X-RA: UMA ANÁLISE DE INTERVALOS PARA PROGRAMAS EM REDE

LUIZ FELIPE ZAFRA SAGGIORO

X-RA: UMA ANÁLISE DE INTERVALOS PARA PROGRAMAS EM REDE

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

Orientador: Leonardo Barbosa e Oliveira
Coorientador: Fernando Magno Quintão Pereira

Belo Horizonte
Agosto de 2015

LUIZ FELIPE ZAFRA SAGGIORO

X-RA: A RANGE ANALYSIS FOR NETWORKED SYSTEMS

Dissertation presented to the Graduate Program in Computer Science of the Federal University of Minas Gerais in partial fulfillment of the requirements for the degree of Master in Computer Science.

ADVISOR: LEONARDO BARBOSA E OLIVEIRA
CO-ADVISOR: FERNANDO MAGNO QUINTÃO PEREIRA

Belo Horizonte

August 2015

Saggioro, Luiz Felipe Zafra

S129x X-RA: a range analysis for networked systems / Luiz Felipe Zafra Saggioro. — Belo Horizonte, 2015
xxi, 34 f. : il. ; 29cm

Dissertação (mestrado) — Universidade Federal de Minas Gerais — Departamento de Ciência da Computação
Orientador: Leonardo Barbosa e Oliveira
Coorientador: Fernando Magno Quintão Pereira

1. Computação - Teses. 2. Sistemas distribuídos (Computadores) – Teses 3. Compiladores (Programas de computador) -Teses. I. Orientador. II Coorientador. III. Título.

CDU 519.6*34(043)

UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO

X-RA: a range analysis for networked systems

LUIZ FELIPE ZAFRA SAGGIORO

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

[illegible]

PROF. LEONARDO BARBOSA E OLIVEIRA - Orientador
Departamento de Ciência da Computação - UFMG

Fernando

PROF. FERNANDO MAGNO QUINTÃO PEREIRA - Coorientador
Departamento de Ciência da Computação - UFMG

[illegible]

PROF. MARIO SÉRGIO FERREIRA ALVIM JÚNIOR
Departamento de Ciência da Computação - UFMG

[illegible]

PROF. OMAR PARANAIBA VILELA NETO
Departamento de Ciência da Computação - UFMG

Belo Horizonte, 17 de junho de 2015.

Acknowledgments

I am thankful to ..

My family for supporting me on this stage of my life. To my parents José Henrique and Lucia, that understood the reasons I could not visit them for several months and still supported me on my decisions.

My advisor Leonardo for keeping me on the right path, for coming up with constructive discussions and for trusting on me to during the execution of this work.

My co-advisor Fernando for the help, for the ideas in every discussion and for helping me better understand the world of compilers.

My fellows for the funny moments at the laboratory and to Fernando Teixeira for the great help.

My girlfriend Ana Paula, my partner for every moment, for the love and support during the hard times and for the awesome and philosophical conversations.

The Computer Science Department for the available infrastructure that allowed me to execute this project.

CAPES for the scholarship.

Intel for the funding of this project.

“The quieter you become, the more you can hear.”

(Ram Dass)

Abstract

New technologies such as Cloud Computing and the Internet of Things are increasing the importance of techniques to analyze and understand distributed systems. One such technique is range analysis: a compiler-oriented way to infer the lower and upper limits of the integer variables used in programs. Range analysis is useful to secure and optimize distributed code. However, the very distributed nature of such systems reduces the precision of range analysis. Imprecisions arise due to a simple observation: in the absence of a holistic view of the distributed system, the compiler must assume imprecise bounds for every value that a module receives from its peers. This work presents a solution to this problem. The solution was designed, tested and implemented, resulting in a range analysis algorithm that preserves information exchanged across the modules that constitute a distributed application. The goal is achieved by combining into a single framework three previous algorithms: classic range analysis, points-to analysis and communication links inference. To glue these technologies into a useful tool, this work introduces two novel techniques: message segmentation and array content inference. To validate our ideas, the proposed strategy was implemented in the LLVM compiler. When applied onto own crafted benchmarks, the distributed range analysis can be from 12% to 39% more precise than its original version.

List of Figures

List of Tables

Contents

Organization

This document is organized as follows. Chapter 1 quickly introduces the problem and approaches the motivation of this work. Chapter 2 describes the fundamental concepts of the used analyses. Chapter 3 presents the solution, which is later evaluated in Chapter 4. Related work is discussed in Chapter 5. The final remarks are presented in Chapter 6.

Chapter 1

Introduction

Since its early days, the analysis of distributed systems has been always an important problem in computer science (10). Recently, the emergence of cloud computing, smartphones and internet of things has increased the importance of such analyses (18). Among several different techniques to analyze distributed systems, one particular approach stands out: range analysis, or range inference, as it is also called. This compiler-based analysis estimates the lowest and the highest values that each integer variable can assume throughout the execution of a program (12). Range analysis has several different purposes. From a software security standpoint, it helps in the detection of buffer overflow and integer overflow vulnerabilities. From an optimization perspective, it gives compilers the chance to eliminate dead code, to improve register allocation and to perform static branch prediction. Given all these usages, it comes as no surprise that nowadays we have several different algorithms to implement range analysis.

Nevertheless, none of these algorithms has been designed to work specifically with distributed systems, even though they are applied onto them (41). The present work claims that such a shortcoming severely limits the capacity of a range analysis algorithm to provide useful information. When applying range inference onto a distributed system, developers usually perform range inference for each program independently. This *modus operandi* forces the static analysis tool to rely on assumptions that compromise its precision. In particular, every information that is received from the network is assumed to have very imprecise ranges, e.g., $[-\infty, +\infty]$. This restriction is unfortunate, because key information that a program manipulates comes indeed from its distributed peers. In this work we deal with this limitation.

The goal of this work is to describe a range analysis that has been customized to the distributed environment. To achieve this end, this novel range analysis algorithm was designed, implemented and tested. This algorithm combines into a common framework three different techniques which are already part of the programming languages literature: classic range analysis on the interval lattice, points-to analysis and communication links inference. To glue these technologies together, two new procedures were created, which shall be called *message segmentation* and *array content inference*. These two static analyses allows the discovery of contents of messages that are exchanged between modules that constitute a distributed system. In this way, when analyzing two modules that may exchange messages, information that is discovered in one program can be reused on its counter-part. This form of propagating information across different instances of a static analysis has not been described in previous literature, and is the key contribution of this work.

To validate the ideas, the proposed strategy have been implemented in the LLVM compiler (26). Section 4 shows that this new way to perform range analysis is substantially more precise than the classic methodology. Moreover, this technique improves the precision of range analysis by up to 38% in simple benchmarks. This strategy was also applied on a distributed application, e.g. `Talk`. In both cases, the new static analysis is able to augment classic range analysis with non-trivial information. As a result, 13% more bounds in `Talk` are inferred.

The remainder of this document is organized as follows: Chapter 2 discusses about the techniques used on this work; Chapter 3 shows with an example how X-RA works; Chapter 4 shows the results obtained through the X-RA analysis; Chapter 5 approaches some previous works related to the techniques used by this work; and Chapter 6 presents the final considerations.

Chapter 2

Background

A pair of programs that exchange messages is shown in Fig. 2.1. Conventional range analysis goes over each of these programs, without taking the other into consideration. As a result, when analyzing line 5 of the Server, a traditional implementation must assume that `t` can have any known range, e.g., `t` can assume any value within $[-\infty, +\infty]$. However, it is possible to see that the Client fills the first 10 positions of `buf` with values from 0 to 9. With this knowledge, one can see that the contents of `t` exists within $[0, 9]$. The analysis described in this work produces such information.

Client

```
1 int8_t  buf[50];
2 int32_t i, j;
3 for(i=0; i<10; i++)
4   buf[i] = i;
5 for(j=10; j<50; j++)
6   buf[j] = j;
7 send(buf);
8 return 0;
```

Server

```
1 int8_t  buf[50];
2 int8_t  data[45];
3 int32_t t;
4 fill_array(data);
5 recv(buf);
6 t = buf[0] * 2;
7 printf("%d\n", data[t]);
8 return 0;
```

Figure 2.1: Program Client and Program Server.

This work combines three previous results into one technique that increases the information of ranges that integer variables can assume during the execution of a program. These previous techniques are: (i) range analysis; (ii) points-to analysis; and (iii) communication links inference. The first technique, range analysis, has two purposes. First, it allows the estimation of values that integer variables may assume throughout the execution of a program. Second, when combined with points-to analysis, range analysis gives the chance to separate *messages* exchanged between processes into *segments*. Messages and segments are described in Section 2.4. Finally, communication links inference helps the task of finding in which parts of a program messages are produced and consumed. The rest of this section briefly discusses each one of these techniques.

2.1 Terminology

Throughout this paper, the term $system$ is used to denote a $Distributed\ System$; $program$ to denote the various modules that comprises a system; $link$ to denote a pair of SEND and RECV.

2.2 Integer Range Analysis

As mentioned before, range analysis is a compiler-related technique used to estimate the minimum and the maximum values that integer variables may assume during the execution of a program. The result of a range analysis is a function I, that maps variable names to *integer intervals*. Integer intervals are elements $[l, u]$ defined over the product lattice $\mathcal{Z}^2$, where $\mathcal{Z} = \mathbb{Z} \cup \{-\infty, +\infty\}$. We have the following ordering between elements of $\mathcal{Z}$: $-\infty < \ldots < -2 < -1 < 0 < 1 < 2 < \ldots + \infty$. For any $x > -\infty$ the following properties are defined:

$$\begin{array}{ll} x+\infty = \infty, x \neq -\infty & x-\infty = -\infty, x \neq +\infty \\ x \times \infty = \infty \text{ if } x > 0 & x \times \infty = -\infty \text{ if } x < 0 \\ 0 \times \infty = 0 & (-\infty) \times \infty = \text{ not defined} \end{array}$$

From the lattice $\mathcal{Z}$ the product lattice $\mathcal{Z}^2$ is defined, which is partially ordered by the subset relation $\sqsubseteq$. $\mathcal{Z}^2$ is defined as follows:

$$\mathcal{Z}^2 = \emptyset \cup \{[z_1, z_2] | \, z_1, z_2 \in \mathcal{Z}, \ z_1 \leq z_2, \ -\infty < z_2\}$$

The result of the range analysis classifies each variable into one of the three following categories: *Defined*, where both lower and upper values are determined. For example, [0, 10]; *Semi-defined*, where either the lower or the upper values could not be determined. For example, [0, ∞] or [−∞, 2]; *Undefined*, where neither the lower nor the upper values could be determined. For example, [-∞, ∞]. These categories are mutually exclusive since a variable cannot be, for example, defined and semi-defined at the same time. Although, for the matter of simplicity, the term *narrowed interval* is used when it was possible to increase the precision e.g. from *Undefined* to *Semi-defined* or *Defined* and from *Semi-defined* to *Defined*.

As an example, a conventional range analysis would return that $I(t) = [-\infty, +\infty]$ in the $Server$ process of Figure 2.1. This analysis would also return that $I(i) = [0,9]$ and $I(j) = [10,49]$ in the program $Client$ of the same figure.

The range analysis of integers on the interval lattice, as defined in this section, is one of the oldest problems in programming languages research. It was first proposed by (12). There are several different algorithms to solve range analysis. They differ on precision and efficiency. Usually the most efficient algorithms are also the less precise. This work uses the range analysis proposed by (40). (Other approaches to resolve this problem are cited on Section 5.)

2.3 Points-to Analysis

Points-to analysis is the problem of finding, for each pointer p in a program, the set of memory locations that can be addressed by p. The solution of a points-to analysis is a function P that maps pointer variables to a set formed by memory locations in the program heap, plus other variable names. Points-to analysis is usually more expensive than range analysis. To solve it, a number of *constraints* are extracted from the program text. These constraints exist in four varieties:

v = &u	$\{\text{u}\} \subseteq P(\text{v})$
v = u	$P(\text{u}) \subseteq P(\text{v})$
v = *u	$\forall t \in P(\text{u}), P(t) \subseteq P(\text{v})$
*v = u	$\forall t \in P(\text{v}), P(\text{u}) \subseteq P(\text{t})$

While iterating these equations, it is guaranteed that a fixed point is reached. This fixed point is a solution to points-to analysis. One shortcoming of traditional points-to analysis is its inability to separate an array into smaller chunks. For instance, in Fig. 2.2, variables a and b. However, b and c are not since b can point to the 20 first positions of a but c can only point to the 20 last positions of a. Nevertheless, typical implementations of range analysis, such as those present in mainstream compilers like `gcc` or LLVM, are not able to get this result.

Imprecision happens because points-to analysis does not use range information. In the example above, if one could associate the range $[0,19]$ with pointer b, and the range $[20,39]$ with pointer c, then it would be able to tell that these two pointers will never point to the same memory location (disambiguate the two pointers). By combining range analysis with points-to analysis in this work, it is possible to achieve this result.

```
int *a = (int*) malloc(40);
int *b, *c;
for (b = a; b < a + 20; b++) {
  putc(*b);
}
for (c = a + 20; c < a + 40; c++) {
  putc(*c);
}
```

Figure 2.2: Variables a and b are aliases, while b and c are not aliases.

Solving Points-to Analysis The problem of conservatively estimating the points-to relations in a C-like program has been exhaustively studied in the compiler literature (1; 20; 39; 47). For this work, points-to analysis available in LLVM was used to find an initial mapping P of pointers to locations. Then, this result was expanded using range analysis, as will be explained in Section 3. LLVM uses a suite of points-to analysis, listed below. The precision of queries is cumulative: if any of these four implementations is able to disambiguate two pointers, than they are marked as no aliases.

- *type-based*: C and C++ forbid aliasing between pointers of different types since C89/C++98. Thus, this analysis flags as no-alias pointers of different types;
- *global-refs-based*: relies on the fact that globals that do not have their address taken cannot alias anything;
- *basic*: uses a suite of heuristics, i.e. the stack does not alias the heap or globals, for instance;
- *scalar-evolution-based*: which tries to place bounds on arrays, and based on these bounds determine if they may overlap each other or not.
- *Dyck-CFL-based*: implements a context-free language (CFL) based context-insensitive alias analysis. This algorithm is implemented after Zheng *et al.* (62) and Zhang *et al.* (61).

2.4 Communication Links Inference

Communication Links Inference is the problem of determining communication links between distributed programs. In this context, the communication links inference problem receives two inputs: the text of two programs, P_1 and P_2, that communicate. A solution to this problem consists of a set of pairs C that relates to special nodes present in P_1 and P_2. These special nodes are called *senders* and *receivers*. Nodes in the former category emit messages, whereas nodes in the latter consume them. If C contains a pair (s, r), then it is known that sender s *may* issue a message that r consumes. However, if such a pair is not present in C, then the algorithm know, for sure, that s *cannot* send a message directly to r. Algorithm 1 describes the algorithm introduced by the work of (52).

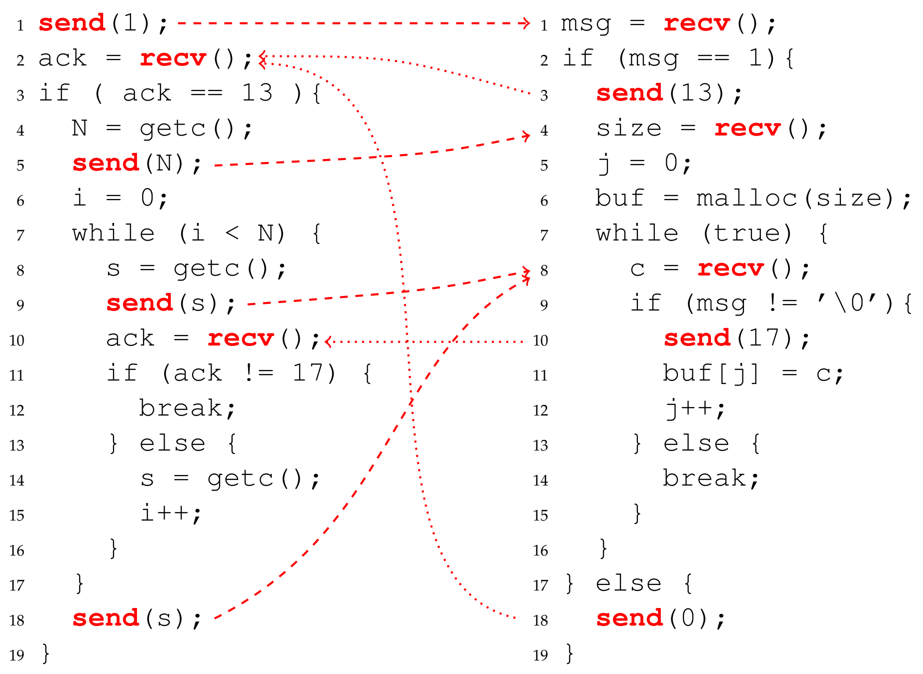
Algorithm 1: Elevator from (52)

```
Input: CFGs {$\mathscr{C}_1$,$\mathscr{C}_2$}, Send-Graphs {$\mathscr{S}_1$,$\mathscr{S}_2$} and Receive-Graphs {$\mathscr{R}_1$,$\mathscr{R}_2$}.
Output: a DCFG $\mathscr{D}$

▷ Set the SEND and RECV levels
foreach $G_i \in \{\mathscr{S}_1, \mathscr{S}_2\} \cup \{\mathscr{R}_1, \mathscr{R}_2\}$ do
    $n \leftarrow 0$
    $L_{G_i,n} \leftarrow \{root\}$
    ▷ While the new generated set $L_{G_i,n}$ is unique
    while $L_{G_i,n} \neq L_{G_i,0..n-1}$ do
        foreach vertex $v$ in $L_{G_i,n}$ do
            $S_{succs} \leftarrow$ successors of $v$
            $L_{G_i,n+1} \leftarrow L_{G_i,n+1} \cup S_{succs}$
        $n \leftarrow n+1$

▷ Link SENDs and RECVs of the same level
$\mathscr{D} \leftarrow \mathscr{C}_1 \cup \mathscr{C}_2$
for $k \leftarrow 1$ to $n$ do
    foreach $v_s \in L_{\mathscr{S}_1,k}$ and $v_r \in L_{\mathscr{R}_2,k}$ do
        add an edge from $v_s$ to $v_r$ in $\mathscr{D}$
    foreach $v_s \in L_{\mathscr{S}_2,k}$ and $v_r \in L_{\mathscr{R}_1,k}$ do
        add an edge from $v_s$ to $v_r$ in $\mathscr{D}$
```

Algorithm 1 takes as input the Control Flow Graph – CFG – of 2 programs and outputs a Distributed Control Flow Graph – DCFG. A CFG is a directed graph where each vertex is a instruction on the program and each edge represents a dependency between two instructions. The CFG represents the control flow of the program. The DCFG introduces edges between network functions that may interact, so the two previously CFGs become a single CFG that represents both programs (52).

Fig. 2.3 shows a solution of communication links inference. Each dotted line is a possible communication link, as inferred by the analysis.

This work adopts the method recently proposed by (52), which estimates communication links between programs by assigning “levels" to senders and receivers. Nodes of similar levels are paired up in a link. (See other approaches to resolve this problem on Section 5.)



```
send(1);
ack = recv();
if ( ack == 13 ){
  N = getc();
  send(N);
  i = 0;
  while (i < N) {
    s = getc();
    send(s);
    ack = recv();
    if (ack != 17) {
      break;
    } else {
      s = getc();
      i++;
    }
  }
  send(s);
}
```

```
msg = recv();
if (msg == 1){
  send(13);
  size = recv();
  j = 0;
  buf = malloc(size);
  while (true) {
    c = recv();
    if (msg != '\0'){
      send(17);
      buf[j] = c;
      j++;
    } else {
      break;
    }
  }
} else {
  send(0);
}
```

Figure 2.3: Communication Links Inference by (52)

Chapter 3

Distributed Range Analysis

The proposed solution aims at filling the gap between classic range analysis, performed over an isolated program, and a typical distributed system scenario in which programs are connected among themselves by means of message-exchange. During the initial research, no other work that explores such an ubiquitous characteristic of network software components were found.

To achieve this goal, this work relies on recently published works and combines their capabilities in a novel way that allows to derive a foundation for distributed range analysis. In particular, the single-program range analysis implementation from (40) and the distributed communication links inference implementation from (52) was used. Both of them are open-source and built on top of LLVM.

The steps of the proposed solution are demonstrated through the pair of programs in Fig. 2.1. The $Client$ program defines an 8-bit buffer buf with 50 positions and two 32-bit integer variables i and j. Later on, two consecutive for statements fill different portions of the buffer and eventually its content is sent over the network. Likewise, the $Server$ program defines two 8-bit buffers, buf and $data$, with 50 and 45 positions, respectively. It also defines a 32-bit integer variable t which is used to store the content of the buffer's first position, filled by the double of a value received over the network. Afterwards, the value of $data[t]$ is printed to the standard output.

Throughout the discussion, the reader is presented to the intermediate stages of the analysis and the important details are highlighted. Eventually, by the end of the chapter, it is expected to have made it clear how the data captured by the distributed range analysis enriches the information that would be obtained if programs were only analyzed in isolation. The Fig. 3.1 provides an overview of the analysis.

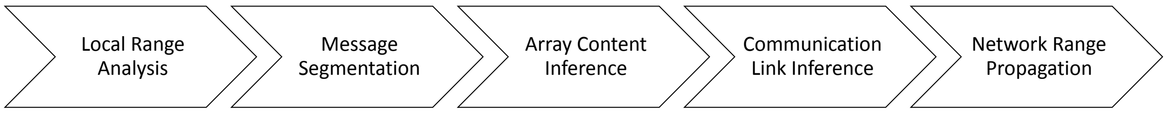


Figure 3.1: X-RA overview.

3.1 Local Range Analysis

The procedure starts by applying traditional range analysis to both programs. The values shown in table 3.1 will be used for Points-to Analysis, Array Content Inference, and Communication Links Inference stages. The notation $R(i)$ indicates the inclusive range of variable i within the program as a whole, while $R(i_{inside})$ indicates its range in the scope of the for statement. The reason for separately categorising these values is because only the ones used inside the loop will compose the actual range that is sent over the network. For instance, the last value assume for variable i, 10 in this case, does not influence further analyses. Similar reasoning and notation applies to variable j.

Client		**Server**	
$R(i)$	$[0,10]$	$R(t)$	$[-\infty,\infty]$
$R(i_{inside})$	$[0,9]$		
$R(j)$	$[10,50]$		
$R(j_{inside})$	$[10,49]$		

Table 3.1: Result of range analysis on the Client-Server example (Fig. 2.1).

Regarding the $Server$ program, the analyzed range of t, which is $[-\infty,\infty]$, is important to notice. Although correct, this result does not take advantage of the fact that it might be possible to deduce the values which will eventually be received by the $Server$, when paired with a particular client program for which range information is available.

3.2 Message Segmentation

This stage combines two techniques: points-to analysis and range analysis. First, points-to analysis will create *alias sets*, sets of pointers to the same underlying location. The goal is to identify as many *no alias* memory regions as possible. The larger is the evaluated memory region, the greater is the chance to having multiple pointers pointing to the same underlying storage. It is therefore essential to target the smallest memory regions to which is possible to obtain the least number of aliased pointers. At this point, range analysis becomes handy, since it produces information about the ranges, thus, allowing the fine-grained memory segmentation needed in order to match the content of the buffers.

The message segmentation results for the $Client$ and $Server$ programs can be seen in Fig. 3.2.

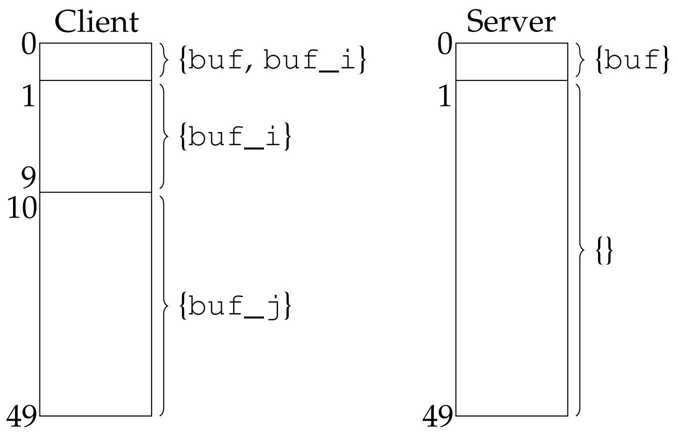


Figure 3.2: Visual representation of message segmentation.

On the $Client$ side, the memory have been segmented into three regions. The first region is related to the $base-pointer\ buf$ and always has the offset 0. The second region has an offset range of $[1,9]$ and has the pointer buf_i. The analogous reasoning applies to the third region. Regarding the $Server$ side, the memory has been segmented into two regions. In this case, only the first region has a pointer – buf – pointing to it. This is because only the first position of buf has been accessed.

Limitations. Although this technique is used to improve the granularity of messages, there are some cases in which the points-to analysis has precision loss. A typical case happens when the same buffer is used as source of data in a SEND and the destination of data in a RECV. This is explained because sometimes there is no information about the data being received simply due to absence of information on the originating program. There are representations such as Static Single Assignment form – SSA form (14) – that, for every assignment made to a variable, creates a new variable. However, this is not the case for points-to analysis.

3.3 Array Content Inference

Once memory is segmented into fine-grained regions and the pointer sets are known, it is possible to draw conclusion about the array contents. Specifically, the main interest lays on a pragmatic interpretation of a *load* instruction. At this point, it is fundamentally a matter of joining the results from the previous stages in order to tell which buffer indexes contain values from which range. For instance, the assignment inside the first *for* statement of the *Client* program comes from expression i, which has $R(i) = [0,9]$ and consequently delimits the content of the first ten positions of buffer *buf*. On the other hand, there is no information about the contents of the *Server*'s buffer, except for its first position, since the remaining ones are associated with an empty set of pointers. The overall result for the *Client* and *Server* programs is illustrated in Fig. 3.3.

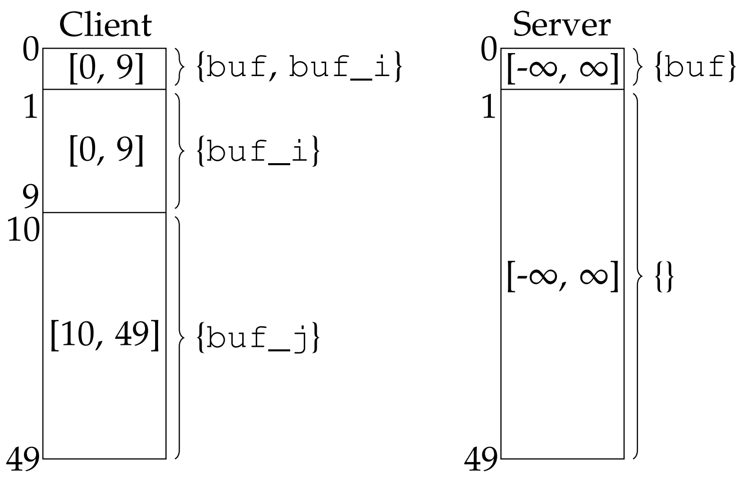


Figure 3.3: Array Content Inference on the running example (Fig. 2.1).

3.4 Communication Links Inference

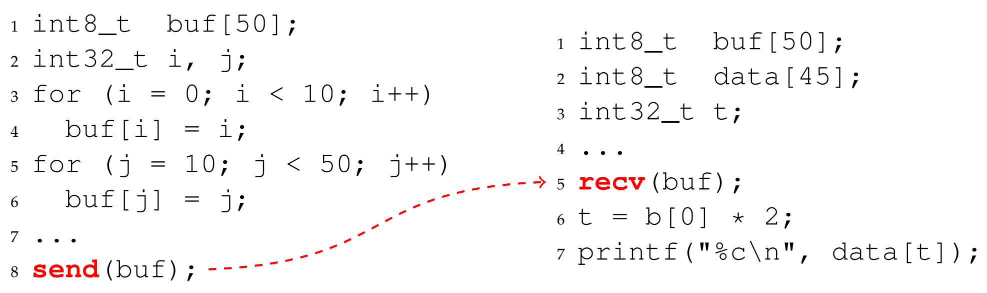
This step is where the work of (52) comes into play. It takes two programs that communicate with each other using message exchanges and creates a link between SENDS and RECVS that may communicate based on the Elevator technique (Algorithm 1).

With this technique on hand, it is possible to infer the communication links on the running example, which can be seen on Fig. 3.4.

The technique is able to infer the only communication link between the programs, thus enabling the next phase of the analysis, described on the following section.

3.5 Network Range Propagation

At this stage, the novel concept of *Network Range Propagation* is introduced, which takes the communication links between two programs and associates their range information.



```
int8_t  buf[50];
int32_t i, j;
for (i = 0; i < 10; i++)
  buf[i] = i;
for (j = 10; j < 50; j++)
  buf[j] = j;
...
send(buf);
```

```
int8_t  buf[50];
int8_t  data[45];
int32_t t;
...
recv(buf);
t = b[0] * 2;
printf("%c\n", data[t]);
```

Figure 3.4: Communication links by using the technique of Teixeira et al. (52).

This is the key insight of this work, which opens new opportunities for a variety of distributed analysis, in particular for the integer range of variables from connecting programs.

The ultimate goal is to identify the memory regions whose values can be sent over the network. More precisely, looking into the available *alias set*s and checking whether there is a buffer that is used as argument of a "send" function. On the other end, the reverse step is performed: for any buffer being used as argument of a "receive" function, a map of the *alias set*s is created to which it belongs to the range information available from the client side. This procedure will typically fill a gap of previously undefined ranges in the $Server$, given that an isolated analysis would not be able to infer the content of such a buffer. The final result is a comprehensive view of how values flow from one program to another in a distributed environment.

Back to the $Client$ and $Server$ programs, the only message-exchanging place is matched, in which the "send" function is SEND and the "receive" function is RECV. (This is done using the analysis proposed in (52) and can be seen in Fig. 3.4.) The variables of interest are buf from the $Client$ program, with range $[0,9]$ and buf from the $Server$ program, which up to this point have undefined range. Both situations can be seen in Fig. 3.3. The later observation is exactly what the distributed range analysis will fix: one can see that in this particular case, the content of the $Server$'s buf will exactly mirror the one from the $Client$, with range $[0,9]$. This is shown in Fig. 3.5.

It is worth to notice that the $Server$ program only accesses $b[0]$, whose value is assigned to t. Nevertheless, the information provided by X-RA is enough to cover all potential use of a particular memory region. In this case, the program use a single variable in order to keep the example simple and tangible.

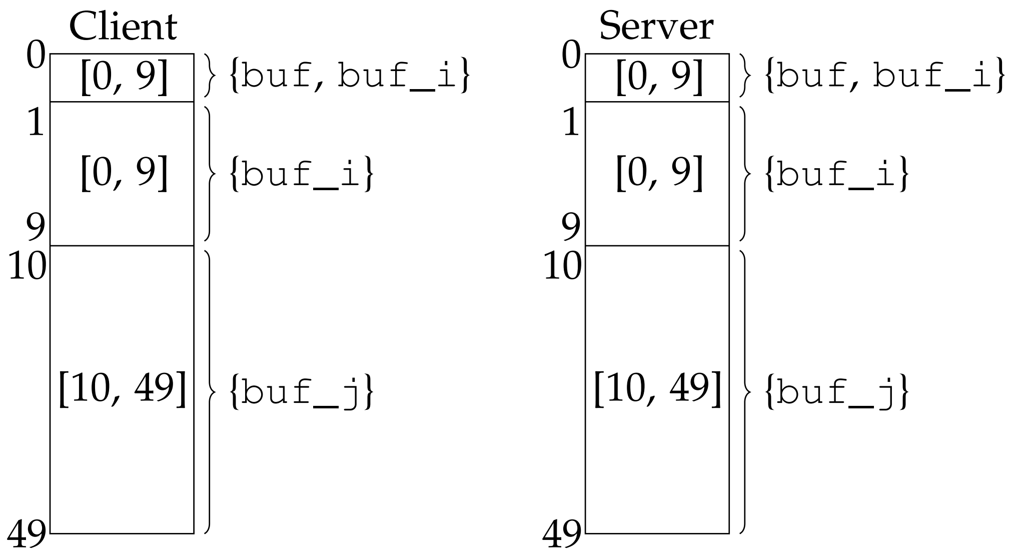


Figure 3.5: Network Range Propagation on the running example (Fig. 2.1).

Chapter 4

Results

In order to evaluate the solution, both tailor-made and real-world programs were used. The primary metric of interest is how the integer range values obtained from traditional range analysis are improved by the network range propagation technique. It is expected that certain variables will migrate from the *undefined* group to the *defined* one. The larger the number of such variables, the better is the result, as it indicates that the analysis generated information that was previously not available.

In addition to the aforementioned results, data regarding to memory usage and execution time of X-RA was collected. X-RA is implemented on top of LLVM and the tests were performed under the following scenario: Notebook Intel i7 and 8GB of RAM.

Tailor-made programs In this test set, four programs were used, properly adapted to exchange results of intermediate steps through the network: Bubble Sort, Quick sort, Matrix Multiplication and Selection Sort. Figure 4.1 shows the number of variables whose intervals were narrowed. Figure 4.2 shows the ratio of the number of defined variables obtained by the proposed analysis (X-RA) and the number obtained by the conventional analysis (RA).

The conservative nature of the whole analysis is worth noticing. This means that in certain cases in which the technique is able to gather range information for variables which are not actually used in the paired program – for instance, one that would correspond to a buffer index which is never accessed. These cases are not quantified. Nevertheless, they can be considered as *potential gain*, since programs frequently go under maintenance and a subsequent run of the analysis could eventually discover those values.

To evaluate the solution, distributed versions of four classic algorithms were created: Bubble Sort, Quick sort, Matrix Multiplication and Selection Sort. Each one of these new versions contains a server and a client. The server sends data to the client. The client

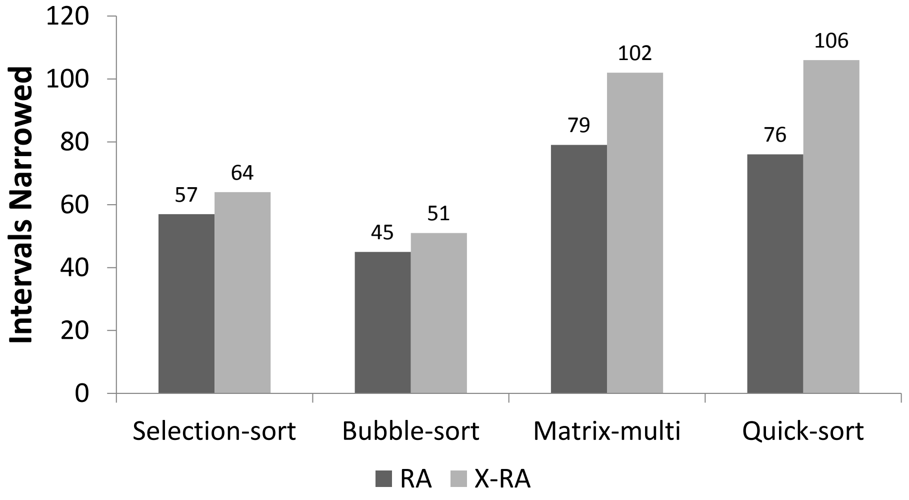


Figure 4.1: Number of narrowed intervals on tailor-made programs.

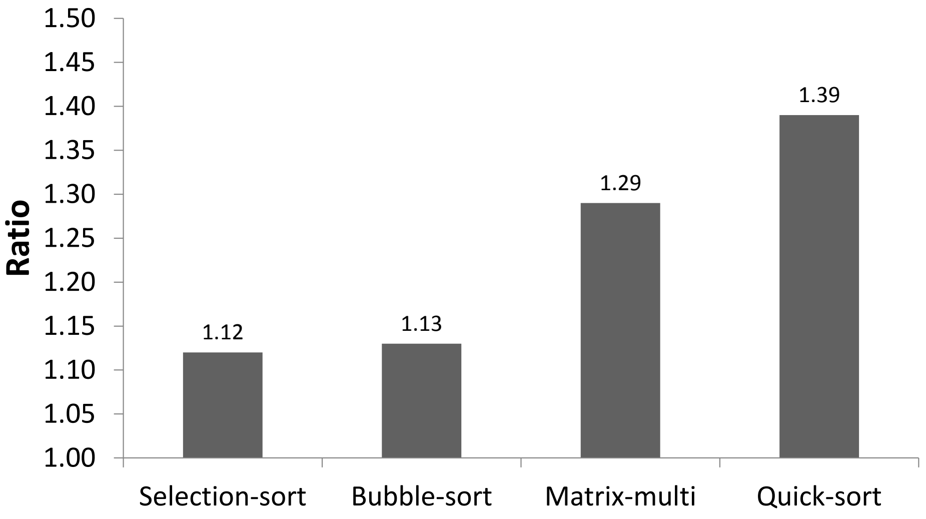


Figure 4.2: Ratio of narrowed intervals from conventional and distributed range analysis on tailor-made programs.

process these data, and sends it back to the server. For instance, to multiply a matrix, the server sends each line of each term matrix to the client, which, on its turn, performs the multiplication, sending the results back to the server.

Table 4.1 shows the results of intermediate steps of X-RA. $Segments$ means the number of memory regions the analysis discovered whose offset is well-defined e.g. Fig. 3.2. Column $Segments\ with\ defined\ content$ shows how many of the discovered regions have a range whose bounds were discovered e.g. Fig. 3.3 on the $Client$; yet on Fig. 3.3, the segments of the $Server$ would not be counted since the contents are undefined. Finally, the last column, $Communication\ links$ is related to the number of links discovered by the work of (52). The data on table 4.1 show that only about 10% of the segments have defined content but still it cause an improvement at the end of the proposed analysis.

Application	Segments	Segments with defined content	Communication links
Quick-sort	77	7	8
Matrix-multi	58	4	10
Bubble-sort	46	6	2
Selection-sort	53	6	2

Table 4.1: Data from the intermediate steps on tailor-made programs.

In order to make the understanding more tangible, a code snippet of both client and server of Bubble-sort can be seen on Fig.4.3.

```
1  int array_s[500], data_s[2], d;
2  int sock, val, c, n = 500;
3  for(c=0; c<n; c++) {
4   val = c * 2;
5   array_s[c] = val;
6  }
7  sock = sockfd(argv[1]);
8  for (c=0; c<(n-1); c++){
9   for (d=0; d<n-c-1; d++){
10   data_s[0] = array_s[d];
11   data_s[1] = array_s[d+1];
12   write(sock, &data_s,
13    2*sizeof(int));
14   ...
15  }
16 }
17 ...
```

[0,998]

```
1  int buf[2];
2  int v1, v2;
3  int sock = sockfd();
4  while(1){
5   read(sock, &buf,
6    2*sizeof(int));
7   v1 = buf[0];
8   v2 = buf[1];
9   ...
10 }
11 ...
```

Figure 4.3: Code snippet of the distributed version of Bubble-sort algorithm.

The WRITE on line 12 communicates with READ on line 5. The data inside the memory region pointer by `data_s` is sent and then stored inside the memory region pointed by `buf`. Since `data_s` is filled with the contents of `array_s`, filled on the `for` loop (lines 3 – 6), the data acquired on this loop can be propagated along the network. The improvement of this approach is that both `v1` and `v2` will have its contents laying within the range [0, 998].

Although the implementation does not seem to cause any extra overhead, its memory usage and execution time numbers can be seen in table 4.2.

The measurement of memory was done using the memory profiler tool `Valgrind`[1] and the execution time was obtained by the linux command named `time`[2]. Regarding the

[1]http://valgrind.org/
[2]http://man7.org/linux/man-pages/man1/time.1.html

		RA		X-RA	
Application	Instructions	Time (s)	Memory (MB)	Time (s)	Memory (MB)
Quick-sort	281	0.013	51.12	0.182	54.76
Matrix-multi	313	0.022	50.98	0.033	53.48
Bubble-sort	168	0.010	50.85	0.028	53.08
Selection-sort	177	0.012	51.12	0.021	53.10

Table 4.2: RA vs. X-RA: Time and memory used analyzing tailor-made programs.

latter, only the `sys` and `user` numbers were considered since they reflect the actual time used by the process.

Case study In order to validate the analysis on real-world programs, the program $Talk$, a messaging service commonly used in UNIX systems, was selected. The programs $talk$ and $talkd$, client and server, respectively, were compiled under LLVM 3.3. Combined, they have 2810 instructions, 4 SENDs and 6 RECVs. Table 4.3 shows the results of the analysis on $Talk$ application.

Application	Segments	Segments with defined content	SENDs	RECVs
talkd	401	13	5	3
talk	444	26	1	1

Table 4.3: Data from the intermediate steps on $Talk$.

One can see that segments with defined content represents less than 5% of the total of segments. This happens because the analysis rely on the programs themselves to provide information about how the buffers are filled. Another characteristic of the analysis is related to the Array Content Inference. If the programs do not use the same pointer for multiple purposes, e.g. SEND and RECV with the same pointer, the Array Content Inference is able to better delimit the range of values that can be stored in a given segment.

Figure 4.4 shows the number of intervals narrowed using RA and the proposed analysis (X-RA). Combined, X-RA narrowed 50 more intervals than the conventional approach. Figure 4.5 shows the ratio of gain compared to the baseline. Regarding $talk$, the novel approach obtained about 16% more narrowed intervals than the baseline. On the other hand, 8% more intervals on $talkd$ were narrowed using X-RA.

In order to estimate the resources used while analyzing a real-world program, time and memory consumption were measured, and can be seen in table 4.4.

The increase of 0.52 seconds and 11.73 megabytes when adopting X-RA is negligible since it is not performed on the target platform.

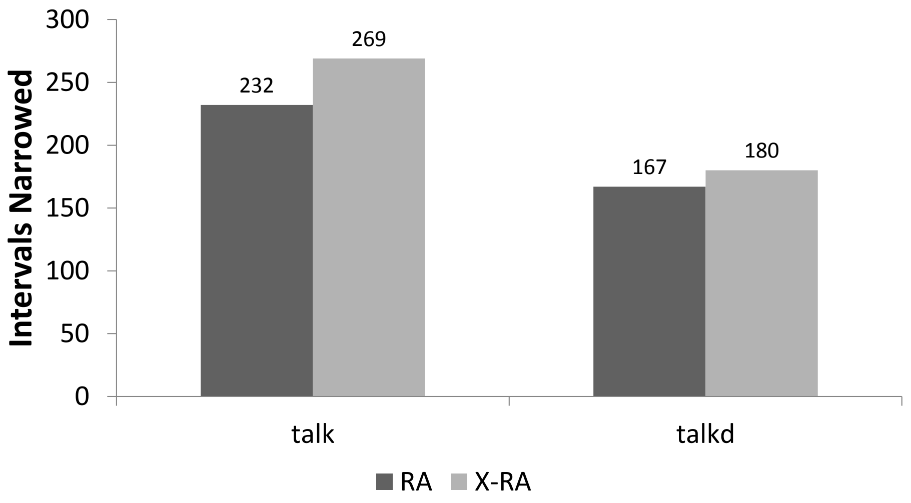


Figure 4.4: Number of narrowed intervals on *Talk*.

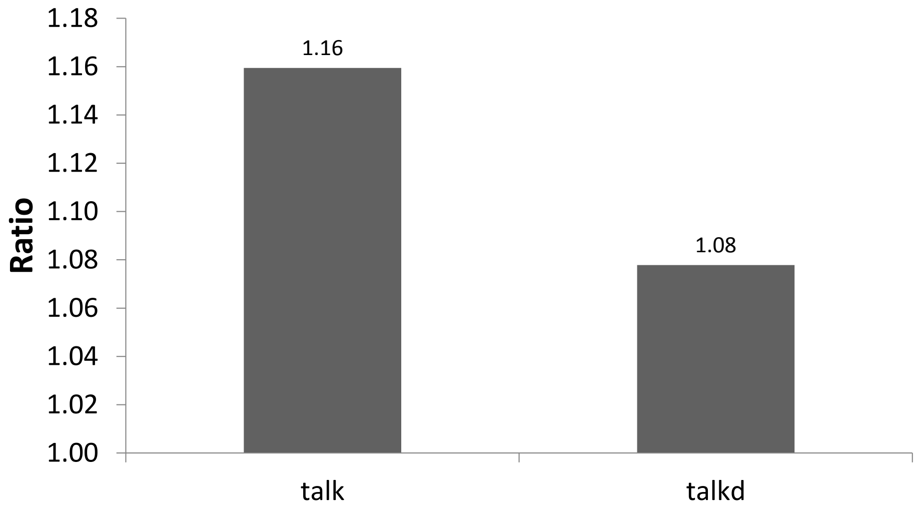


Figure 4.5: Ratio of narrowed intervals from conventional and distributed range analysis on *Talk*.

		RA		X-RA	
Application	Instructions	Time (s)	Memory (MB)	Time (s)	Memory (MB)
Talk	2810	1.36	54.24	1.88	65.97

Table 4.4: RA vs. X-RA: Time and memory used analyzing *Talk*.

Chapter 5

Related work

Range Analysis The range analysis of integers on the interval lattice was first proposed by (12). There are several different algorithms to solve range analysis. They differ on precision and efficiency. Usually the most efficient algorithms are also the less precise. Among the alternatives, there are the faster algorithms, such as Gawlitza's (17) or Su's (49). However, these algorithms are unable to handle comparisons between variables. In other words, they do not use the results of operations such as $x < y$ to narrow down the range of neither x nor y.

The most expensive algorithms (13; 32) are based on relational analysis which associate information not with a single variable name, but with sets of variables. The approach of Miné (32), for instance, finds, for each pair of variables, x and y, a constant c, such that $x - y < c$. This algorithm is quadratic on the number of variables. There are even more expensive approaches, such as Cousot's (13), which related not pairs, but general sets of variables. Since it is desired to provide a practical solution to the analysis of distributed programs, the solution design settled for an algorithm that is linear on the number of variables in the program text (40).

(40) present an algorithm that uses static range analysis to avoid Integer Overflow instrumentation whenever possible. Their range analysis contains novel techniques, such as prediction bounds to handle comparisons between variables. However, they do not handle the network interactions, and assume that the range of data coming from the network can not be inferred. The technique here presented shows that the range analysis precision can be improved analyzing a distributed system as whole because the information present in the messages exchanging over network can be seen.

The distributed range analysis proposed in this paper can be used of example to implement an efficient solution to counter Integer Overflow attacks in the Internet of Things. The literature has a good number of solutions to perform Integer Overflow detections in standalone programs (e.g. (60), (57) and (15)). Zhang *et al.* (60) have used static analysis to instrument integer operations in paths from a source to a sink to sanitize programs against integer overflow based vulnerabilities. However, they do not use any form of range analysis to limit the number of checks inserted in the transformed code. `IntScope` proposed by Wang *et al.* (57) combines symbolic execution and tainted flow analysis to detect integer overflow vulnerabilities. The authors have been able to use this tool to successfully identify many vulnerabilities in industrial quality software. Dietz *et al.* (15) have implemented a tool, IOC, that instruments the source code of C/C++ programs to detect integer overflows. The authors have used IOC to carry out a study about the occurrences of overflows in real-world programs, and have found that these events are very common. This work differs from these existing solutions since it targets distributed applications. In other words, none of these works is concerned about drawing information from the network's communication structure.

Analysis of Distributed Systems In order to statically analyze a system as whole, it is necessary to infer the communication links between programs. The literature describes a few works whose goal is to infer communication links between different processes. These techniques resort to different approaches to estimate links between programs. Some of them require the user to annotate code (38), others are fully automatic (8; 19; 52). Pascual and Hascoët (38) have defined a system of annotations which the user can employ to point out to the compiler implicit communication links in a distributed system. (8) finds a matching between sends and receives in an MPI program. It executes the program symbolically, separating processes by their IDs. His analysis is precise but the link inference may take too long to converge, as loops, for instance, may lead to the generation of many different symbolic sets. The technique shown in the work of (52) can identify the implicit links of interconnected programs. Such implicit links are discovered through the analysis of the network commands (e.g. SENDs and RECVs) present in the source code of programs. They show that is possible to reduce the number of Array Bound Checks against buffer overflow using the holistic view of system instead of analyzing the programs separately. They implement the solution into LLVM and tested Contiki OS applications. However, they do not handle the layout of the message exchanges in the network.

There are several proposals that analyze codes of distributed systems using symbolic execution, as (45), (28), (25) and (22). The symbolic execution of a program, allows the discovery of problems and automatic exploitation of execution paths. When an assertion fails, the test case can be stored so that it can be repeated. Both in (45) and (28), the developer must mark variables as symbolic and must create assertions about the system state. In these cases, the user must inform the static analyzer how messages are formed.

This makes necessary some knowledge of the logic of the program and its data structures. In this work, however, this task is more automatic, because the programmer indicates only which functions are involved in network communication and the proposed static analyzer learns how messages are formed without programmer intervention.

In order to turn more automatically the analysis and test of distributed protocol implementation, some recent works (e.g. (25) and (22)) have combined different techniques to analyze the distributed system as whole and interpret messages exchanging. The goal is to find bugs in protocol implementations that can be used to attack the network, for example. (25) propose a method to discover manipulation attacks in protocol implementations. The method propose combines static analysis, symbolic execution, dynamic analysis and concrete execution to find vulnerable code paths and emulate adversary's actions.(22) propose an automated adversarial testing of real-world implementations of wireless routing protocols. They extend the Turret (27) platform to differentiate the routing messages from data messages in order to detect bugs and attacks. This work proposes the analysis of the content of buffers used in message exchanging in a general way instead of a focus on a specific attack or bug. In this way, the techniques proposed in this work can be used to improve the precision or performance of protocol implementation analysis as proposed by (25) and (22).

Chapter 6

Conclusion

This work has presented a way to improve range analysis precision in distributed systems, which can be used to implement an efficient solution to counter Integer Overflow attacks in the Internet of Things. The **key insight** is to look at a distributed system as a single entity, rather than as multiple separate message-exchanging programs. With this holistic view, one can crosscheck integer information between different programs that converse through a network.

The novel techniques presented in this work are: (i) message segmentation, that joins points-to analysis and range analysis to segment fields in messages and (ii) network range propagation, that propagates the discovered integer ranges through the communication links between programs. The use of message exchange as source of new information allows a range analysis with more information about the ranges of integer variables. The combination of message exchange with network range propagation enables the static analyzer to be less conservative by providing more information, that further analyses could also benefit from.

To validate this claim, the solution was implemented on top of LLVM compiler (26) and evaluated using 4 tailor-made programs and a real-world program. The results show that X-RA is able to discover more information in comparison to the baseline algorithm, producing between 12% and 39% more narrowed intervals than the solution provided by (40). Although this work has presented a new way to perform range analysis on distributed systems, there are some limitations that reduce its precision.

From this work, two papers were published: (i) “Prevenção de Ataques em Sistemas Distribuídos via Análise de Intervalos **??**” has been published on the proceedings of XIV Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais (2014) – SBSeg and (ii)“Crosschecking Distributed Data to Detect Integer Overflow” (44) has been published on Latin America Transactions, IEEE. The next steps of this work are to obtain more real-world benchmarks and the submission of another paper gathering the most recent results obtained with X-RA.

Bibliography

[1] Andersen, L. O. (1994). *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen.

[2] Aranha, D. F., Karam, M. M., Miranda, A., and Scarel, F. (2012). Software vulnerabilities in the Brazilian voting machine. Tech Report.

[3] Ashton, K. (2009). That 'Internet of Things' Thing. *RFiD Journal*, 22:97--114.

[4] Atzori, L., Iera, A., and Morabito, G. (2010). The Internet of Things: A survey. *Computer Networks*, 54(15):2787--2805.

[5] Babar, S., Mahalle, P., Stango, A., Prasad, N., and Prasad, R. (2010). Proposed security model and threat taxonomy for the Internet of Things (IoT). In *Recent Trends in Network Security and Applications*. Springer.

[6] Bandyopadhyay, D. and Sen, J. (2011). Internet of things: Applications and challenges in technology and standardization. *Wireless Personal Communications*, 58(1):49--69.

[7] Bell, T. (1999). The concept of dynamic analysis. *SIGSOFT Softw. Eng. Notes*, 24(6):216--234. ISSN 0163-5948.

[8] Bronevetsky, G. (2009). Communication-sensitive static dataflow for parallel message passing applications. In *International Symposium on Code Generation and Optimization (CGO)*. IEEE.

[9] Brumley, D., Song, D. X., cker Chiueh, T., Johnson, R., and Lin, H. (2007). RICH: Automatically protecting against integer-based vulnerabilities. In *Network and Distributed System Security Symposium (NDSS)*. USENIX.

[10] Chandy, K. M. and Lamport, L. (1985). Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63--75.

[11] Chess, B. and West, J. (2007). *Secure Programming with Static Analysis.* Addison-Wesley Professional, first edition. ISBN 9780321424778.

[12] Cousot, P. and Cousot, R. (1977). Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Symposium on Principles of Programming Languages (POPL)*, pages 238--252. ACM.

[13] Cousot, P. and Halbwachs, N. (1978). Automatic discovery of linear restraints among variables of a program. In *Symposium on Principles of Programming Languages (POPL)*, pages 84--96. ACM.

[14] Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., and Zadeck, F. K. (1991). Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(4):451--490.

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

[16] Ernst, M. D. (2003). Static and dynamic analysis: Synergy and duality. In *WODA 2003: ICSE Workshop on Dynamic Analysis*, pages 24--27. Citeseer.

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

[18] Ghosh, S. (2014). *Distributed Systems: An Algorithmic Approach.* Chapman and Hall. ISBN 978-1466552975.

[19] Gopalakrishnan, G., Kirby, R. M., Siegel, S. F., Thakur, R., Gropp, W., Lusk, E. L., de Supinski, B. R., Schulz, M., and Bronevetsky, G. (2011). Formal analysis of mpi-based parallel programs. *Commun. ACM*, 54(12):82--91.

[20] Hardekopf, B. and Lin, C. (2007). The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. In *Programming Language Design and Implementation (PLDI)*, pages 290–299. ACM.

[21] Heer, T., Garcia-Morchon, O., Hummen, R., Keoh, S. L., Kumar, S. S., and Wehrle, K. (2011). Security challenges in the IP-based Internet of Things. *Springer Wireless Personal Communications*, 61(3):527--542.

[22] Hoque, M. E., Lee, H., Potharaju, R., Killian, C. E., and Nita-Rotaru, C. (2013). Adversarial testing of wireless routing implementations. In *Proceedings of the sixth ACM conference on Security and privacy in wireless and mobile networks*, pages 143--148. ACM.

[23] Huang, Y.-W., Yu, F., Hang, C., Tsai, C.-H., Lee, D.-T., and Kuo, S.-Y. (2004). Securing web application code by static analysis and runtime protection. In *Proceedings of the 13th International Conference on World Wide Web*, WWW '04, pages 40--52, New York, NY, USA. ACM.

[24] Jones, J. R. (2007). Estimating software vulnerabilities. *Security & Privacy, IEEE*, 5(4):28--32.

[25] Kothari, N., Mahajan, R., Millstein, T., Govindan, R., and Musuvathi, M. (2011). Finding protocol manipulation attacks. In *ACM SIGCOMM Computer Communication Review*, volume 41, pages 26--37. ACM.

[26] Lattner, C. and Adve, V. S. (2004). LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization (CGO)*. IEEE.

[27] Lee, H., Seibert, J., Killian, C. E., and Nita-Rotaru, C. (2012). Gatling: Automatic attack discovery in large-scale distributed systems. In *Network and Distributed System Security Symposium (NDSS)*.

[28] Li, P. and Regehr, J. (2010). T-check: bug finding for sensor networks. In *Proceedings of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks*, pages 174--185. ACM.

[29] Louridas, P. (2006). Static code analysis. *Software, IEEE*, 23(4):58--61.

[30] Mahlke, S., Ravindran, R., Schlansker, M., Schreiber, R., and Sherwood, T. (2001). Bitwidth cognizant architecture synthesis of custom hardware accelerators. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 20(11):1355--1371.

[31] McGraw, G. (2006). *Software security: building security in*, volume 1. Addison-Wesley Professional.

[32] Miné, A. (2006). The octagon abstract domain. *Higher Order Symbol. Comput.*, 19:31--100. ISSN 1388-3690.

[33] Misra, D. K. (1987). A quasi-static analysis of open-ended coaxial lines (short paper). *Microwave Theory and Techniques, IEEE Transactions on*, 35(10):925--928.

[34] Mock, M. (2003). Dynamic analysis from the bottom up. In *WODA 2003 ICSE Workshop on Dynamic Analysis*, page 13. Citeseer.

[35] Molnar, D., Li, X. C., and Wagner, D. A. (2009). Dynamic test generation to find integer bugs in x86 binary linux programs. In *Proceedings of the 18th conference on USENIX security symposium*, pages 67--82.

[36] Nethercote, N. (2004). *Dynamic binary analysis and instrumentation.* PhD thesis, PhD thesis, University of Cambridge.

[Paisante et al.] Paisante, V. M., Saggioro, L. F. Z., Rodrigues, R. E., Oliveira, L. B., and Pereira, F. M. Q. Prevenç ao de ataques em sistemas distribuıdos via análise de intervalos.

[38] Pascual, V. and Hascoët, L. (2012). Native handling of message-passing communication in data-flow analysis. In *Recent Advances in Algorithmic Differentiation*, volume 87 of *LNCSE*, pages 83–92. Springer Berlin Heidelberg.

[39] Pereira, F. M. Q. and Berlin, D. (2009). Wave propagation and deep propagation for pointer analysis. In *International Symposium on Code Generation and Optimization (CGO)*, pages 126–135. IEEE.

[40] 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 *International Symposium on Code Generation and Optimization (CGO)*, pages 1–11. ACM.

[41] Rugina, R. and Rinard, M. C. (2003). Pointer analysis for structured parallel programs. *TOPLAS*, 25(1):70--116.

[42] Rus, S., Rauchwerger, L., and Hoeflinger, J. (2003). Hybrid analysis: Static & dynamic memory reference analysis. *International Journal of Parallel Programming*, 31(4):251--283.

[43] Russo, A. and Sabelfeld, A. (2010). Dynamic vs. static flow-sensitive security analysis. In *CSF*, pages 186--199. IEEE Computer Society.

[44] Saggioro, L. F. Z., Paisante, V. M., Rodrigues, R. E., Oliveira, L. B., and Pereira, F. M. Q. (2015). Crosschecking distributed data to detect integer overflow. *Latin America Transactions, IEEE (Revista IEEE America Latina)*, 13(4):1083--1089.

[45] 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 *Proceedings of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks*, pages 186--196. ACM.

[46] Serebryany, K., Bruening, D., Potapenko, A., and Vyukov, D. (2012). Addresssanitizer: A fast address sanity checker. In *USENIX ATC*, volume 12.

[47] Steensgaard, B. (1996). Points-to analysis in almost linear time. In *Symposium on Principles of Programming Languages (POPL)*, pages 32–41.

[48] Stephenson, M., Babb, J., and Amarasinghe, S. (2000). Bidwidth analysis with application to silicon compilation. In *ACM SIGPLAN Notices*, volume 35, pages 108--120. ACM.

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

[50] Tan, L. and Wang, N. (2010). Future internet: The internet of things. In *Advanced Computer Theory and Engineering (ICACTE), 2010 3rd International Conference on*, volume 5, pages V5--376. IEEE.

[51] Tanenbaum, A. and Van Steen, M. (2007). *Distributed systems*. Pearson Prentice Hall.

[52] Teixeira, F. A., Machado, G. V., Pereira, F. M., Wong, H. C., Nogueira, J., and Oliveira, L. B. (2015). Siot: securing the internet of things through distributed system analysis. In *Proceedings of the 14th International Conference on Information Processing in Sensor Networks*, pages 310--321. ACM.

[53] Teixeira, F. A., Pereira, F., Vieira, G., Marcondes, P., Wong, H. C., Nogueira, J. M. S., and Oliveira, L. B. (2014). Siot–defendendo a internet das coisas contra exploits. *XXXII Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos*.

[54] Toth, T. and Kruegel, C. (2002). Accurate buffer overflow detection via abstract pay load execution. In *Recent Advances in Intrusion Detection*, pages 274--291. Springer.

[55] Wagner, D. and Dean, R. (2001). Intrusion detection via static analysis. In *Security and Privacy, 2001. S&P 2001. Proceedings. 2001 IEEE Symposium on*, pages 156--168. IEEE.

[56] Wagner, D., Foster, J. S., Brewer, E. A., and Aiken, A. (2000). A first step towards automated detection of buffer overrun vulnerabilities. In *Network and Distributed System Security Symposium (NDSS)*, pages 2000--02.

[57] Wang, T., Wei, T., Lin, Z., and Zou, W. (2009). Intscope: Automatically detecting integer overflow vulnerability in x86 binary using symbolic execution. In *Network and Distributed System Security Symposium (NDSS)*. Citeseer.

[58] Wang, X., Pan, C.-C., Liu, P., and Zhu, S. (2010). Sigfree: A signature-free buffer overflow attack blocker. *Dependable and Secure Computing, IEEE Transactions on*, 7(1):65--79.

[59] Warren, H. S. (2002). *Hacker's Delight.* Addison-Wesley Longman Publishing Co., Inc.

[60] Zhang, C., Wang, T., Wei, T., Chen, Y., and Zou, W. (2010). Intpatch: Automatically fix integer-overflow-to-buffer-overflow vulnerability at compile-time. In *Computer Security–ESORICS 2010*, pages 71--86. Springer.

[61] Zhang, Q., Lyu, M. R., Yuan, H., and Su, Z. (2013). Fast algorithms for dyck-cfl-reachability with applications to alias analysis. In *Programming Language Design and Implementation (PLDI)*, pages 435--446. ACM.

[62] Zheng, X. and Rugina, R. (2008). Demand-driven alias analysis for c. In *Symposium on Principles of Programming Languages (POPL)*, pages 197--208. ACM.

[63] Zoumboulakis, M. and Roussos, G. (2009). Efficient pattern detection in extremely resource-constrained devices. In *Sensor, Mesh and Ad Hoc Communications and Networks, 2009. SECON'09. 6th Annual IEEE Communications Society Conference on*, pages 1--9. IEEE.