Feita a segmentação da memória nos programas que formam o sistema distribuído, passamos à fase de inferência de valores em mensagens. Nessa etapa, estamos interessados em encontrar quais os intervalos de valores que podem ser armazenados em cada segmento. Fazemos isso via o laço abaixo:

- Para cada instrução de carregamento $*a = x$ presente no programa:
  - Para cada segmento s que contém a in $T(a)$ faça
    - $R(s) = R(s) \cup R(x)$

A figura 7 mostra o resultado da inferência de mensagens quando aplicada no programa visto na figura 3 (a). Inicialmente, todos os segmentos da tabela associada ao arranjo a contém intervalos inteiros indefinidos, indicados pela notação $[?, ?]$. Durante o processamento das instruções de carregamento, os valores desconhecidos são substituídos pelos valores encontrados via a análise local de intervalos. Se duas ou mais instruções, tais como $*a = x_1$ e $*a = x_2$, carregam valores nos mesmos segmentos, então esse segmento recebe a união dos intervalos $R(x_1) \cup R(x_2)$. Esse fenômeno pode ser visto durante o processamento da instrução $*a_7 =$ EOF, na figura 7, que expande o segmento associado à $\{a_5, a_9\}$, de $[0, 99]$ para $[-1, 99]$.

[2]Uma função é *externa* se o seu código fonte não está disponível para nosso compilador.

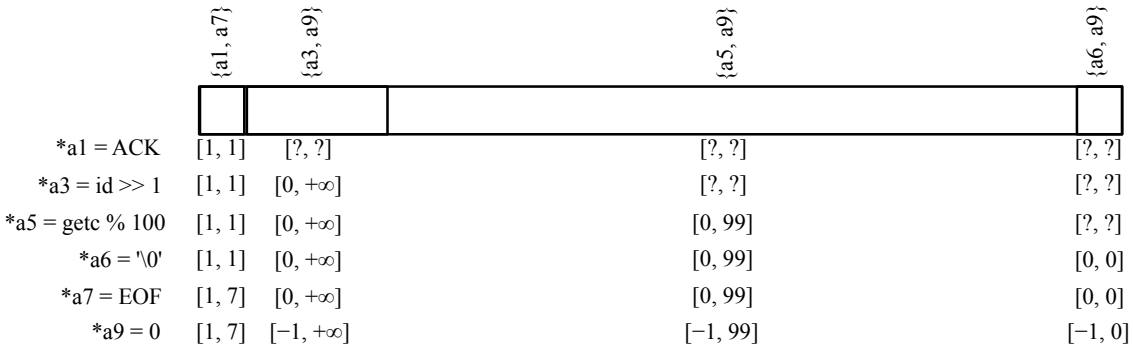

Figura 7. Inferência de intervalos inteiros na mensagem vista na figura 3.

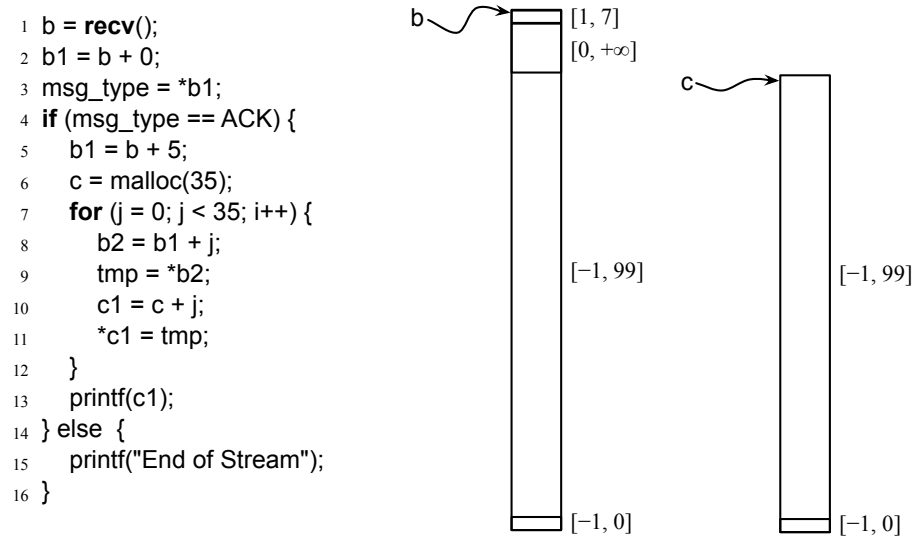

Figura 8. A esquerda, vê-se um trecho de código que recebe mensagens enviadas pelo programa da figura 3 (a). À direita, vêem-se mensagens recuperadas via o emparelhamento por canais implícitos.

A inferência de valores de mensagens termina tão logo todas as instruções de carregamento no programa sejam processadas. Cada instrução é visitada somente uma vez, então a terminação é garantida. Assim, a complexidade computacional dessa análise é proporcional ao número de instruções de carregamento no programa.

3.2. Propagação de Informação entre Programas

Finda a fase local de nosso algoritmo, que envolve os três passos descritos na seção 3.1, passamos à fase distribuída de nossa análise. Nessa etapa, tabelas de segmentos em programas diferentes são emparelhadas, de acordo com os canais de comunicação inferidos usando-se o algoritmo de Teixeira *et al.*. Para cada canal de comunicação inferido, as tabelas enviadas são emparelhadas com as tabelas recebidas. A figura 8 ilustra esse processo. O programa visto nessa figura recebe mensagens enviadas pelas instruções mostradas na figura 3 (a). Uma vez que existe um canal de comunicação implícito entre a instrução **send** da figura 3 (a) e a instrução **recv** da figura 8, temos que as tabelas associadas aos arranjos a e b devem ser emparelhadas.

A partir de um canal de comunicação implícito formado por uma operação de **send** s e uma operação de **recv** r, podemos definir as tabelas de segmentos origem e destino. Chamamos de tabela *origem* aquela associada a qualquer arranjo passado para s e *destino* a tabela associada a algum arranjo recebido por r. Dados esses conceitos, o emparelhamento de tabelas de segmentos é uma operação simples, e consiste no casamento de campos cujos índices se correspondem nas tabelas origem e destino. A figura 8 mostra as tabelas de segmentos associadas aos arranjos b e c, as quais obtemos via emparelhamento com a tabela origem associada ao arranjo a da figura 3 (a).

3.3. Análise Global de Largura de Variáveis

Terminado o emparelhamento, começamos a última fase da abordagem que esse artigo propõe: a análise global de intervalos. Essa etapa não requer qualquer algoritmo especialmente adaptado para o universo dos sistemas distribuídos. Para encontrar os intervalos associados às variáveis inteiras de cada programa que integra o sistema, podemos usar qualquer algoritmo já descrito na literatura. As informações necessárias ao correto funcionamento do algoritmo já foram inferidas nos passos anteriores. Essa flexibilidade é uma das vantagens de nossa abordagem.

Continuando com o nosso exemplo, nós temos que a variável `tmp`, inicializada na linha 9 da figura 8, pode conter somente valores dentro do intervalo $[-1, 99]$. Esse intervalo foi inferido para segmentos apontados pelo ponteiro b_2 no passo de propagação de informação entre nós comunicantes. Caso analisássemos o programa da figura 8 em separado, teríamos de assumir que a variável `tmp` pudesse ser inicializada com qualquer valor inteiro. Essa perda de precisão deve-se ao fato de uma análise individual não nos dar qualquer informação sobre dados recebidos via operações de **recv**.

4. Estudo de Caso

Nós implementamos nossa análise sobre o compilador LLVM, versão 3.3, pois tanto o trabalho de Teixeira *et al.* quanto a análise de largura de variáveis de Rodrigues *et al.* foram construídas nesse compilador. A fim de demonstrar o funcionamento da técnica proposta neste artigo, esta seção descreve sua utilização sobre um par cliente-servidor real. O código presente aqui pode ser compilado e testado diretamente[3]. O cliente usado neste estudo de caso envia para um servidor uma quantidade indeterminada de pares formados por nomes de funcionários e horas trabalhadas. As mensagens que carregam essas informações possuem dois campos. O servidor, ao receber cada um desses pares, multiplica a quantidade de horas trabalhadas por um valor de salário-base e envia de volta para o cliente uma tripla, formada pelo nome do funcionário, suas horas trabalhadas e seu salário. A figura 9 mostra o código de nossa aplicação cliente, e a figura 10 mostra o código de nossa aplicação servidora. Por simplicidade, neste exemplo operaremos somente a nível de *bytes*. Assim, nomes são cadeias de *bytes*, horas trabalhadas são um *byte* e o salário-base é um *byte* também.

A figura 9 mostra, além do programa cliente, o protocolo de comunicação de nosso estudo de caso. O cliente inicialmente informa ao servidor a quantidade de

[3]Devido a restrições de espaço, não mostramos as diretivas `#include` em nosso código. Assim, os seguintes arquivos devem ser incluídos em cada programa: `stdio.h`, `string.h`, `sys/socket.h` e `arpa/inet.h`.


```
1  int main() {
2     int i, num, salario_calc, descriptor, ack = 0;
3     char buffer[42], *message, server_reply[2000];
4     struct sockaddr_in server;
5
6     descriptor = socket(AF_INET , SOCK_STREAM , 0);
7     server.sin_addr.s_addr = inet_addr("localhost");
8     server.sin_family = AF_INET;
9     server.sin_port = htons(2000);
10    connect(descriptor, (struct sockaddr *)&server , sizeof(server));
11
12    // Informa ao servidor quantas mensagens serao enviadas:
13    fscanf(stdin, "%d", &num);
14    buffer[0] = num < 100 ? num : 0;
15    send(descriptor, buffer, 1, 0);
16    recv(descriptor, &ack, sizeof(int), 0);
17    if (ack) {
18       // Envia os pares: (nome do trabalhador x horas trabalhadas):
19       for (i = 0; i < num; i++) {
20          // Leia o nome do trabalhador:
21          fscanf(stdin, "%s", buffer);
22          // Leia o numero de horas trabalhadas:
23          fscanf(stdin, "%d", &buffer[40]);
24          // Permite-se no maximo dez horas-extras trabalhadas:
25          buffer[40] = buffer[40] > 10 ? 10 : buffer[40];
26          // Envia a mensagem:
27          send(descriptor, buffer, strlen(buffer) * sizeof(char), 0);
28          // Recebe o valor a ser pago ao trabalhador:
29          recv(descriptor, buffer, strlen(buffer) * sizeof(char), 0);
30          salario_calc = (int)buffer[41];
31          printf("Valor a ser pago = %d\n", salario_calc);
32       }
33    }
34    return 0;
35 }
```

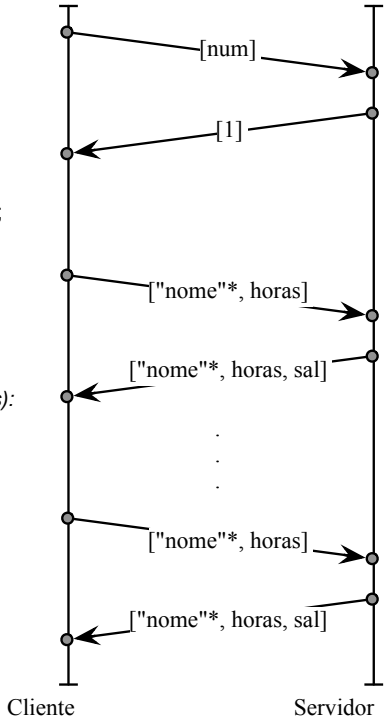

Figura 9. (Esquerda) Programa cliente. (Direita) Protocolo de comunicação.

```
1  int main() {
2     int num, i, salario, descriptor, connection, clilen;
3     char buffer[42];
4     struct sockaddr_in server, client;
5
6     descriptor = socket(AF_INET , SOCK_STREAM , 0);
7     server.sin_addr.s_addr = inet_addr("localhost");
8     server.sin_family = AF_INET;
9     server.sin_port = htons(2000);
10    bind(descriptor, (struct sockaddr *)&server, sizeof(server));
11    connection = accept(descriptor, (struct sockaddr *)&client, &clilen);
12    // Recebe o numero de mensagens que vao chegar:
13    recv(connection, &num, sizeof(int), 0);
14    // Envia um ack para o cliente:
15    buffer[0] = 1;
16    send(connection, buffer, 1, 0);
17    // Leia todas as mensagens:
18    for (i = 0; i < num; i++) {
19       recv(connection, &buffer, strlen(buffer)*sizeof(char), 0);
20       salario = HORA * buffer[40]; // HORA == 12
21       buffer[41] = salario;
22       // Envia o salario final para o cliente:
23       send(connection, buffer, strlen(buffer)*sizeof(char), 0);
24    }
25    return 0;
26 }
```

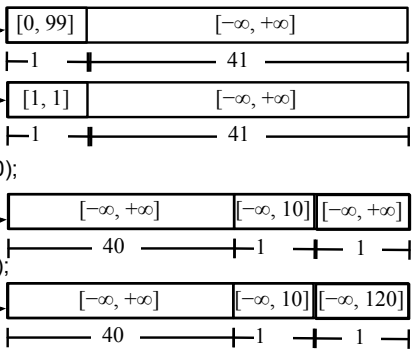

Figura 10. Programa servidor que interage com o cliente visto na figura 9. Os segmentos de mensagens que nossa análise infere automaticamente são mostrados à direita do programa.

mensagens trocadas. Essa primeira mensagem é seguida de uma confirmação de recebimento, por parte do servidor. A partir desse ponto, o cliente passa a enviar os pares (nomes $\times$ horas) para o servidor, que lhe envia de volta as triplas (nomes $\times$ horas $\times$ salário). Esse protocolo possui quatro canais implícitos de comunicação[4]: (i) `cliente:send:15` $\rightarrow$ `servidor:recv:13`, (ii) `servidor:send:16` $\rightarrow$ `cliente:recv:16`, (iii) `cliente:send:27` $\rightarrow$ `servidor:recv:19` e (iv) `servidor:send:23` $\rightarrow$ `cliente:recv:29`. O algoritmo proposto por Teixeira *et al.* descobre esses quatro canais e somente eles. Note que nesse caso, o algoritmo não reporta falsos-positivos: todos os canais inferidos são, de fato, canais válidos.

O algoritmo proposto na seção 3.1 descobre os formatos de mensagens vistos à direita da figura 10. Nesse exemplo, nosso algoritmo foi capaz de inferir, de forma precisa, os intervalos de valores associados a cinco dos dez campos de mensagens trocadas entre cliente e servidor. Por exemplo, o programa cliente possui um teste na linha 25 (figura 9) que garante que trabalhadores não podem fazer mais que 10 horas extra. Esse teste permite-nos determinar que somente valores no intervalo $[-\infty, 10]$ podem ser transferidos na posição 40 das mensagens. Valores imprecisos, associados aos intervalos $[-\infty, +\infty]$, devem-se à pouca informação disponível no código fonte do programa. São imprecisos, por exemplo, os valores dos *bytes* associados a nomes de funcionários. Esses *bytes* podem cobrir qualquer intervalo entre $[-128, 127]$, exatamente o domínio do tipo `char`.

4.1. Como utilizar nossa análise para aumentar a segurança e a eficiência de programas distribuídos.

Estamos, atualmente, usando os resultados de nossa análise para eliminar guardas sobre operações que podem causar estouro em aritmética de inteiros. Linguagens como C, C++ ou Java tratam operações inteiras segundo uma semântica modular. Se o resultado de uma instrução inteira for maior que o tamanho do registrador onde esse resultado será armazenado, então os *bits* mais significativos desse valor são descartados. Por exemplo, $6_{char} \times 22_{char} = -124_{char}$. A literatura contém várias descrições de ataques baseados nesse semântica [Brumley et al. 2007, Dietz et al. 2012].

Um ataque baseado em estouro de arranjos é possível em nosso estudo de caso. Considere, por exemplo, que um usuário malicioso informe um valor negativo de horas na linha 23 do programa cliente (variável `buffer[40]` na figura 9). Suponhamos que tal valor seja o inteiro negativo -75. Temos então que o teste na linha 25 do programa cliente é falso. Consequentemente -75 será transmitido para o servidor. No código do servidor (figura 10), a multiplicação na linha 20, e.g., $-75_{char} \times 12_{char} = 124_{char}$, produz um valor maior que o máximo número de horas cuja intenção do desenvolvedor seria permitir, isto é, $10_{char} \times 12_{char} = 120_{char}$. Esse tipo de falha de segurança é difícil de ser detectado sem o auxílio de ferramentas de análise estática, e pode levar a situações catastróficas. A título de exemplo, em 1996, o foguete Ariane 5 foi perdido devido a um estouro de inteiros – tal erro de software custou ao programa espacial europeu cerca de US$ 370 milhões [Dowson 1997].

Existem diversas técnicas para sanear programas contra estouros de operações inteiras. Recentemente, por exemplo, Rodrigues *et al.* propuseram um gerador de código que instrumenta operações inteiras em um programa. Essa instrumentação invoca código

[4]Números ao lado do nome do programa denotam linhas nas figuras 9 e figuras 10.

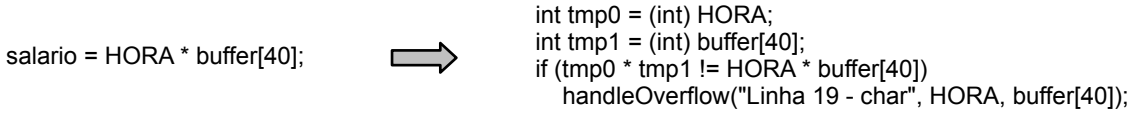

Figura 11. Exemplo de código para verificar se houve estouro de inteiro

de tratamento de erros sempre que um estouro é detectado. Tal técnica, quando aplicada ao programa da figura 10, insere guardas na soma da linha 18, e na multiplicação da linha 20. Cada uma dessas guardas é implementada como uma combinação de testes condicionais e instruções de desvio, conforme mostrado na figura 11.

Essas guardas tornam o programa instrumentado mais lento que o programa original. Conforme reportado por Dietz *et al.*, essa lentidão pode comprometer até 15% do tempo de execução do programa modificado [Dietz et al. 2012]. Nossa técnica nos permite eliminar alguns desses testes, e também indicar ao desenvolver quais testes precisam ser mantidos. Por exemplo, considerando-se o programa servidor, visto na figura 10, nossa análise elimina o teste sobre o incremento realizado na linha 18, pois a variável `i` é limitada por `num`, cujo intervalo superior pode ser no máximo 99. Por outro lado, não podemos eliminar a guarda da multiplicação da linha 20, pois `buffer[40]`, caso fosse um número negativo muito pequeno, causaria um estouro aritmético. Nossa análise detecta também essa possibilidade e mantém a instrumentação na linha 20.

5. Trabalhos Relacionados

O presente trabalho relaciona-se a pesquisa desenvolvida tanto em análise de código, quanto em sistemas distribuídos. No primeiro caso, nossa inspiração mais importante deve-se a Cousot e Cousot [Cousot and Cousot 1977], que introduziram o conceito de análise de largura de variáveis. No segundo caso, contudo, nossa inspiração é bem mais recente: muito do que discutimos neste artigo foi possível somente devido ao arcabouço construído por Teixeira *et al.*. No restante dessa seção discutiremos como nosso trabalho se relaciona com outras pesquisas nesses dois campos.

Análise de largura de variáveis. A análise de largura de variáveis é um dos exemplos clássicos de interpretação abstrata. A técnica de interpretação abstrata, introduzida por Cousot e Cousot, é um arcabouço teórico que permite a compiladores obter informações de um programa, garantindo que os algoritmos usados terminam. Existem muitas formas de se implementar análise de largura de intervalos [Gawlitza et al. 2009, Mahlke et al. 2001, Stephenson et al. 2000, Su and Wagner 2005]. Essas técnicas seguem duas avenidas principais, que, embora levem ao mesmo objetivo, atravessam caminhos muito diferentes. As técnicas mais conhecidas, como o trabalho de Stephenson *et al.* [Stephenson et al. 2000] ou Mahlke [Mahlke et al. 2001], baseiam-se em algoritmos iterativos. Em outras palavras, esses métodos interpretam as instruções de um programa abstratamente. O programa é interpretado de forma que o valor abstrato, isso é, o intervalo, associado a cada variável inteira somente cresce. Um operador especial, conhecido como alargamento, assegura que esse crescimento termina após algumas iterações. Existem implementações desses algoritmos em compiladores industriais, como Open64 ou LAO, usado pela companhia STMicroelectronics. A maior parte dos artigos acadêmicos, contudo, descrevem algoritmos que resolvem a análise de largura de

intervalos de forma não iterativa. Entre esses trabalhos, citam-se as técnicas de Su e Wagner [Su and Wagner 2005], Gawlitza *et al.* e Rodrigues *et al.*. Exceto o algoritmo de Rodrigues *et al.*, presente em LLVM, não sabemos de outras implementações de algoritmos não iterativos em compiladores de uso industrial.

Em que nosso trabalho difere das técnicas já existentes. O foco deste artigo não é em algoritmos de largura de variáveis *per se*. Nós queremos aplicar tais técnicas em sistemas distribuídos. Com tal propósito, podemos utilizar qualquer algoritmo existente. Neste trabalho usamos o método de Rodrigues *et al.*. Nossa escolha foi motivada por razões puramente pragmáticas: esse método já estava implementado sobre o compilador LLVM, o qual usamos em nossos experimentos. Nessa flexibilidade, conforme já mencionamos antes, reside muito da beleza de nossa abordagem: as técnicas descritas neste artigo, como a propagação de valores entre nós de programas distribuídos, a inferência de formatos de mensagens e a associação de valores abstratos a campos de mensagens são ortogonais à técnica de largura de variáveis usada.

Análise de sistemas distribuídos. Teixeira *et al.* introduziram o algoritmo que usamos para encontrar canais implícitos em sistemas distribuídos. Aquele trabalho identifica canais de comunicação com base nos comandos de rede. A partir desses comandos, Teixeira *et al.* realizam a interconexão dos grafos de controle de fluxo de cada programa. Existem outros trabalhos desenvolvidos com propósito semelhante. Entre eles, destacamos Kleenet, de Sasnauskas *et al.* [Sasnauskas et al. 2010] e T-Check, de Li *et al.* [Li and Regehr 2010]. Essas ferramentas executam o sistema simbolicamente a procura de defeitos de software e permitem a exploração automática de caminhos de execução em aplicações distribuídas. Se uma asserção falha, essas ferramentas registram o caso de teste para que o cenário possa ser repetido.

Em que nosso trabalho difere das técnicas já existentes. O trabalho de Teixeira *et al.* propõe um arcabouço para a análise de sistemas distribuídos, mas não implementa qualquer análise sobre ele. Os autores daquele projeto não tiveram, por exemplo, de lidar com o *layout* de mensagens. Tampouco foi essa uma preocupação de Sasnauskas *et al.* e Li *et al.*. Nesses dois casos, o desenvolvedor deve marcar variáveis para serem simbólicas e deve escrever asserções sobre o estado do sistema, ou seja, usuário deve, explicitamente, indicar ao analisador estático como dados trafegam em mensagens. Esse passo é manual e requer conhecimento sobre a lógica da aplicação e estruturas de dados. Assim, a complexidade da solução depende das entradas simbólicas e do número de nós. Os autores de Kleenet, por exemplo, reportaram que mesmo com entradas simbólicas pequenas e poucos nós, algumas aplicações geram milhares de caminhos de execução. Nossa abordagem é mais automática: o desenvolvedor indica quais funções fazem a comunicação de rede (passo também necessário para os trabalhos relacionados) e o compilador descobre como os dados são passados, sem qualquer intervenção do usuário.

6. Conclusão

Este artigo descreveu uma forma de inferir a largura de variáveis em programas distribuídos. Essa técnica dá a uma ferramenta de análise de código mais subsídios para encontrar vulnerabilidades em aplicações distribuídas. Demonstramos esse fato mostrando como nossa análise nos permite proteger código contra vulnerabilidades devido a estouro de operações aritméticas em valores inteiros. Nossa técnica exige mínima intervenção do

usuário, a saber, a indicação de quais funções fazem acesso à rede. Como trabalho futuro, pretendemos usar o arcabouço descrito neste artigo para sanear programas contra ataques de estouro de *buffer*. Estamos já trabalhando ativamente para alcançar tal objetivo.

Agradecimentos Este projeto é financiado pela Intel, pelo CNPq e pela FAPEMIG.

Referências

Brumley, D., Song, D. X., cker Chiueh, T., Johnson, R., and Lin, H. (2007). RICH: Automatically protecting against integer-based vulnerabil ities. In *NDSS*. USENIX.

Cousot, P. and Cousot, R. (1977). Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252. ACM.

Dietz, W., Li, P., Regehr, J., and Adve, V. (2012). Understanding integer overflow in c/c++. In *ICSE*, pages 760–770. IEEE.

Dowson, M. (1997). The ariane 5 software failure. *SIGSOFT*, 22(2):84–.

Gawlitza, T., Leroux, J., Reineke, J., Seidl, H., Sutre, G., and Wilhelm, R. (2009). Polynomial precise interval analysis revisited. *Efficient Algorithms*, 1:422 – 437.

Lattner, C. and Adve, V. S. (2004). LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, pages 75–88. IEEE.

Li, P. and Regehr, J. (2010). T-check: Bug finding for sensor networks. In *IPSN*, pages 174–185.

Logozzo, F. and Fahndrich, M. (2008). Pentagons: a weakly relational abstract domain for the efficient validation of array accesses. In *SAC*, pages 184–188. ACM.

Mahlke, S., Ravindran, R., Schlansker, M., Schreiber, R., and Sherwood, T. (2001). Bitwidth cognizant architecture synthesis of custom hardware accelerators. *TCADICS*, 20(11):1355–1371.

Oh, H., Brutschy, L., and Yi, K. (2011). Access analysis-based tight localization of abstract memories. In *VMCAI*, pages 356–370. Springer.

Rodrigues, R. E., Campos, V. H. S., and Pereira, F. M. Q. (2013). A fast and low overhead technique to secure programs against integer overflows. In *CGO*. ACM.

Sasnauskas, R., Landsiedel, O., Alizai, M. H., Weise, C., Kowalewski, S., and Wehrle, K. (2010). Kleenet: discovering insidious interaction bugs in wireless sensor networks before deployment. In *IPSN*, pages 186–196. ACM.

Stephenson, M., Babb, J., and Amarasinghe, S. (2000). Bitwidth analysis with application to silicon compilation. In *PLDI*, pages 108–120. ACM.

Su, Z. and Wagner, D. (2005). A class of polynomially solvable range constraints for interval analysis without widenings. *Theoretical Computer Science*, 345(1):122–138.

Teixeira, F., Pereira, F., Viera, G., Marcondes, P., Wong, H. C., and Nogueira, J. M. (2014). Siot: defendendo a internet das coisas contra exploits. In *SBRC*, pages 85–96. SBC.

