PROTEGENDO SISTEMAS EMBARCADOS EM REDE ATRAVÉS DA ANÁLISE DE SISTEMAS DISTRIBUÍDOS

FERNANDO AUGUSTO TEIXEIRA

PROTEGENDO SISTEMAS EMBARCADOS EM REDE ATRAVÉS DA ANÁLISE DE SISTEMAS DISTRIBUÍDOS

Tese 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 Doutor em Ciência da Computação.

Orientador: José Marcos Silva Nogueira
Coorientador: Leonardo Barbosa e Oliveira

Belo Horizonte

Novembro de 2015

FERNANDO AUGUSTO TEIXEIRA

SECURING NETWORKED EMBEDDED SYSTEMS THROUGH DISTRIBUTED SYSTEMS ANALYSIS

Thesis presented to the Graduate Program in Computer Science of the Universidade Federal de Minas Gerais – Departamento de Ciência da Computação in partial fulfillment of the requirements for the degree of Doctor in Computer Science.

Advisor: José Marcos Silva Nogueira
Co-Advisor: Leonardo Barbosa e Oliveira

Belo Horizonte
November 2015

Teixeira, Fernando Augusto

T266s Securing Networked Embedded Systems through Distributed Systems Analysis / Fernando Augusto Teixeira. — Belo Horizonte, 2015
xxi, 85 f. : il. ; 29cm

Tese (doutorado) — Universidade Federal de Minas Gerais – Departamento de Ciência da Computação
Orientador: José Marcos Silva Nogueira

1. Computação – Teses. 2. Sistemas Distribuídos – Teses. 3. Redes de Computadores – Medidas de Segurança – Teses. 4. Sistemas embutidos de computador. I. 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

Securing Networked Embedded Systems Through Distributed Systems Analysis

FERNANDO AUGUSTO TEIXEIRA

Tese defendida e aprovada pela banca examinadora constituída pelos Senhores:

[illegible]

PROF. JOSÉ MARCOS SILVA NOGUEIRA - Orientador
Departamento de Ciência da Computação - UFMG

[illegible]

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

[illegible]

PROF. EDUARDO LUZEIRO FEITOSA
Instituto de Computação - UFAM

Fernando

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

[illegible]

PROFA. FLÁVIA COIMBRA DELICATO
Instituto de Matemática - UFRJ

[illegible]

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

Belo Horizonte, 19 de novembro de 2015.

Resumo

Novas tecnologias como a Internet das Coisas e Cloud Computing estão aumentando a importância de técnicas para analisar e entender os Sistemas Embarcados com acesso à Rede. Essa crescente relevância aumenta a necessidade de ferramentas capazes de fornecer aos usuários sistemas corretos, confiáveis e seguros.

Neste trabalho, nós defendemos que as abordagens tradicionais para compilar e analisar os programas dos sistemas distribuídos não são expressivas o suficiente para enfrentar este desafio. Tal limitação decorre do fato de que atualmente os programadores precisam compilar e analizar cada programa de um sistema distribuído individualmente; falta, portanto, uma visão inter-programa do sistema.

Como solução para este problema, apresentamos o *Distributed Systems Analysis* (DSA), um arcabouço para analisar sistemas em rede durante a compilação. Nossa ideia chave é olhar para um sistema distribuído como uma única entidade e não como programas separados que trocam mensagens. Ao fazer isso, nós podemos cruzar informações inferidas a partir de diferentes programas para aumentar a precisão de análises estáticas tradicionais. Para construir uma visão inter-programa de um sistema distribuído nós introduzimos um novo algoritmo que descobre ligações entre dois programas automaticamente. Essas ligações nos permitem construir uma visão das comunicações entre programas, um conhecimento que pode ser transmitido para uma ferramenta de análise estática tradicional. Provamos que nosso algoritmo sempre termina e que modela corretamente a semântica do sistema distribuído.

Para validar a nossa solução, nós a implementamos como um arcabouço de análise estática no topo do compilador LLVM. Este arcabouço possui diferentes aplicações e nós implementamos duas destas aplicações. A primeira, uma ferramenta que chamamos de SIoT, usa o arcabouço proposto para proteger seis aplicações do CointiOS contra ataques de estouro de buffer. SIoT produz código tão seguro quanto código garantido por análises tradicionais, mas nossos binários são em média 18% mais eficientes em termos de energia. A outra, uma ferramenta que chamamos de DistViewer, usa os grafos gerados pelo arcabouço para fornecer uma visão inter-programa do sistema distribuído.

Abstract

New technologies such as the Internet of Things and Cloud Computing are increasing the importance of techniques to analyze and understand Networked Embedded Systems. This growing importance calls for tools able to provide users with correct, reliable and secure systems.

In this work, we claim that traditional approaches to compile and analyze programs of distributed systems are not expressive enough to address this challenge. Such limitation stems from the fact that nowadays the programmer needs to compile and analyze each program of a distributed system individually; hence, missing a inter-program view of said system.

As a solution to this problem, we present DSA – short for Distributed Systems Analysis, a framework to analyze networked systems at compile phase. Our key insight is to look at a distributed system as a single entity, and not as separate programs that exchange messages. By doing so, we can crosscheck information inferred from different programs to increase the precision of traditional static analyses.

To construct this global view of a distributed system we introduce a novel algorithm that discovers inter-program links automatically. Such links lets us build a inter-program view of the distributed system, a knowledge that we can thus forward to a traditional static analysis tool. We prove that our algorithm always terminates and that it correctly models the semantics of a distributed system.

To validate our solution, we implemented it as a static analysis framework on top of the LLVM compiler. This framework has different applications, and we implement two such applications. The first, a tool that we call SIoT, uses the proposed framework to secure six ContikiOS applications against buffer overflow attacks. SIoT produces code that is as safe as code secured by more traditional analyses; however, our binaries are on average 18% more energy-efficient. The other, a tool that we call DistViewer, uses the graphs produced by the framework to furnish a inter-program visualization of the distributed system.

List of Figures

List of Tables

Acronyms

ABC	Array-Bound Checks.
BOF	Buffer Overflow.
CFG	Control Flow Graph.
CoAP	Constrained Application Protocol.
DBMS	Database Management System.
DCFG	Distributed Control Flow Graph.
DDG	Distributed Dependence Graph.
DG	Dependence Graph.
DistViewer	Distributed System Code Viewer.
DS	Distributed Systems.
DSA	Distributed Systems Analysis.
eCoSoC	Energy-Efficient Instrumentation to Secure Systems-on-a-Chip Devices.
ED	Embedded Devices.
ES	Embedded Systems.
H2T	Human-to-Thing.
ICFG	Interprocedural Control Flow Graph.
IOF	Integer Overflow.
IoT	Internet of Things.

LLN	Lowpower and Lossy Networks.
MPI	Message Passing Interface.
MPI-ICFG	MPI Interprocedural CFG.
NED	Networked Embedded Devices.
NES	Networked Embedded Systems.
ROLL	Routing Over Low power and Lossy.
RPL	IPv6 Routing Protocol for Lowpower and Lossy Networks.
SIOT	Securing Internet of Things.
T2T	Thing-to-Thing.
TFA	Tainted Flow Analysis.
VANET	Vehicular Ad Hoc Network.
WSN	Wireless Sensor Network.

Contents

Chapter 1

Introduction

The emergence of the Internet of Things (IoT) and Cloud Computing has increased the importance of Networked Embedded Systems (NES) [Ghosh, 2014]. This raise in importance is due to the fact that, more than ever, the everyday person and "things" are surrounded by NES in a most varied set of devices, which perform a very diverse list of services [Borgia, 2014; Li et al., 2014; Macedo et al., 2012]. However, programming these systems is more challenging, due, not only to their shear volume, but also to this diversity. Above all, software embedded in appliances, cars and sensors scattered throughout cities, running in hardware of different capacities, and subject to very different natural conditions [Engoulou et al., 2014; Jabeen and Nawaz, 2015; Atzori et al., 2010; Teixeira et al., 2014a].

An Embedded Systems (ES) is a combination of computer hardware and software that is specifically designed for a particular function [Marwedel, 2010]. Industrial machines, automobiles, medical equipment, cameras, household appliances, airplanes, vending machines and toys (as well as the more obvious cellular phone and tablets) are among the myriad possible hosts of an embedded system. Networked Embedded Systems are distributed systems of Embedded Devices (ED) and have the potential to change radically the way people interact with their environment by linking together a range of devices and sensors that will allow information to be collected, shared, and processed in the unprecedented ways [Marwedel, 2010].

The ability to connect embedded devices with limited CPU, memory and power resources means contributed to leverage the development of the IoT paradigm [Atzori et al., 2010]. IoT consists of a world of physical objects embedded with sensors and actuators linked by wireless networks which communicate using the Internet, shaping a network of smart objects able to capture environmental variables, and to react to external stimuli [Delicato et al., 2013]. IoT promotes the connection of the virtual and physical worlds by extending the existing interaction between men and machines provided by the Internet to new dimensions of Human-to-Thing (H2T) and Thing-to-Thing (T2T) communications [Pires et al., 2014]. According to Gartner, Inc. (a technology research and advisory corporation), there will be nearly 26 billion devices on the IoT by 2020 [Rivera and van der Meulen, 2013]. Besides, ABI Research estimates that more than 30 billion devices will be connected to the IoT by 2020 [Intelligence, 2013].

Ensuring the security of such NES is a problem of increasing relevance [Kermani et al., 2013; Ravi et al., 2004; Wu and Mueller, 2011; Heer et al., 2011]. In fact, DARPA has elected security the central point in its Cyber Grand Challenges call[1].

Two main factors, however, make NES security even more critical. First, things act as bridges between user's physical and cyber worlds, and their exploitation can potentially have more impact on users' daily lives. Secondly, the nature of things makes the scale of attacks even larger. For instance, a botnet made up by one hundred thousand things was recently launched[2]. Such bot had targets like home routers, set-top boxes, smart TVs, and smart appliances.

NES faces a plethora of security problems [Kermani et al., 2013; Heer et al., 2011; Ravi et al., 2004]. It suffers from the same security issues as traditional Internet-based and/or wireless systems, including jamming, spoofing, replay, and eavesdropping [Heer et al., 2011; Babar et al., 2010]. In addition, it is more prone (as compared to traditional systems) to other issues such as out-of-bound memory accesses.

NES increased vulnerability to out-of-bound memory accesses is due to a few factors. Notably, ES devices costs must be kept as low as possible, and to meet this requirement, they are usually endowed with the least amount of resources necessary to accomplish their duties. Accordingly, applications for ES are commonly developed using lightweight languages such as C. On the one hand, applications may run more efficiently, allowing for better end-user response time and slower power depletion. On the other hand, the use of C in code development also makes applications more vulnerable to attacks.

Language C is an inherently unsafe language [Chess and West, 2007]. For instance, its semantics allows out-of-bound memory accesses. It is worth recalling that an array access in C or C++, such as $a[i]$, is safe if the variable i is greater than or equal to zero, and its value is less than the maximum addressable offset starting from the base pointer a. This type of accesses are dangerous because they give room to buffer overflow attacks [Cowan et al., 2000].

[1] `http://cgc.darpa.mil`
[2] `http://slashdot.org/topic/datacenter/100k-thingbot-net-shows-risk-of-smart-devices/`

```

#define BUFSIZE 512

int main() {
  int buffer[BUFSIZE];
  int a;
  int  i,j;
  ...
  for(i;i<j;i++){
     ...

     buffer[i] = a;
     ...
  }
  ...
}
```

```

#define BUFSIZE 512

int main() {
  int buffer[BUFSIZE];
  int a;
  int  i,j;
  ...
  for(i;i<j;i++){
     ...
if((i >= 0)&&(i < BUFSIZE))
        buffer[i] = a;
     ...
  }
  ...
}
```

Figure 1.1. Vulnerable (left-hand-side) and ABC protected (right-hand-side) C code.

A Buffer Overflow takes place whenever a system allows data to be accessed out of the bounds of an array. The Morris worm[3] and the Heartbleed flaw[4] illustrate how effective these attacks can be. Back in 1988, the former made use of the then-novel technique of buffer over-write to compromise around 10% of computers connected to the Internet. The latter exploited a buffer over-read vulnerability, compromising half a million web servers.

Much work has already been done to turn C into a safer language (e.g. SAFE-Code [Dhurjati et al., 1996] and AddressSanitizer [Serebryany et al., 2012]). Many existing proposals resort to Array-Bound Checks (ABC), which are tests done at run-time to ensure that a particular array access is safe. For instance, see code without ABC in Fig. 1.1 (right) – vulnerable version – versus the other one with ABC (left) – ABC version. The "ABC version" checks if an index is within the bounds of the array before accesses the memory (*buffer[i]* $=$ *a*). If the index i is less than zero or greater than or equal to *BUFSIZE*, in the vulnerable version occurs a memory access violation, while in the ABC version this problem not happen.

[3]`http://www.cs.indiana.edu/docproject/zen/zen-1.0_10.html#SEC91`
[4]`http://heartbleed.com/`

These proposals work in a two-pass fashion. They first scan programs' assembly representation to find code snippets containing vulnerabilities; in a second step, they return to the potential vulnerabilities and insert ABCs. While effective in preventing out-out-bound memory accesses from taking place, these proposals impose a significant overhead on compiled programs, and are thus inadequate *as-is* to NES. As an example, AddressSanitizer is known to slowdown programs by over 70%, and to increase their memory consumption by over 200%. It is therefore paramount to develop more efficient techniques that can be used to protect NES.

1.1 The goal

The goals of this work are:

- Propose a framework for static analysis of distributed systems programs. We call our framework Distributed Systems Analysis (DSA).

- Come up with a Buffer Overflow prevention mechanism tailor-made for NES, secure against Buffer Overflow (BOF) and light enough to be run in battery-powered devices.

To achieve this end, we have designed, implemented and tested an algorithm to analyze the source code of NES. Our solution combines into a common framework different techniques which are already part of the programming languages literature (e.g. classic control flow graphs [Allen, 1970], points-to analyses [Andersen, 1994], dependency graphs [Ottenstein et al., 1990]) and a new algorithm to find the communication links between programs.

Our key insight is to look at a distributed system source code as a single entity, rather than as multiple separate message-exchanging programs. Traditional approaches see a distributed system as different programs. For example, they analyze a client and its respective server programs separately. In this way, all information received from network must be considered undefined. However, if we analyze the inter-program relationships we can use the information of a program P_A to improve the analysis of another program P_B.(We discuss a detailed example in the Section 1.3).

Using a novel algorithm, we can infer the communication links between different programs that exchange messages through a network. This knowledge lets us model how data flows across distributed programs; hence, it gives us a inter-program view of the NES. Such view can be coupled with traditional static analysis tools to improve their precision and to produce program slicing that helps developers to visualize and analyze the NES.

To validate our claims, we have designed and implemented our framework as an extension of the LLVM compiler. Using our framework, we have constructed two new tools: Securing Internet of Things (SIOT) and Distributed System Code Viewer (DistViewer).

SIOT uses the proposed framework to implement a Distributed Tainted Flow Analysis (TFA) which is broad enough to secure ContikiOS [Dunkels et al., 2004] applications against buffer overflow attacks in a more energy-efficient manner. More specifically, we applied TFA [Balzarotti et al., 2008] on the model we proposed, and sanitized C programs against out-of-bound memory accesses. TFA tracks potentially malicious data (i.e., data that can be influenced by attackers) flows across the program. Memory indexed by tainted data can then be guarded against invalid access during runtime using ABCs. Because the analysis has a inter-program view of the system, we can reduce the number of tainted flows by cross-checking the dependencies between data received by network and user inputs. Therefore our approach produces a smaller number of false-positives than if each module of the system were analyzed individually. This extra precision yields a smaller runtime overhead.

DistViewer builds programming slices of distributed systems. The DistViewer allows the programmer to use the DSA as a tool to compile the programs of a distributed system and receive, as output, visions in the form of graphs that summarize the interactions via network. Therefore, through the DistViewer, the programmer can visually check if there are bugs in the communication protocol. For example, the programmer can visually check if there are sending messages without a corresponding receiving command or vice versa (sends without receives or receives without sends). Generally, this type of defect is only discovered during the testing phase or after the system is deployed. Using DistViewer the programmer can detect this type of defect during the system build, reducing time and cost to remove this type of bug.

DistViewer and SIOT are available on line as an extension of DSA[5].

1.2 Contributions

This work brings forth both theoretical and practical contributions. On the theoretical side, we propose a way to model distributed systems as single entities. More specifically:

[5]`http://cuda.dcc.ufmg.br/siot/`

- We propose an extension to the standard CFG [Allen, 1970], called Distributed Control Flow Graph (DCFG), that is expressive enough to model the control flow spanning multiple programs that communicate over a network.
- We propose an algorithm that infers communication links between different programs from a distributed system, and prove that the algorithm (i) never misses possible communication paths between programs; and (ii) always reaches a fixed point, and hence always terminates.
- We propose the DSA as static analysis framework that has been customized to the distributed environment. We define the DSA framework architecture and highlight its extensions points as interfaces.

On the practical side, we have implemented our algorithm, and showed that it can protect NES against buffer overflows, and can do so more efficiently than traditional approaches. More specifically:

- We implemented our algorithm, our framework and its companion distributed tainted flow analysis in the LLVM compiler. (Our implementation is publicly available[6].)
- We applied this analysis to six pair of applications present in the ContikiOS [Dunkels et al., 2004] (an operational system for IoT). The results show that our proposal is 18% more energy-efficient than a baseline solution.
- We implemented the DistViewer which uses the graphs produced by the framework to furnish a inter-program visualization of the distributed system.

1.3 Overview of our Solution

In this section, we present an overview of our proposal using the SIOT framework instance as example.

Many of existing proposals to secure code against Buffer Overflow attacks have three phases: First, at compiling time, they find the tainted flows, i. e., buffers reachable from untrusted sources (Fig. 1.2). They then insert array bounds checks – ABC – to guard buffers. And, finally, if an ABC is not satisfied at execution time, they abort the program.

[6]`https://code.google.com/p/ecosoc`

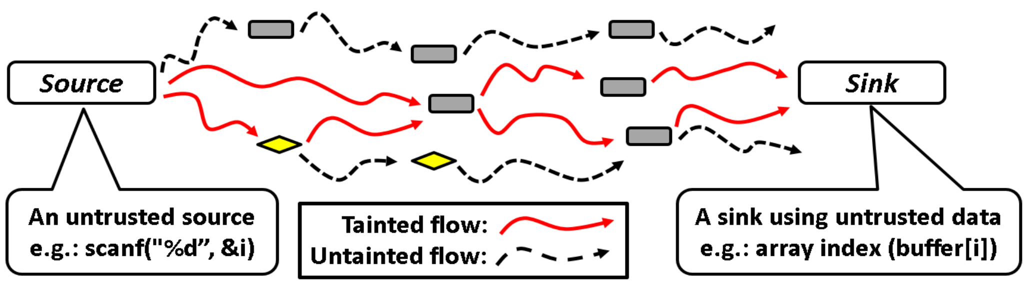


Figure 1.2. Tainted flow: paths between untrusted sources and sinks.

The main difference between our approach and traditional proposals are that we see the programs of a distributed system as a single entity. We then can reduce the number of user sources, the number of reachable arrays and the number of ABC. Traditional approaches see a distributed system as different programs. For example, they analyze a client and its respective server programs separately. Their list of sources include conventional sources, like `scanf`, but also network functions, like `receives`. Therefore there are more sources than necessary. In consequence, many reachable arrays are protected with ABC and then more energy overhead is introduced in the application.

SIOT sees all programs of a distributed system as a single system. Network functions are not system sources anymore but connections points between system's programs[7]. As we mentioned above, existing proposals focus on each program separately. We call existing proposals by Baseline approach.

To illustrate this point we have an Echo Distributed System in Fig. 1.3. It is composed by Echo Client (Fig. 1.3a) and Echo Server (Fig. 1.3b). Echo Client try to start the communication with the Echo Server. If the communication is successful the Echo Client gets a character from user, sends to the Echo Server and wait for an acknowledgment. The Echo Server receives the character, shows it up at screen and send back an acknowledgment message. When the Echo Client receives the acknowledgment, it gets one new character and repeats the process. This routine continues until the user puts an enter to finalize the message. In this way, all caracteres input by the user are showed at server as an "echo".

In Baseline approach (Fig. 1.4), we have seven input sources: two `getc` and five `receives`. However if we look at the system as single entity, we can reduce the input sources because the receive function is not a external input anymore. We then have

[7]We assume messages exchanged are authenticated (we discuss this in Section 2.4.2)

```
send(1);
ack = recv();
if ( ack == 1 ){
  s = getc();
  while (s != '\0') {
    send(s);
    ack = recv();
    if (ack != 1) {
      break;
    } else {
      s = getc();
    }
  }
  send(s);
}
```

(a)

```
msg = recv();
if (msg == 1){
    send(1);
    do {
        msg = recv();
        putc(msg);
        if (msg != '\0')
            send(1);
        else
            break;
    }
} else {
    send(0);
}
```

(b)

Figure 1.3. Echo application's programs: (a) Echo Client and (b) Echo Server .

only two input sources – the `getc` functions (Fig. 1.5).

To construct this inter-program vision of distributed system we propose an algorithm that infers communication links between different programs from a distributed system (Algorithm Elevator – Chapter 3). With this algorithm we can extend some traditional standard compiler data structure that are not expressive enough to model the control flow spanning multiple programs that communicate over a network (Chapter 2 and Chapter 3).

In order to validate our proposal we have: (i) proved that the Elevator algorithm never misses possible communication paths between programs and always reaches a fixed point, and hence always terminates (Chapter 3); (ii) implemented our algorithm and its companion distributed tainted flow analysis in the LLVM compiler[8] (Chapter 4); and (iii) applied this analysis on six applications present in ContikiOS [Dunkels et al., 2004]. The results show that our proposal is 18% more energy-efficient than existing solutions (Chapter 5).

[8]`https://code.google.com/p/ecosoc`

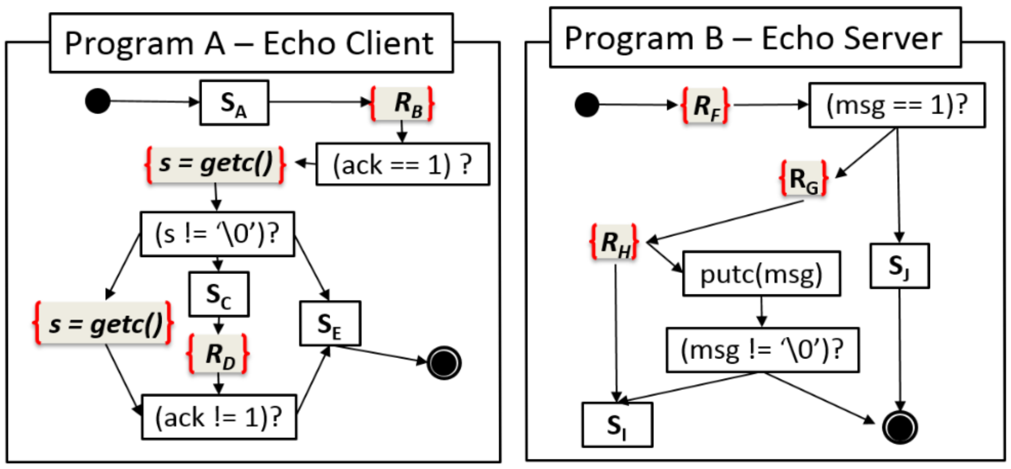


Figure 1.4. Traditional approach: seven input sources (between {}) – `getc` and `receives`

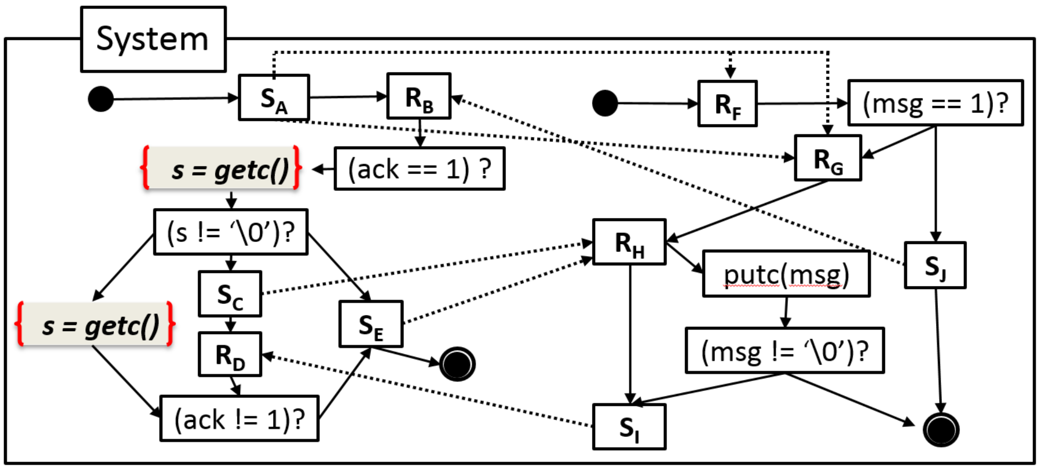


Figure 1.5. SIoT approach: two input sources (between {}) – `getc`. The `receives` are connection points between programs and not external sources.

1.4 Organization

The remainder of this work is structured as follows.

In **Chapter 2** we present the fundamental concepts of code analysis. We then introduce some compiler fundamental concepts used as basis for this work, starting with a brief description of language-base techniques to analyze code. After that, we describe data structures used to code analysis and show their limitations to analyze more than one program.

In **Chapter 3** we describe the DSA key insight, which is to look at a distributed system as a single entity, and not as separate programs that exchange messages. To construct such view of a distributed system, we introduce the concepts of Distributed Control Flow Graph and Distributed Dependence Graph, two data structures that would enable us to model control flow and data flow across different programs in a distributed system. We then introduce a novel algorithm that discovers inter-program links efficiently. Such links let us build a inter-program view of the distributed system, a knowledge that we can thus forward to a traditional tool. We also prove that our algorithm always terminates and that it correctly models the semantics of a distributed system.

In **Chapter 4** we describe the architecture and the implementation details of our framework constructed on top of the LLVM compiler, and of the two applications that uses it. We discuss the architecture of the framework core and of each instance (SIOT and DistViewer). We also describe a second one instance – DistViewer. DistViewer uses the graphs exported by our framework to generate a code view (programming slices) that highlights the part of code that has dependency of network.

In **Chapter 5** we show that to validate our solution, we have applied SIOT and DistViewer on six ContikiOS [Dunkels et al., 2004] applications. SIoT secures six ContikiOS applications against buffer overflow attacks. SIoT produces code that is as safe as code secured by more traditional analyses; however, our binaries are on average 18% more energy-efficient. In Chapter 5 also show that DistViewer can provide graphs of sending and receiving messages of the programs and the interrelationship between them, and provide program slices of distributed system with size equivalent to 5% of the original program.

In **Chapter 6** we present a review of related works about distributed system code analysis. First, we explain the adopted literature review methodology. We then review and characterize a set of works that verify memory access in distributed systems. Finally, we discuss works about link inference that is an important step to do an inter-program analysis of distributed system.

We conclude in **Chapter 7** and present some future works that can be derived from this thesis.

We also present in Appendix A a list of our publications.

Chapter 2

Background and Assumptions

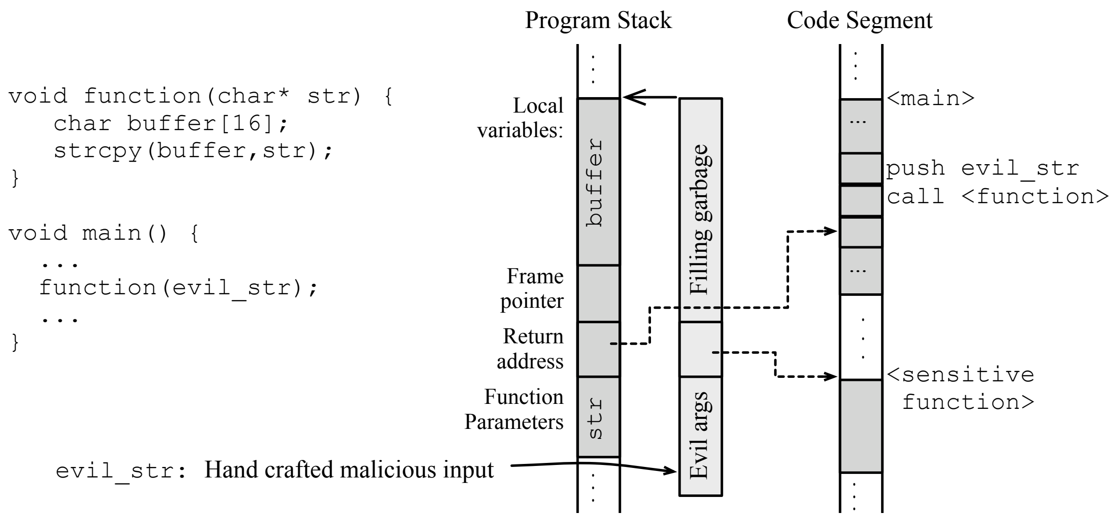
In this chapter, we describe the fundamental concepts of system security and the attack model used in this work. We start (Section 2.1) discuss Buffer Overflow (BOF). Next (Section 2.2), we present a brief description of language-base techniques to deal with BOF. We then (Section 2.3) describe two data structures used to code analyses – Control Flow Graph (CFG) and Dependence Graph (DG), techniques used to analyze memory accesses – Points-To, TFA, and literature proposals of Memory Safety tools – AddressSanitizer, SAFECode. Finally, we present (Section 2.4) the assumptions and the attack model used in this work.

2.1 The Anatomy of a Buffer Overflow Attack

The SIOT that we shall describe henceforth deal with a kind of software attack known as Buffer Overflow (BOF). A *buffer*, also called an array or vector, is a contiguous sequence of elements stored in memory. Some programming languages, such as Java, Python and JavaScript are *strongly typed*, which means that they only allow combinations of operations and operands that preserve the type declaration of these operands. As an example, all these languages provide arrays as built-in data structures, and they verify if indices are within the declared bounds of these arrays. There are other languages, such as C or C++, which are *weakly typed*. They allow the use of variables in ways not predicted by the original type declaration of these variables. C or C++ do not check array bounds, for instance. Thus, one can declare an array with n cells in any of these languages, and then read the cell at position $n + 1$. This decision, motivated by efficiency [Stroustrup, 2007], is the reason behind an uncountable number of worms and viruses that spread on the Internet [Bhatkar et al., 2003].

Programming languages normally use three types of memory allocation regions:

static, heap and stack. Global variables, runtime constants, and any other data known at compile time usually stays in the static allocation area. Data structures created at runtime, that outlive the lifespan of the functions where they were created are placed on the heap. The activation records of functions, which contain, for instance, parameters, local variables and return address, are allocated on the stack. In particular, once a function is called, its return address is written in a specific position of its activation record. After the function returns, the program resumes its execution from this return address.



```
void function(char* str) {
   char buffer[16];
   strcpy(buffer,str);
}

void main() {
  ...
  function(evil_str);
  ...
}
```

Figure 2.1. An schematic example of a stack overflow. The return address of `function` is diverted by a maliciously crafted input to another procedure.

A *buffer overflow* consists in writing in a buffer a quantity of data large enough to go past the buffer's upper bound; hence, overwriting other program or user data. It can happen in the stack or in the heap. In the *stack overflow* scenario, by carefully crafting this input string, one can overwrite the return address in a function's activation record; thus, diverting execution to another code area. The first buffer overflow attacks included the code that should be executed in the input array [Levy, 1996]. However, modern operating systems mark writable memory addresses as non-executable – a protection mechanism known as *Read*$\oplus$*Write* [Shacham et al., 2004, p.299]. Therefore, attackers tend to divert execution to operating system functions such as `chmod` or `sh`, if possible. Usually the malicious string also contains the arguments that the cracker wants to pass to the sensitive function. Figure 2.1 illustrates an example of buffer overflow.

A buffer overflow vulnerability gives crackers control over the compromised program even when the operating system does not allow function calls outside the memory segments allocated to that program. Attackers can call functions from `libc`, for instance. This library, which is share-loaded in every UNIX system, allows users to fork processes and to send packets over a network, among other things. This type of attack is called *return to* `libc` [Shacham et al., 2004]. Return to `libc` attacks have been further generalized to a type of attack called *return-oriented programming* (ROP) [Shacham, 2007]. If a binary program is large enough, then it is likely to contain many bit sequences that encode valid instructions. Hovav Shacham [Shacham, 2007] has shown how to derive a Turing complete language from these sequences in a CISC machine, and Buchanan *et al.* [Buchanan et al., 2008] have generalized this method to RISC machines.

The program in Figure 2.2 contains an example of a buffer overflow exploit, tested on an Intel Core Duo running Linux Ubuntu 32-bits with stack protection disabled. The function `print` is unreachable in this program; however, we are invoking it by changing the return value of `foo`. Function `foo` is called at line 18 of our example, and should, in principle, return to line 19. However, `fun` is copying a string of characters into a local buffer, which has only 10 cells. By carefully preparing this string, we can overwrite its return value, as we have shown in Figure 2.2. In this example, we are replacing this value with the address of `print`. Upon returning, `foo` diverts the program flow to `print`, which loops forever, causing the command at line 19 to never execute.

2.2 Language-Based Techniques for Addressing Buffer Overflow Vulnerabilities

Software code can harbor different types of security vulnerabilities, and those susceptible to buffer overflow attacks are the most exploited. Solutions to address this class of vulnerabilities have long existed, and are largely based on static analysis [Chess and West, 2007], dynamic analysis [Serebryany et al., 2012], or a combination of both.

In static analysis [Chess and West, 2007] (also known as code analysis), analysis is performed without actually running the program; instead either the source code or the object code is inspected, and vulnerabilities flagged. The advantage of this approach is that it does not incur in runtime overhead. Its downside is that the analysis is not able to use information that is only available at runtime, which can determine more

```
void fun(char *s) {
  char t[10];
  strcpy(t, s);
}

  void print() {
    while (1)
      printf(".");
  }

int main() {
  char s[27];
  strcpy(s, "AAAAAAAAAAAAAAAAAAAABBBB");
  sprintf(s + 22, "%c%c%c%c", (int)print & 0xFF,
      ((int)print >> 8) & 0xFF,
      ((int)print >> 16) & 0xFF,
      ((int)print >> 24) & 0xFF);
  fun(s);
  printf("Hello World\n");
  return 0;
}
```

Figure 2.2. An example of C program vulnerable to a buffer overflow exploit.

accurately whether a code fragment indeed harbor a vulnerability. Left without runtime information to help with the decision, static analysis usually flags more vulnerabilities, many of them false-positives.

Dynamic analysis [Serebryany et al., 2012], on the other hand, is performed during system executions, and takes advantage of information that is available only at runtime. Armed with runtime information, it is then able to accurately flag problems in actual runs of the system. (Often, with the same code, a system in execution may or may not end up in an exploitable or insecure state, depending on its input.) Dynamic analysis generates fewer false-positives, but incurs a higher runtime cost, and the results are applicable to only those runs that were analyzed.

Due to their complementary nature, it is common to use hybrid analysis, i.e., the combination of static and dynamic techniques. Usually, static analysis is used first, to identify potential vulnerabilities; the vulnerable stretches are then instrumented and monitored at runtime by dynamic analysis. Note that the higher the number of (potential) vulnerabilities flagged by static analysis, the higher the overhead incurred at runtime. Thus, for efficiency, it is crucial that static analysis flag as few false-positives as possible.

2.3 Code Analysis

In this subsection, we briefly introduce two compiler data structures directed related to our work – Control Flow Graph (CFG) and Dependence Graph (DG) – and point out their limitations in modeling distributed systems. These data structures are very used in code analysis for flagging buffer overflow vulnerabilities and to do code optimizations, for example. We also explain two code analysis we have use PointsTo and TFA.

2.3.1 Control Flow Graph

The CFG [Allen, 1970] is a compiler data structure used to model the control flow of computer programs. The CFG of a program P is a directed graph defined as follows. For each instruction $i \in P$, we create a vertex v_i; we add an edge from v_i to v_j if it is possible to execute instruction j immediately after instruction i. There are two additional vertices, start and exit, representing the start and the end of control flow. Fig. 2.3 shows two examples of CFG.

One class of potential buffer overflow vulnerabilities we might be interested in flagging is variables assignments where the data being assigned are originated externally from user or environment input. If we assume that neither the data sent over the network, nor the executable of the various distributed modules can be tampered with (see Section 2.4.2 for a discussion of these assumptions), then we would see differently the assignments in lines 1 and 5 of Echo Server (Fig. 2.3b). Even though they both involve data coming from the network (through the RECV function), we would deem the one in line 5 as vulnerable, but not the one in line 1. The assignment in line 5 is vulnerable because the data being assigned to `msg` comes from `getc` (line 4, Fig. 2.3a), which could provide malicious data from attackers (`msg` has been used in buffer access). The first assignment in server program (line 1, Fig. 2.3b) is not vulnerable because the data being assigned is a hard-coded constant from the client program (line 1, Fig. 2.3a).

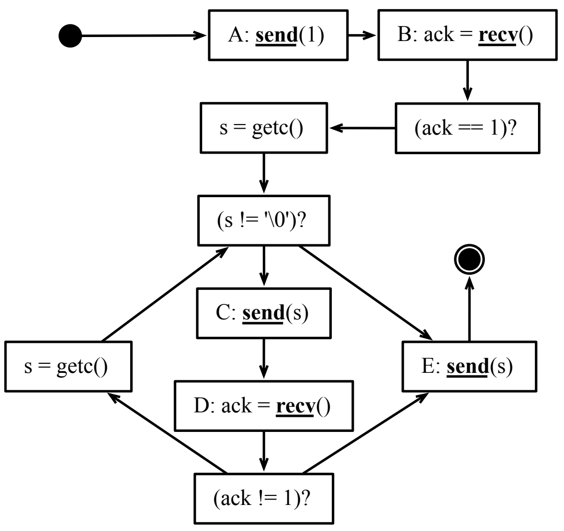
In its standard form, CFG is unable to model the overall control flow of programs that span multiple distributed processes. Thus, they do not provide support to distinguish the two assignments mentioned above. To be safe, both assignments are usually flagged as vulnerable, yielding one true-positive (line 5, Fig. 2.3b) and one false-positive (line 1, Fig. 2.3b). We present a proposal that addresses this issue in Chapter 3.

(a)

```
send(1);
ack = recv()
if (ack == 1) {
    s = getc();
    while (s != '\0') {
        send(s)
        ack = recv();
        if (ack != 1) {
            break;
        } else {
            s = getc();
        }
    }
    send(s);
}
```

(b)

```
msg = recv();
if (msg == 1) {
    send(1);
    do {
        msg = recv();
        putc(msg);
        if (msg != '\0')
            send(1);
        else
            break;
    } while (1);
} else {
    send(0);
}
```



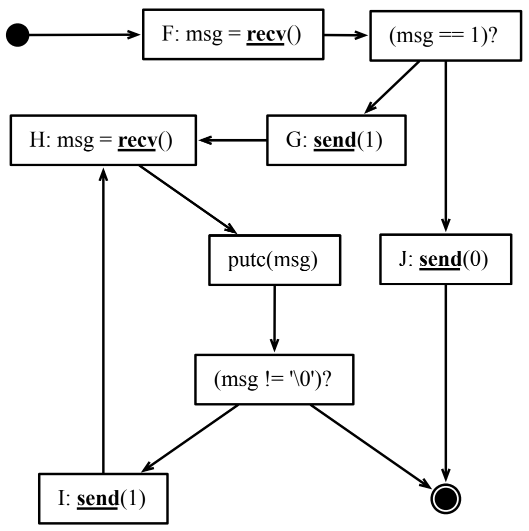


Figure 2.3. Echo application's programs and their respective CFG. (a) Echo client. (b) Echo server.

2.3.2 Dependence Graph

The Dependence Graph (DG) [Ottenstein et al., 1990] is a compiler data structure used to model the dependency of data and instructions in a program. Given a program in a format known as Static Single Assignment form [Cytron et al., 1991] (where each variable has a single definition site), a Dependence Graph (DG) has a vertex for each variable and each operation in the program. There is an edge from variable v to operation i if i denotes an instruction that uses v. Similarly, there is an edge from i to v if i defines variable v.

We show a DG example in Fig. 2.4. The pointer p is mapped to *Memory 0* and $p2$ is mapped to *Memory 1*. There are direct dependences between *Memory 0* and *malloc*

```
  int main(int argc, char** argv) {
    int a = 0;
    a++;
    int* p  = (int*) malloc(sizeof(int));
    int* p2 = (int*) malloc(sizeof(int));
    *p = a;
    if (argc % 2) {
      free(p2);
      p2 = p;
    } else {
      *p2 = *p;
    }
    return 0;
}
```

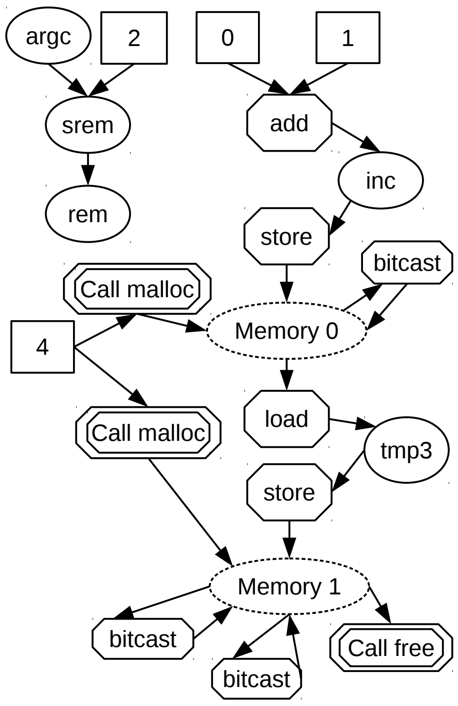


Figure 2.4. Dependence Graph (DG) example of a memory allocation code.

or *store* operations, for instance. The graph also shows that there is a dependency between *Memory 0* and *Memory 1*, because there is a path that link them (*Memory 0*, *load*, *tmp3*, *store* and *Memory 1*).

The DG is a data structure frequently used with the CFG. While CFG model the control flow of a program, DGs focus on the data flow, i.e., dependences between instructions and data. Just like standard CFG, standard DGs are unable to model the data flow in programs that span multiple distributed processes, as discussed in the next section.

To find the Memory nodes, the Points-to Analysis are used before the construction of the DG.

2.3.3 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. To solve it, we extract a number of *constraints* from the program text. These constraints exist in four varieties:

$$\begin{array}{ll} v = \&u & \{u\} \subseteq P(v) \\ v = u & P(u) \subseteq P(v) \\ v = *u & \forall t \in P(u), P(t) \subseteq P(v) \\ *v = u & \forall t \in P(v), P(u) \subseteq P(t) \end{array}$$

If we iterate these equations, then we are guaranteed to reach a fixed point. This fixed point is a solution to points-to analysis. For instance, in Fig. 2.5, we have that variables a and b are aliases. However, b and c are not.

```
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 2.5. Variables a and b are aliases, while b and c are not aliases.

The problem of conservatively estimating the points-to relations in a C-like program has been exhaustively studied in the compiler literature [Andersen, 1994; Hardekopf and Lin, 2007; Pereira and Berlin, 2009; Steensgaard, 1996]. For this work, we use the points-to analysis available in LLVM to find a initial mapping P of pointers to locations. This mapping is used to add Memory vertices which are added to Dependence Graph.

2.3.4 Tainted Flow Analysis

Tainted flow analysis [Denning and Denning, 1977] is a well-known technique used to point out paths in which information can move from the program inputs towards sensitive operations. This kind of compiler analysis has been used with great success to find vulnerabilities such as SQL Injection [Wassermann and Su, 2007], Cross-site Scripting [Rimsa et al., 2010] and Buffer Overflows [Cowan et al., 2000; Levy, 1996]. For an overview of the field, see [Schwartz et al., 2010].

In this work we are interested in finding the set of *taint-writable* arrays in the program to detect buffer overflows. An array is taint-writable if it can be filled with data that comes from a public channel that an adversary can manipulate. The paths that link buffers reachable from untrusted sources are called tainted flows. Any path from an input node in this graph to a vertex that represents a sensitive operation indicates a tainted flow vulnerability. (We discuss an example in the Section 1.3).

There are many tools and frameworks to perform taint analyses in actual programs, including distributed systems such as web applications (e.g. [Tripp et al., 2009; Sridharan et al., 2011; Jovanovic et al., 2006; Lam et al., 2008]). However, they all model the distributed programs in a system as individual entities, and assume that data coming from the network may be malicious. This is also the case of state-of-the-art tools such as TAJ [Tripp et al., 2009], F4F [Sridharan et al., 2011] and Pixy [Jovanovic et al., 2006]. This work crosscheck information inferred from different network programs to improve the precision of taint flow analysis [Denning and Denning, 1977]. We have implemented our taint analysis using the Distributed Dependence Graph that we describe in Chapter 3.

2.3.5 Memory Safety

The research community and the industry have spent a substantial amount of time and energy to find ways to prevent buffer overflows in C programs. However, while there are ways to hamper this kind of attack, the most effective protections tend to impose a heavy burden on the guarded system.

The instance SIoT described in Section 4.2 detects memory accesses that are vulnerable to buffer overflow attacks. The literature has a good number of solutions to perform such detections in standalone programs. They are software-based (e.g. [Criswell et al., 2009; Ghose et al., 2009; Serebryany et al., 2012; Nazaré et al., 2014]), hardware-based (e.g. [Devietti et al., 2008; Nagarakatte et al., 2012]), or hybrid (e.g. [Nagarakatte et al., 2014]).

As example of recent protection mechanisms againts buffer overflow we can highlight the SAFECode [Criswell et al., 2007, 2009], and AdressSanitizer [Serebryany et al., 2012]. Both tools protect a C program by instrumenting it. However, they provide this instrumentation in very different ways.

SAFECode replaces the standard `malloc/free` functions of `stdlib.h` by a custom storage allocator, which tracks the size of each block of memory that it allocates. AddressSanitizer shadows every chunk of memory that it allocates. Each memory access is matched against its shadow area, and an attempt to read or write unallocated data triggers a runtime exception. The main drawback of these tools is the overhead that they impose on compiled programs. Nazaré et al. [2014] present a suite of static analyses that removes part of this overhead.

We understand that both systems are too costly to be used in Networked Embedded Systems. Our work differs from these existing solutions in that we target distributed applications. In other words, none of these works is concerned about drawing information from the network's communication structure to reduce the overhead of solution.

2.4 System Assumptions & Security Model

In this section we discuss the assumptions of the work. We start by discussing the Distributed System execution model and then we present the Security Model assumed in this work.

2.4.1 Distributed System Execution Model

For the purpose of this work, we assume networks of distributed and embedded nodes. We will use *node* or *nodes* to refer to embedded nodes, such as a sensor node, a *RFID* network element or a smart TV, for example. We will use *vertex* or *vertices* to refer to elements of graphs that represent the programs of each node. Each node can interact with its environment through sensors, actuators, or user interfaces. And each node can interact with another node via a *communication channel.*

Our work assumes a standard message passing execution model. Two nodes execute in parallel and may exchange information using send (SEND) or receive (RECV) operations. We have assumed that each node is identified by a unique id, like IP or an other identification number. Each SEND and RECV takes an argument that uniquely identifies the ID of its communication partner. No RECV may match messages from more than one node sender (i.e. no wildcard receives). We will use the word *channel* to refer the communication channel between *nodes* and the word *link* to refer the links between SENDs and RECVs of programs of different nodes. For each pair of processes there exists one or more bidirectional communication channels. All messages sent along a channel are delivered in *First In First Out* (FIFO) order. SENDs program commands are non-blocking and any number of messages may be in-flight at the same time. When a RECV command is reached, the program is blocked until the arrival of a SEND message.

Nodes may read arbitrary input data from user or environment, such as sensor reads, *GPS* data, images in a smart video camera, internal machine commands (e.g.

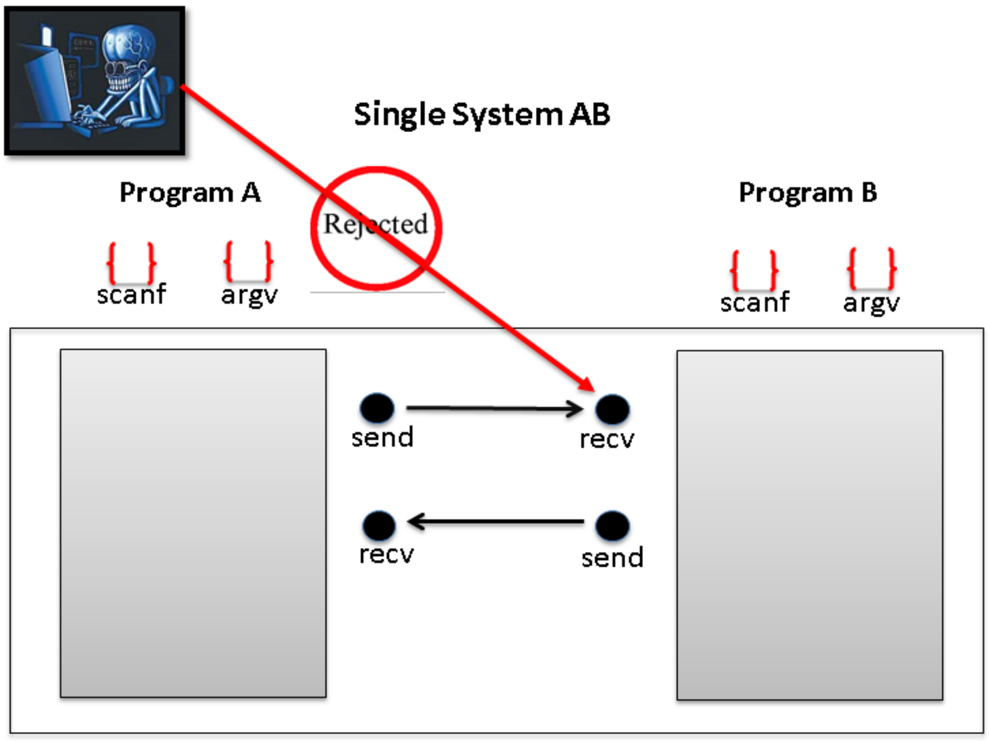


Figure 2.6. Attackers are not able to input data onto the system through the network sources.

temperature actuators), other machine commands (e.g. *RFID* reader signal), or user commands (e.g. television remote control signals, car break pedal sensor, refrigerator's command buttons).

2.4.2 Security Model

We assume that both the system running at each node and the communication between different nodes is protected against tampering. Different security mechanisms can be used to implement such protections. For example, Trusted Platform Module (TPM) [Kinney, 2006; Boukerch et al., 2007; Seshadri et al., 2004] can be employed to ensure the integrity of nodes systems and cryptographic solutions like [Perrig et al., 2002; Kothmayr et al., 2011; Oliveira et al., 2008] can be used to establish a secure communication channel.

Attackers can have control over the input data that the nodes receive from its environment (Fig. 2.7). This includes data captured by the sensors or input from the user interfaces, but excludes data coming from network interfaces (Fig. 2.6), we assume a secure communication channel.

Though limited in the type of attacks they can launch, such attackers can poten-

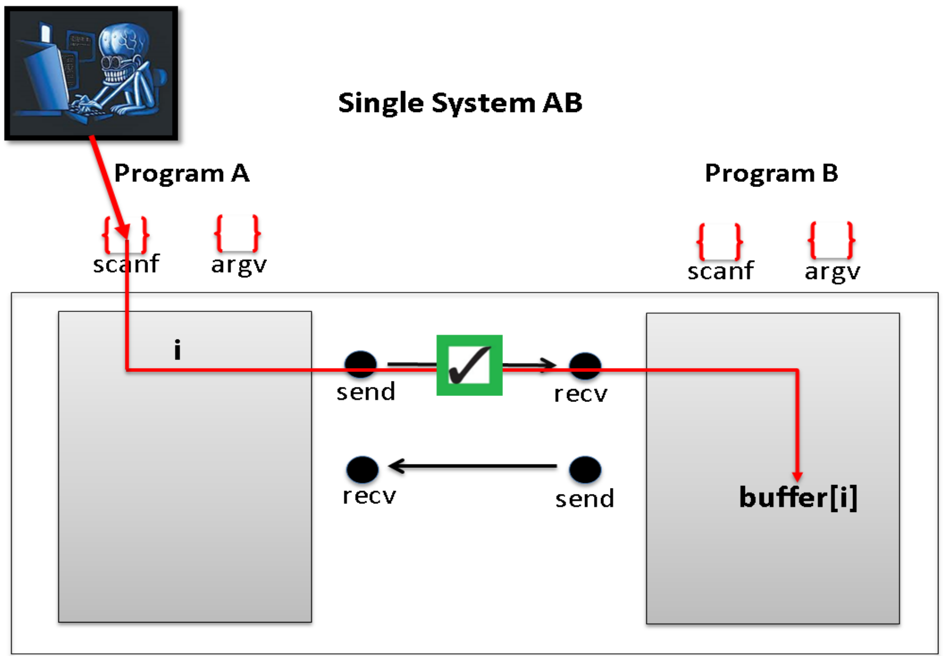


Figure 2.7. Attackers can have control over the input data that the nodes receive from its environment.

tially cause security problems if the code running on the nodes harbors certain types of vulnerabilities. For example, if the code does not check array bounds, certain inputs may cause buffer overflows. Attackers can then manipulate the environment (to produce spurious sensor readings) or provide spurious user input to launch a buffer overflow attack, leading the nodes to denial of service or malicious behavior. Because these nodes are connected to the Internet, misbehaving nodes can be used as a proxy to attack other nodes in the network. Note that malicious input injection attacks can be most effectively exploited if the attacker has information of vulnerabilities in the code. This information is readily available in case of open source programs, and can also be obtained from code reverse engineering, and program fuzzing exercises.

2.5 Chapter Summary

In this chapter, we discuss BOF attacks and present the fundamental concepts of code analysis used in our work. Code analysis works handle the system as standalone programs. However, as we have shown in this chapter, a whole vision of the distributed system can help improve the precision of the code analyses of buffer overflow, for example.

We show that the CFG and DG are important data structures used in code analysis. We describe some techniques used to analyze memory accesses as Points-To and TFA, and literature proposals of Memory Safety tools as AddressSanitizer and SAFECode. However, they are not designed to handle more than one program (see Chapter 6 for a discussion of other related works).

We also describe the distributed system execution and security models assumed in this work. Our distributed models assumes that two nodes execute in parallel and may exchange information using send (SEND) or receive (RECV) operations. When a RECV command is reached, the program is blocked until the arrival of a SEND message. Our model attack assumes that attackers are not able to input data onto the system through the network sources, however attackers can have control over the input data that the nodes receive from its environment.

In the next chapter (Chapter 3) we present our solution proposal to see a distributed system as whole and prove that our algorithm is correct and always terminates.

Chapter 3

Communication Links Inference

In this chapter, we present our proposal for the analysis of distributed systems. We start (Section 3.1) by introducing the concepts of DCFG and DDG (Section 3.3), two data structures that would enable us to model control flow and data flow across different programs in a distributed system. We then describe the Elevator algorithm (Section 3.2) used to inference communication links between the programs. Finally (Section 3.4), we formalize the level assignment algorithm briefly described in Section 3.1, and prove that the algorithm terminates, and is correct.

3.1 Distributed Control Flow Graph

To analyze an entire distributed system, we need to work with control flow graphs that transcend program boundaries. CFGs model the control flow of individual programs. We propose the notion of DCFG as a way to model the communication between two programs in a system. We describe how DCFGs are built below.

Let $\{\mathcal{C}_1, \mathcal{C}_2\}$ be a pair of CFGs that constitute a system and $\mathcal{D}$ the resulting DCFG. $\mathcal{D}$ contains $\mathcal{C}_1$ and $\mathcal{C}_2$ as subgraphs. Inter-program edges connecting $\mathcal{C}_1$ and $\mathcal{C}_2$ are then added to $\mathcal{D}$: for each pair of SEND and RECV vertices (each from the two different CFGs) that may communicate, we add an edge from the former to the latter. That is, for each pair of vertices $s_i \in \mathcal{C}_1$ and $r_j \in \mathcal{C}_2$, if there is an execution sequence in which a message issued by s_i reaches r_j, we add to $\mathcal{D}$ an inter-program edge from s_i to r_j. And, for each pair of vertices $s_k \in \mathcal{C}_2$ and $r_t \in \mathcal{C}_1$, if there is an execution sequence in which a message issued by s_k reaches r_t, we add to $\mathcal{D}$ an inter-program edge from s_k to r_t.

In principle, we can add inter-program edges linking every send vertex in one of the CFGs to every receive vertex in the other CFG in the system. However, the

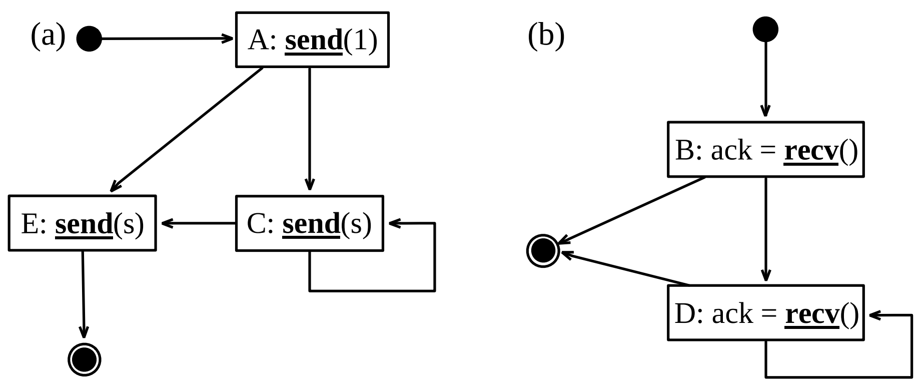


Figure 3.1. (a) Send-Graph and (b) Receive-Graph for echo client (Fig. 2.3a).

resulting DCFG would have inter-program edges linking sends and receives that could not be the matching ends of a communication. For instance, in Fig. 2.3, sends from vertex A are not received by vertex H; every send from A will be received by F before H has a chance to execute. To define a DCFG that better models the workings of a system, we ([Teixeira et al., 2015b]) propose the *Elevator Algorithm* (Algorithm 1).

3.2 Elevator Algorithm: Linking Distributed Systems Programs' Codes

To explain the Elevator Algorithm, we introduce the notions of Send-Graph, Receive-Graph, and levels. Given a CFG $\mathcal{C}$ of a program, we define its associated Send-Graph $\mathcal{S}$ and Receive-Graph $\mathcal{R}$ as follows. For each vertex $v \in \mathcal{C}$ labeled with a send operation, we add a vertex v' to $\mathcal{S}$. We also add $start'$ and $exit'$ vertices, which correspond to start and exit in $\mathcal{C}$. Edges in $\mathcal{S}$ correspond to paths between sends in the original $\mathcal{C}$. For every pair of vertices $u, v \in \mathcal{C}$, we add an edge $\overrightarrow{u'v'}$ to $\mathcal{S}$ if, and only if: (i) there exists a path p from u to v in $\mathcal{C}$, and (ii) p does not contain any other sends. We create $\mathcal{R}$ in a similar way, replacing sends by recvs in the procedure described above. Fig. 3.1 shows the $\mathcal{S}$ and $\mathcal{R}$ derived from the CFG in Fig. 2.3a.

Next, we move on to the concept of *level*. Given a Send-Graph, its level 0 contains the start vertex. Level 1 contains the sends that are reachable, in one step, from the root. More generally, level $n + 1$ contains the immediate successors of vertices in level n. The procedure is complete when the vertices in the just-generated level do not have successors, or the just-generated level is a duplicate of a previously existing one. The concept of level can be similarly defined for Receive-Graphs. We show an example in

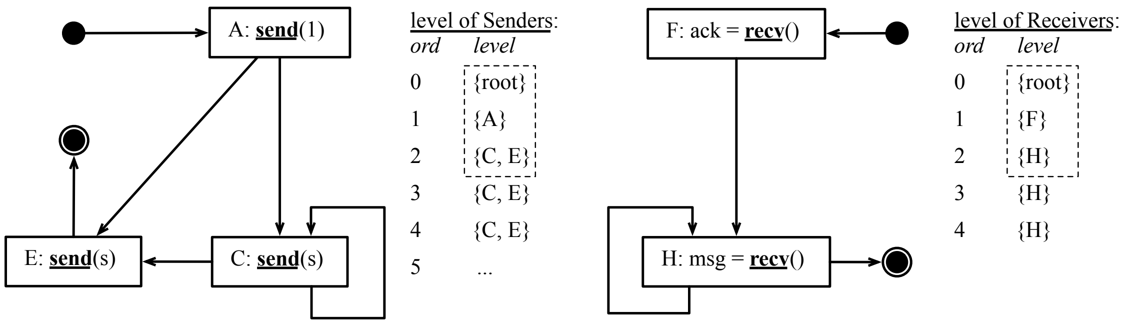


Figure 3.2. Levels for echo client's SENDs (left-hand-side) and echo server's RECVs (right-hand-side). Dashed boxes delineate L when *solution*(L) is true. The predicate *solution* is defined by Rule [Sol] in Fig. 3.4.

Fig. 3.2.

Consider echo client program Send-Graph (Fig. 3.2 left-hand-side). Its level 0 contains the root of the graph. Its level 1 contains the immediate successors of root node, i.e., $\{A\}$. Its level 2 contains the immediate successors of each send node of level 1, i.e., $\{C, E\}$. The successors of C is $\{C, E\}$, and E does not have successors. We find ourselves in a cycle, and the traversal can now stop. The levels for echo server Receive-Graph can be similarly determined.

Given the CFGs in Fig. 2.3, their resulting DCFG can be built by linking the send vertices in one CFG with the receive vertices of the same level in the other. Links are established between SENDs and RECVs that have the same level because they model matching ends of message exchanges. Fig. 3.3 shows the links between the SENDs and RECVs in our example. Links between the sends in echo server and the receives in echo client are omitted for simplicity. The steps described above are captured in the *Elevator Algorithm* (Algorithm 1).

3.3 Distributed Dependence Graph

Distributed Dependence Graph (DDG) can be built following similar steps. We first create the DG of each program in the distributed system. For each instruction that accesses the network, we create a vertex in the graph to represent this operation. Finally, we used the levels defined in the Send-Graph and Receive-Graph to decide which edges should be inserted between SENDs and RECVs in a similar way as previously described. Thus, the previously final graph contains the dependences of all the programs of the

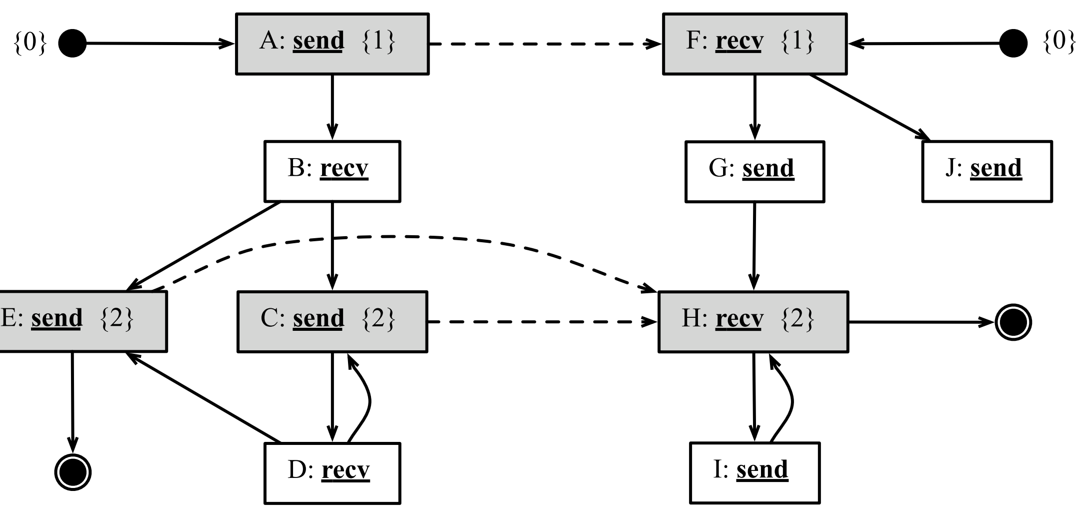


Figure 3.3. Links between echo client's SENDs and echo server's RECVs of our running example. Numbers next to vertices denote their respective levels.

distributed system as if they were a single program system. The DDG can be used for different security analyses like the detection of buffer overflow or integer overflow [Rodrigues et al., 2013] vulnerabilities. For instance, in Chapter 5 we describe how we have used the DDG to find dependences between user or environment inputs and memory accesses that can be used for buffer overflow attacks and how to mitigate false positives due to network access.

3.4 Formalization of Elevator Algorithm

Equation 3.1 defines the levels of either a Send-Graph or Receive-Graph graph, named here as Message-Graph *mg* as follows:

$$\begin{aligned} \mathit{level}(mg, 0) &= \{\mathit{start}\} \\ \mathit{level}(mg, n) &= \{v \mid \overrightarrow{uv} \in mg \wedge u \in \mathit{level}(mg, n-1)\} \end{aligned} \tag{3.1}$$

Equation 3.1 gives us a way to generate levels, which we formalize in Fig. 3.4. To produce all the levels of a program, we continually generate new levels, until we produce a set that has been created before. Rule [SUC] constructs the successor of a level, following Equation 3.1. Rules [LVR] and [LVN] determine a recurrence relation that generates levels. The first rule, [LVR], gives us the base case. The second rule, [LVN], gives us the inductive step. Finally, Rule [SOL] defines a solution to the problem

Algorithm 1: Elevator

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

▷ Set the SEND and RECV levels
foreach $G_i \in \{\mathcal{S}_1, \mathcal{S}_2\} \cup \{\mathcal{R}_1, \mathcal{R}_2\}$ 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
$\mathcal{D} \leftarrow \mathcal{C}_1 \cup \mathcal{C}_2$
for $k \leftarrow 1$ to $n$ do
    foreach $v_s \in L_{\mathcal{S}_1,k}$ and $v_r \in L_{\mathcal{R}_2,k}$ do
        add an edge from $v_s$ to $v_r$ in $\mathcal{D}$
    foreach $v_s \in L_{\mathcal{S}_2,k}$ and $v_r \in L_{\mathcal{R}_1,k}$ do
        add an edge from $v_s$ to $v_r$ in $\mathcal{D}$
```

$$[\textsc{Suc}] \qquad \frac{\forall v \in V,\ (v \in S' \Leftrightarrow \exists u \in S, \overrightarrow{uv} \in E)}{succs(S, S')}$$

$$[\textsc{Lvr}] \qquad levels([start])$$

$$[\textsc{Lvn}] \qquad \frac{levels([L_2 \mid L]) \qquad succs(L_2, L_1)}{levels([L_1,\ L_2 \mid L])}$$

$$[\textsc{Sol}] \qquad \frac{levels([L_1 \mid L]) \qquad L_1 \in L}{solution(L)}$$

Figure 3.4. The fixed point computation of levels for a Message-Graph $mg = (V, E)$. We use Prolog syntax: $[H_1, H_2 \mid T]$ denotes a list of elements H_1 (first), H_2 (second) and T (tail).

of producing levels to messages graphs. According to this rule, we stop generating levels as soon as we produce a set of vertices that we had generated before. As we prove later, in Lemma 3.4.6, this algorithm always terminates.

If $solution(L)$ is true, we number the levels of a program according to the rules below:

$$[\textsc{Zer}] \qquad \mathit{ord}([\,], 0)$$

$$[\textsc{Num}] \qquad \frac{\mathit{ord}(L, N)}{\mathit{ord}([L_1 \mid L], N+1)}$$

In other words, the last element of L is given number zero, and the first is given number $N-1$, given that L has N elements. This numbering is consistent with that used in Equation 3.1. Fig. 3.2 (a) shows the sets of levels for the Send-Graph in the echo client of our running example. Fig. 3.2 (b) shows the levels for the Receive-Graph of the echo server.

3.4.1 Correctness

The essential property that we want to ensure with the idea of levels is the invariant that only SENDs and RECVs in the same level can communicate. To prove that our algorithm deliver us this property, we define the semantics of a toy language that abstracts message exchange between two processes.

Core Language Fig. 3.5 shows the semantics of a simple language [Nielson et al., 1999], that allows us to define communication protocols between two programs. These two programs, P_1 and P_2, share two integer counters, N_1 and N_2, where N_i simulates the input queue of P_i. Our language has five different instructions: `send`, `recv`, `choose`, `jump` and `halt`. A program is a list of such instructions, which is indexed by an integer, henceforth called `pc`. The first instruction, `send`, lets P_i increase the counter N_{1-i}; hence, simulating the transit of data from P_i to its peer process P_{1-i}. The command `recv` gives process P_i the opportunity to read data, which we, concretely, translate to a decrement of N_i. We do not simulate “waiting" in our language: if a process tries to read an empty counter, e.g., $N = 0$, then the program is stuck. Given our simple communication model, the correctness proofs that we present in the rest of this section require only the existence of one execution in which no process is stuck. We incorporate control flow in our language via instructions `choose` and `jump`. The former is a non-deterministic branch, the latter is an unconditional jump. A program terminates once it reaches instruction `halt`. When both programs reach this instruction, we say that the entire application terminates.

$$((P_1, \texttt{eof}, N_1)/(P_2, \texttt{eof}, N_2)) \rightarrow (N_1, N_2)$$

$$\frac{\mathit{eval}(P_1, \texttt{pc}_1, N_1, N_2) \rightarrow (\texttt{pc}_1', N_1', N_2') \qquad ((P_1, \texttt{pc}_1', N_1')/(P_2, \texttt{pc}_2, N_2')) \rightarrow (N_1'', N_2'')}{((P_1, \texttt{pc}_1, N_1)/(P_2, \texttt{pc}_2, N_2)) \rightarrow (N_1'', N_2'')}$$

$$\frac{\mathit{eval}(P_2, \texttt{pc}_2, N_2, N_1) \rightarrow (\texttt{pc}_2', N_2', N_1') \qquad ((P_1, \texttt{pc}_1, N_1')/(P_2, \texttt{pc}_2', N_2')) \rightarrow (N_1'', N_2'')}{((P_1, \texttt{pc}_1, N_1)/(P_2, \texttt{pc}_2, N_2)) \rightarrow (N_1'', N_2'')}$$

$$\mathit{eval}(P, \texttt{pc}, N_{in}, N_{out}) \rightarrow \begin{cases} (\texttt{pc}+1, N_{in}, N_{out}+1), & \mathit{if}\ P[\texttt{pc}] = \texttt{send} \\ (\texttt{pc}+1, N_{in}-1, N_{out}), & \mathit{if}\ P[\texttt{pc}] = \texttt{recv} \wedge N_{in} > 0 \\ (l, N_{in}, N_{out}), & \mathit{if}\ P[\texttt{pc}] = \texttt{choose}(\texttt{pc}_1, \texttt{pc}_2) \wedge l \in \{\texttt{pc}_1, \texttt{pc}_2\} \\ (\texttt{pc}', N_{in}, N_{out}), & \mathit{if}\ P[\texttt{pc}] = \texttt{jump}(\texttt{pc}') \\ (\texttt{eof}, N_{in}, N_{out}), & \mathit{if}\ P[\texttt{pc}] = \texttt{halt} \end{cases}$$

Figure 3.5. The semantics of our message passing language.

Essential Properties To prove that only SENDs and RECVs in the same level can exchange messages, we introduce the notion of *depth*, which we define as follows:

Definition 3.4.1 *Given a Send-Graph (or a Receive-Graph) mg, a vertex v has depth d if there exists a path from* start *to v passing through $d-1$ vertices. The same vertex might have several depths.*

If mg_s is the Send-Graph inferred from a program P, written in our core language, and $s_i \in mg_s$ is the vertex that corresponds to send operation i in P, we shall refer to them as the pair i/s_i. The depth of s_i determines how many messages have been sent by P once i is evaluated. We state this property in Lemma 3.4.2. There exists an analogous result for receivers, which we omit, for the sake of brevity.

Lemma 3.4.2 *Let mg_s be the Send-Graph of program P, such that i/s_i is a pair. If s_i has depth d, then there exists an execution of P in which i is evaluated after d messages are sent.*

Proof: The proof is by induction on d.
Base case: Let $d = 0$, which corresponds to the point before any instruction of P,, then $v_s = start$. Because there is no send before the first instruction of P, the lemma is vacuously true.
Induction hypothesis : the lemma is true until depth d.
Induction step : If a vertex s has depth $d+1$, then it is preceded by a vertex s' of depth d, by the definition of depth. If $i' \in P$ corresponds to s', by induction it is reached

after d messages are sent. Because instruction $i \in P$, that corresponds to s, is the first send after i', the lemma is true. $\square$

There exists a very close relationship between levels and depths. Lemma 3.4.3 makes this relationship explicit. According to this lemma, if two vertices belong into the same level, then there exist paths of same length linking these vertices back to the start vertex. In other words, the idea of levels group vertices that share common depths. Notice that the same vertex can have different depths. As an example, vertex C in Fig. 3.1 (a) has infinite different depths, as it is part of a loop. Thus, it is possible to have the same vertex in different levels.

Lemma 3.4.3 *If $v \in level(mg, n)$, then there is a path of depth n from v to* start.

Proof: The proof is by induction on n.
Base case: by the first case of Equation 3.1, only *start* is in $level(mg, 0)$. The depth of *start* to *start* is zero.
Induction hypothesis : the lemma is true until level n.
Induction step : All the vertices in $level(mg_s, n+1)$ are successors of vertices in $level(mg_s, n)$, by the second case of Equation 3.1, which, by induction, can be reached after n hops. From the definition of depth, we conclude that vertices in $level(mg_s, n{+}1)$ can be $n + 1$ hops distant from *start*. $\square$

The concept of depth is key to prove the correctness of our method to build distributed graphs, because only vertices with the same depth can communicate, as we state in Lemma 3.4.4.

Lemma 3.4.4 *Given a Send-Graph mg_s, a Receive-Graph mg_r, vertex $s \in mg_s$ and vertex $r \in mg_r$, if i_s/s and i_r/r are pairs, then i_r can receive a message sent by i_s if, and only if, s and r share a common depth.*

Proof: From Lemma 3.4.2, we know that the depth of a sender s corresponds to the number of messages sent before i_s is executed. Similar result applies to the pair i_r/r.
Necessity: If r and s do not share a common depth, then any path from $start_s$, the start of mg_s to s will cause the issuing of a number k of messages that is different than the number of messages that can be received in any path from $start_r$, the start of mg_r to r.
Sufficiency: If s and r share a common depth d, then there exists a path $(start_s, s_1, \ldots, s_{d-1}, s)$ in mg_s, and another path $(start_r, r_1, \ldots, r_{d-1}, r)$ in mg_r in which d messages are sent from senders corresponding to s_i to receivers corresponding to r_i.
$\square$

As a corollary of Lemma 3.4.4, we have that two vertices can communicate if, and only if, they belong into the same level. We state formally this property in Theorem 3.4.5.

Theorem 3.4.5 *Given a Send-Graph mg_s and a Receive-Graph mg_r, and pairs i_s/s, i_r/r, i_s can send a message to i_r if, and only if, s and r belong into the same level.*

Proof: *Sufficiency:* From Lemma 3.4.3, we know that two vertices at the same level have a common depth, and by Lemma 3.4.4 we know that they can communicate.
Necessity: still by Lemma 3.4.4, if two vertices do not share a common depth, then they cannot communicate. $\square$

3.4.2 The Computation of Levels Terminates

Equation 3.1 gives us an algorithmic way to compute the set of levels of messages graphs. We can construct a level n from the definition of levels $1 \ldots, n-1$. Eventually we will get two levels, e.g., n and $n+1$ which are the same. In this case, we know that we will have a cycle of known levels, as we state in Lemma 3.4.6.

Lemma 3.4.6 *If levels n and $n+k$ are the same, then the levels $n+1$ and $n+k+1$ are the same.*

Proof: According to the recurrence relation seen in the second part of Equation 3.1, $level(k+1)$ is totally defined by $level(k)$. $\square$

Lemma 3.4.6 gives us an interactive way to build levels for a Send-Graph (or a Receive-Graph) graph: it is enough to solve Equation 3.1 successively, until we find two levels that are the same. This process is guaranteed to terminate, as we prove in Theorem 3.4.7.

Theorem 3.4.7 *The iterative construction of levels terminate.*

Proof: The number of levels is finite, because a graph with N vertices might have at most 2^N different levels. Thus, successive applications of Equation 3.1 will eventually produce two levels that are the same. From Lemma 3.4.6, we know that we have found a cycle, and no new level will be discovered. $\square$

3.4.3 Complexity

The complexity of our algorithm is the sum of the complexities of Rules LVN and SOL. The number of levels is upper bounded by 2^N, where N is the number of vertices – SEND or RECV – in the messages graph (Send-Graph or Receive-Graph).

The computation of successors, via Rule SUC, in Fig. 3.4 has an $O(N^2)$ worst case. Hence, Rule LVN might have an $O(2^N \times N^2)$ worst case. The pertinence test, performed in Rule SOL is $O(2^N) \times O(N)$, i.e., maximum number of levels multiplied by the time to check if two levels are the same.

Therefore, our algorithm might have exponential complexity. We emphasize that we have not found a graph that gives us the exponential number of levels, although we can construct graphs that gives us a quadratic number.

For instance, $mg = (\{start, b, c, d, e\},\ \{\overrightarrow{start\ b}, \overrightarrow{bc}, \overrightarrow{cd}, \overrightarrow{de}, \overrightarrow{e\ start}, \overrightarrow{ec}\})$ has five vertices and yields 16 levels. As we show empirically (Section 5.4), our algorithm seems to be polynomial in practice.

3.5 Chapter Summary

In this chapter, we have discussed how to obtain a inter-program vision of the distributed system by the creation of Distributed Control Flow Graph – DCFG, and Dependence Graph – DG as an extension of de data structures CFG and DG (discussed in Chapter 2).

To construct this vision, we introduced the Elevator algorithm that link SENDs and RECVs of two program using the concept of Level. We then have formalized Elevator algorithm, proved that its correctness and termination and discussed its complexity.

With this vision is possible improve the precision of distributed system code analysis by crosschecking information between different programs. For example, in the Chapter 5 we will describe how we can improve the precision of buffer overflow's analysis using this inter-program vision.

In the next chapter we show how we have implemented our Elevator algorithm and the other compiler structures discussed in this chapter (e.g., DCFG, DG, Send-Graph and Receive-Graph) as a framework on top of the LLVM compiler.

Chapter 4

Architecture and Implementation

In this chapter, we describe the architecture of our framework and some implementation aspects. We start (Section 4.1) by describing how we have implemented our algorithm and organize our framework using the LLVM compiler. We then present architecture of two DSA instances: SIoT and DistViewer. Firstly (Section 4.2), we discuss the SIoT architecture and how we have implemented its companion distributed tainted flow analysis. Finally (Section 4.3), we present the DistViewer tool, we define the Network Programing Slice (NetSlice) and we discuss architecture and implementation of DistViewer.

4.1 DSA Architecture and Implementation

Our framework is a template for deriving static analyses tools, which we We have implemented the DSA as an extension of the LLVM compiler. Therefore, the DSA can be directly applied to distributed systems made in C. No change is required in the source code of the programs that make up the distributed system, even to include annotations in the code (as is required on many static analysis solutions). The source code is available online.

A tool derived from DSA is composed of three layers: merging, linking and specific analysis. Fig. 4.1 shows the DSA architecture.

The first layer, *Merging*, provides as output a program that is the junction of the distributed system programs. This output is a program that represents the distributed system as a whole. Through this output, it is possible to extend conventional static analysis structures (as we did for the CFG and the DG) or create new structures (as we did with the Send-Graph and Receive-Graph).

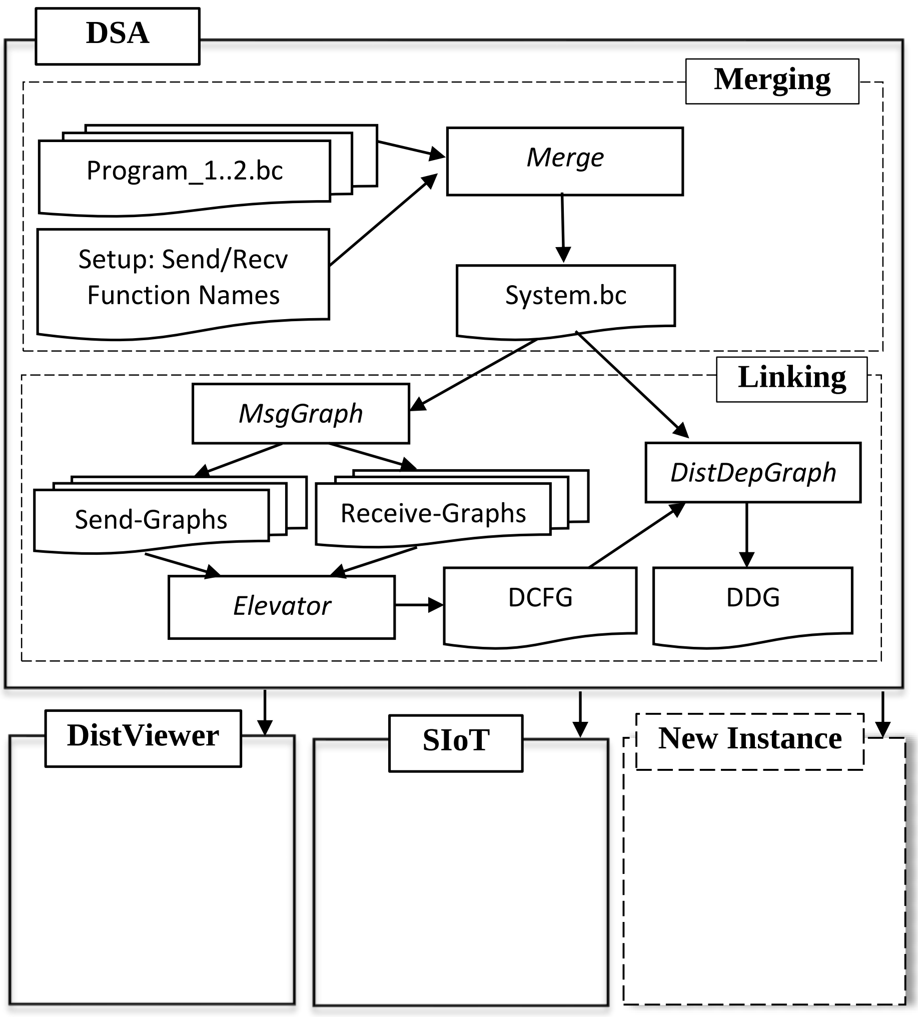


Figure 4.1. DSA Architecture.

The second layer, *Linking*, provides structures used in conventional static analysis extended to the context of distributed systems. Such structures can be used as a basis to extend conventional analyzes for treating distributed systems. For example, the Distributed Dependence Graph (DDG) and DCFG, supplied as the output of the second layer, can be used to develop a distributed integer overflow analysis or a distributed range analysis. Another example: our DistViewer extends the first and second layers to provide the programmer an overview of programs focusing on the interaction over a network.

The third layer, *DSA Instance*, is composed by specific static analysis like the SIoT. SIoT extends the DSA and performs a tainted flow analysis in order to detect buffer overflows. This layer can be extended or used for other types of analysis such as, for example, analysis of leakage of confidential information. In order to create a new specific static analysis tool, the developer must implement your analysis using the support of DSA framework – we call the specific analysis the DSA *instance*.

In the next subsections we describe this architecture in more detail.

4.1.1 Merging

DSA Merging receives as input files written in the LLVM IR called *bytecodes*. The LLVM IR is a low-level programming language, formed by three-address instructions called bytecodes, which manipulate typed operands. Usual types are integers of several different sizes, floating-point numbers, program labels, bit vectors and arrays.

Merging combine two bytecode files into a single file. In this layer we must resolve name conflicts, i.e., different files may define the same names. We solve such conflicts through renaming functions and variables.

Merging different bytecode files only requires the names of SEND and RECV functions and parameters. Our tool analyzes all the network functions present in each bytecode file, based on usual libraries of the C language. If necessary, the user can modify the setup file to change or add others network functions. We then output this list to the user, who determines which functions transmit, and which functions receive data.

Based on the network functions information, DSA adds special tags to the LLVM bytecode files, identifying SEND and RECV and their data parameters. The annotated bytecodes are then merged into a single unit (*System Bytecode* in Fig. 4.2) and exported to file system to facilitate subsequent analyses. This output is a program that represents the distributed system as a whole. Through this output, it is possible to extend conventional static analysis structures (as we did for CFG and DG) or create new structures (as we did with the Send-Graph and Receive-Graph). Notice that the *System Bytecode* with the meta-informations inserted by DSA can be used as input to static analysis focus on inter-program view.

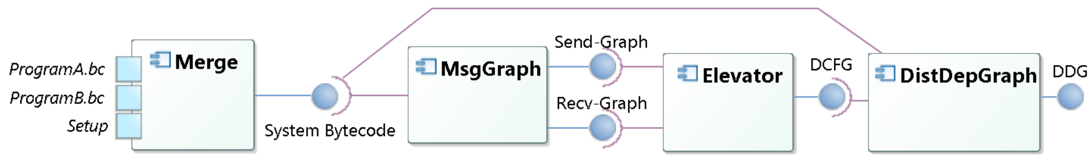


Figure 4.2. DSA Architecture UML Component Diagram.

In addition to this file, we also produce, during the merging phase a “setup" file, containing meta-information to guide the automatic analyses. This meta-information describes, for instance, which functions are `send` or `recv` operations.

4.1.2 Linking

In the linking layer we start extracting the Send-Graph and Receive-Graph for each program, and, using the Elevator algorithm, we define the level of each SEND and RECV and produce the DCFG and DDG.

Firtly, *MsgGraph* extracts the Send-Graph and Receive-Graph for each program, that are exported as interfaces to possible instances of DSA. Firstly, using *MsgGraph* we extract the CFG of each function using the LLVM infrastructure. Then we construct the CFG of each program as whole (an Interprocedural Control Flow Graph (ICFG)). Using this ICFG, we extract the Send-Graph and Receive-Graph and export them as interfaces to be used by *Elevator*.

The *Elevator* then assigns levels to functions marked as SEND or RECV (using the steps described in Section 3.2). After determining levels, *Elevator* creates the DCFG by linking the SENDs and RECVs vertices in the same level. DCFG is also exported as interface.

Finally, using the DCFG as input, a third component, *DistDepGraph*, builds the DDG. *DistDepGraph* firstly find the memory vertices using LLVM Points-to Analysis, which maps the pointer to memory nodes in the DG (we discuss Points-to Analysis in 2.3.3). Then *DistDepGraph* builds the DG for each program. Finally, using the construct the DG of each program, *DistDepGraph* adds links between vertices marked as SEND and RECV in the same level. This data structure is then exported as interface to possible instances of DSA.

Notice that the compile structures exported by DSA framework (DDG, DCFG, Send-Graph and Receive-Graph) are ready to be used to derived new analyses.

To validate our proposal, we have designed and implemented two instances: SIoT and DistViewer. Fig. 4.3 shows framework's interface usage of the two instances described in this work. The former uses the proposed framework to implement a distributed tainted flow analysis to secure ContikiOS applications against buffer overflow attacks in a more energy-efficient manner. SIoT uses the *System Bytecode* and DDG interfaces (Fig. 4.3). DistViewer uses the graphs generated by the framework to provide programing slices of distributed systems. DistViewer uses as input the following DSA interfaces: Send-Graph, Receive-Graph, DCFG and DDG(Fig. 4.3).

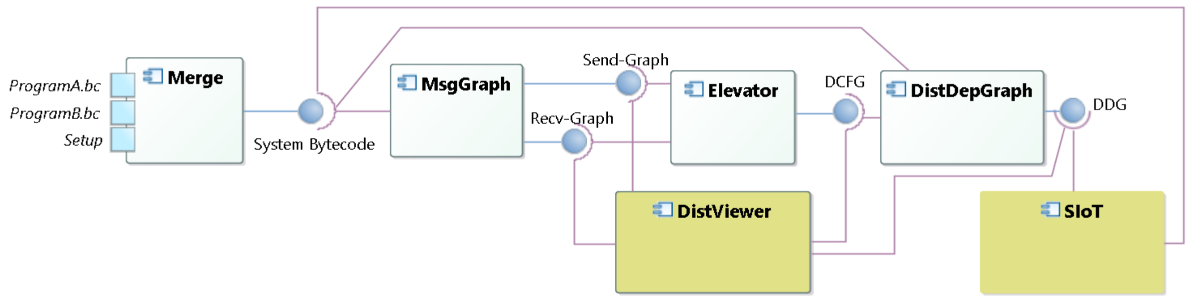


Figure 4.3. Instances of DSA: DistViewer and SIoT.

We describe SIoT (Section 4.2) and DistViewer(Section 4.3.1) architectures in more detail in the following sections.

4.2 SIoT Architecture and Implementation

The DDG exported by DSA let us track the flow of information throughout the program, giving us a way to point out vulnerabilities. Fig. 4.4 summarizes SIoT architecture. To detect program vulnerabilities, we firstly find the untrusted sources. *Input Value Analisys* component receives the *System Bytecode* exported by DSA framework and search for program inputs. The inputs are classified on user or network inputs. The list of inputs values is, then, exported as an interface.

We then feed the DDG and the input list to the *Tainted Flow Analysis* component, the pass which implements our Tainted Flow Analysis. This pass searches for paths, within the DDG, between untrusted inputs and memory access operations.

After analyzing a program, our *Tainted Flow Analysis* component produces the following outputs: (i) ABCs statistics, i.e., the number of true-positives, and potential false-positives (if we had analyzed each program of a system independently); (ii) the graph of vulnerable paths in the program (TFA Graph); and (iii) the *ABC Lines* which list the lines of code which must receive ABCs.

Finally, the *Instrumentator* component return the *System Bytecode* to original form (ProgramA and ProgramaB bytecodes) and inserts ABCs in the lines highlighted by the TFA.

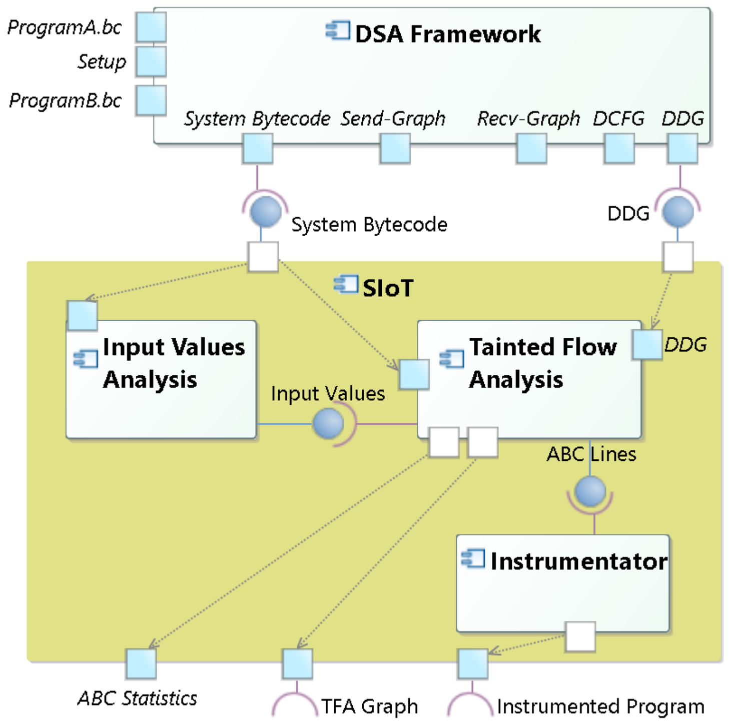


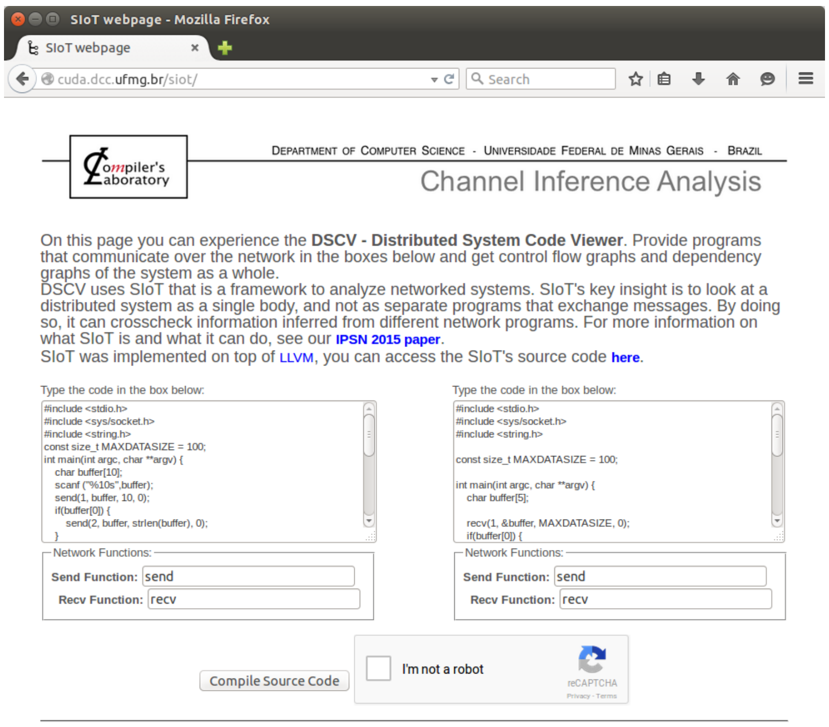
Figure 4.4. SIoT Instance Architecture.

4.3 DistViewer - Distributed System Code Viewer

Using the DSA framework, we have designed, developed and tested a tool which we call Distributed System Code Viewer (DistViewer). DistViewer receives as input two C programs that communicate over the network, sets up and runs the DSA framework and displays, via a web interface [1], graphs that summarize the results of the communication links inference, namely: the Send-Graph, the Receive-Graph, the Distributed Dependence Graph, and the NetSlice – the latter, a graph that summarizes the dependence between data and network). Fig. 4.5 shows the initial web screen of DistViewer.

DistViewer can be used to to assist developers of distributed systems and/or designers of network protocols. From the graphs of the distributed system generated by DistViewer, the designer or developer can, for instance, (i) check for semantic errors, aided by graphs that summarize and highlight the interaction over a network; (ii) perform optimization in the code that take into account interaction among network elements; and (iii) check paths from user inputs to memory nodes to find data dependencies.

[1] DistViewer is available at http://cuda.dcc.ufmg.br/siot/


SloT webpage - Mozilla Firefox
SloT webpage
cuda.dcc.ufmg.br/siot/
Search
Compiler's Laboratory
Department of Computer Science - Universidade Federal de Minas Gerais - Brazil

Channel Inference Analysis

On this page you can experience the **DSCV - Distributed System Code Viewer**. Provide programs that communicate over the network in the boxes below and get control flow graphs and dependency graphs of the system as a whole.
DSCV uses SIoT that is a framework to analyze networked systems. SIoT's key insight is to look at a distributed system as a single body, and not as separate programs that exchange messages. By doing so, it can crosscheck information inferred from different network programs. For more information on what SIoT is and what it can do, see our **IPSN 2015 paper**.
SIoT was implemented on top of LLVM, you can access the SIoT's source code **here**.

Type the code in the box below:

```
#include <stdio.h>
#include <sys/socket.h>
#include <string.h>
const size_t MAXDATASIZE = 100;
int main(int argc, char **argv) {
   char buffer[10];
   scanf ("%10s",buffer);
   send(1, buffer, 10, 0);
   if(buffer[0]) {
      send(2, buffer, strlen(buffer), 0);
   }
```

Network Functions:
Send Function: send
Recv Function: recv

Type the code in the box below:

```
#include <stdio.h>
#include <sys/socket.h>
#include <string.h>

const size_t MAXDATASIZE = 100;

int main(int argc, char **argv) {
   char buffer[5];

   recv(1, &buffer, MAXDATASIZE, 0);
   if(buffer[0]) {
```

Network Functions:
Send Function: send
Recv Function: recv

Figure 4.5. DistViewer Initial Screen.

The current version of DistViewer takes as input two C programs that communicate over a network (Fig. 4.5). The tool compiles these source codes using clang Lattner and Adve [2004], analyzes the intermediate code using the DSA framework and renders graphs. All these actions are performed automatically, i.e., without human intervention. In other words, users can analyze their programs without the need to install or configure compilers and other libraries. The final product of DistViewer is an online interface to visualize the graphs it produces. These graphs can also be downloaded for being further processed.

To summarize the modus operandi of our tool, we list a few of its key features: it (i) reads source code in C; (ii) compiles them using the LLVM infrastructure; (iii) merges bytecodes using the DSA framework; (iv) formats and displays, via a web interface, the network graphs (Send-Graph and Receive-Graph); (v) formats and displays the DDG; and (vi) extracts the DDG summary, which we call NetSlice. The NetSlice highlights the interactions between data and network. We explain NetSlice below.

In the rest of this section, we shall use an example to illustrate how DistViewer works. This example will use the programs P_A and P_B of Fig. 4.6. P_A (Fig. 4.6a) has five SENDs and two RECVs. P_B (Fig. 4.6a) has five RECVs and two SENDs. To initiate the generation of graphs, the user must list the name of functions – present in the C

```c
const size_t MAX = 100;


int main( ) {
 char buf[MAX];
 scanf ("%10s",buf);
 size = strlen(buf);
 send(1,buf,size,0);
 if(buf[0]) {
  send(2,buf,size,0);
 }

 if(buf[1]) {
  send(3,buf,size,0);
 } else {
  send(4,buf,size,0);
 }
 send(5, buf, size, 0);
 recv(1,&buf,MAX,0);
 recv(2,&buf,MAX,0);
 return 0;
}
```

(a) Program A (P_A)

```c
const size_t MAX = 100;


int main( ) {
 char buf[MAX];




 recv(1,&buf,MAX,0);
 if(buf[0]) {
  recv(2,&buf,MAX,0);
 } else {
  recv(3,&buf,MAX,0);
 }
 recv(4,&buf,MAX,0);
 recv(5,&buf,MAX,0);
 int size = strlen(buf);
 send(1,buf,size,0);
 send(2,buf,size,0);
 return 0;
}
```

(b) Program B (P_B)

Figure 4.6. Network programs in C language.

source – that perform network accesses (Fig. 4.5). These functions will be regards by DistViewer as either SENDs or RECVs.

The first output of DistViewer is a view of the control flow graph of SENDs (Send-Graph) and RECVs (Receive-Graph) of each program. For instance, Fig. 4.3 shows the Send-Graph of P_A and in the Fig. 4.3 we have the Receive-Graph of P_B. Besides, the Send-Graph of program P_B is shown in Fig. 4.7; and Fig. 4.7 outlines the Receive-Graph of the program P_A. With these graphs, developers or protocol designers have a whole view of the distributed program flow, which allows them to abstract away implementation details. Such abstraction not only makes the analysis of the protocol easier, but also assist designers and developers to detect errors and/or optimize network interactions.

The DistViewer therefore processes the raw DDG and, subsequently, extracts the corresponding NetSlice. The NetSlice is a programming slice that summarizes the DDG focusing on network interactions. This programming slice is represented by a graph which highlights the vertices that represent the network operations (i.e., SENDs and RECVs) or the memory addresses that contribute to the flow of information across the network. More formally, given a DDG $\mathcal{G}$, we define its associated NetSlice $\mathcal{N}$ as

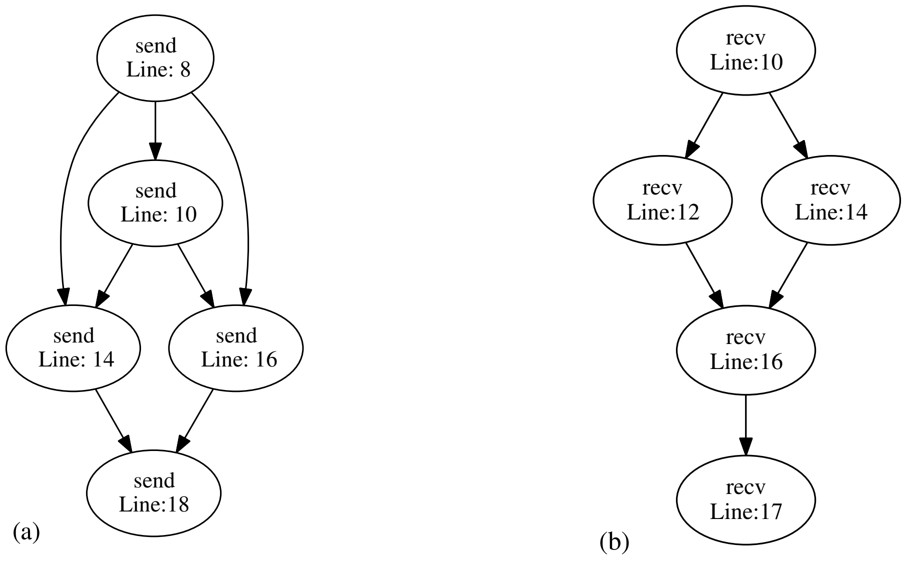


Figure 4.7. P_A's Send-Graph (a) and P_B's Receive-Graph (b).

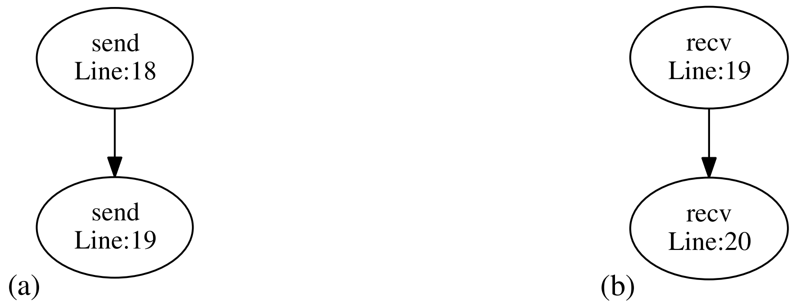


Figure 4.8. P_B's Send-Graph (a) and P_A's Receive-Graph (b).

follows: For each vertex $v \in \mathcal{G}$, we add a vertex v to $\mathcal{N}$ if and only if: (i) v is labeled as a SEND, or (ii) v is labeled as a RECV, or (iii) v is labeled as a *memory* data. Edges in $\mathcal{N}$ correspond to paths in the original $\mathcal{G}$. For each pair of vertex $u, v \in \mathcal{G}$, we add an edge $\overrightarrow{u'v'}$ to $\mathcal{N}$ if, and only if: (i) there exists a path p from u to v in $\mathcal{G}$, and (ii) p does not contain any other vertex in $\mathcal{G}$. Fig. 4.9 shows the NetSlice $\mathcal{N}$ of the programs P_A and P_B (Fig. 4.6).

Note this view of the whole system enables the developer to tell which SENDs of a program can exchange information with RECVs of its counterpart. Further, DistViewer allows one to check dependencies between different memory regions, even if they are located in independent network nodes.

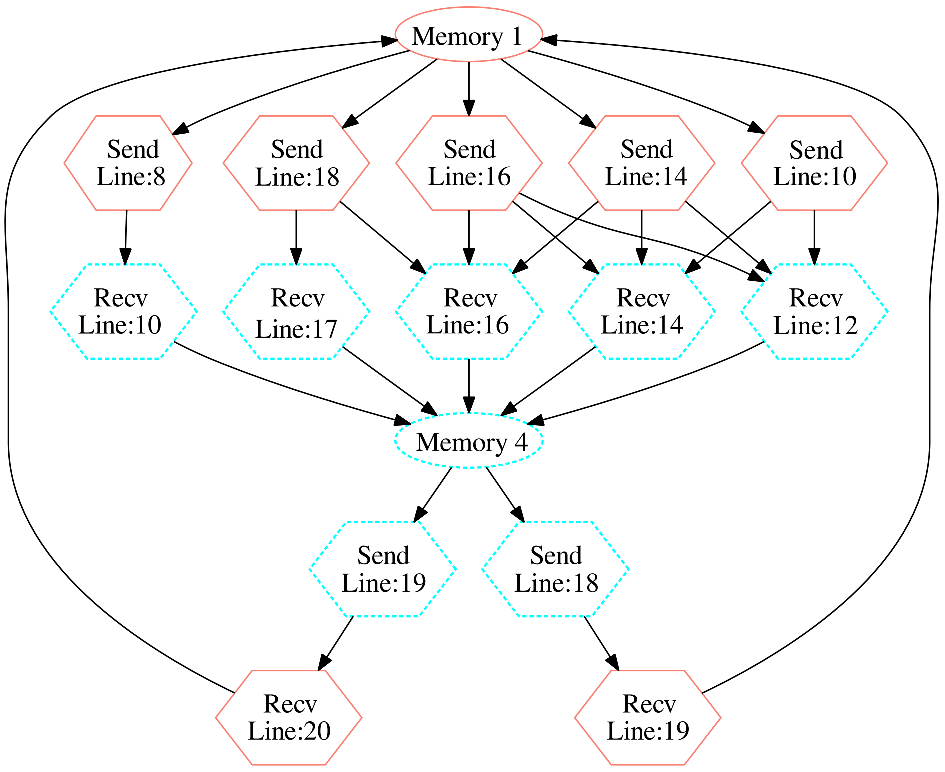


Figure 4.9. NetSlice of Program P_A and Program P_B. Vertices of P_A are represented with continuous salmon line and vertices of Program P_B with dashed cyan line.

4.3.1 DistViewer Architecture and Implementation

The DistViewer architecture comprises four componets (Fig. 4.10): (i) Compile, (ii) DSA program inference links and generation of raw data graphs; (iii) Programming Slices Constructor; and (iv) Transform and View.

Firstly, by the *Compile* component, the DistViewer processes the user input and performs the necessary validations of the program source code.

Next, the DistViewer triggers the LLVM to compile the code and generate the corresponding *bytecode*. DistViewer then generates the configuration files and uses the DSA to obtain the raw data of graphs: Send-Graph, Receive-Graph, DCFG and DDG.

Using the raw data graphs as input, the *Programming Slices Generator* extracts the corresponding programming slices, namely: Send-Graph, Receive-Graph, Distributed Dependence Graph and the NetSlice.

Finally, *Transform View* formats the program slices and generates *pdf* and *html*

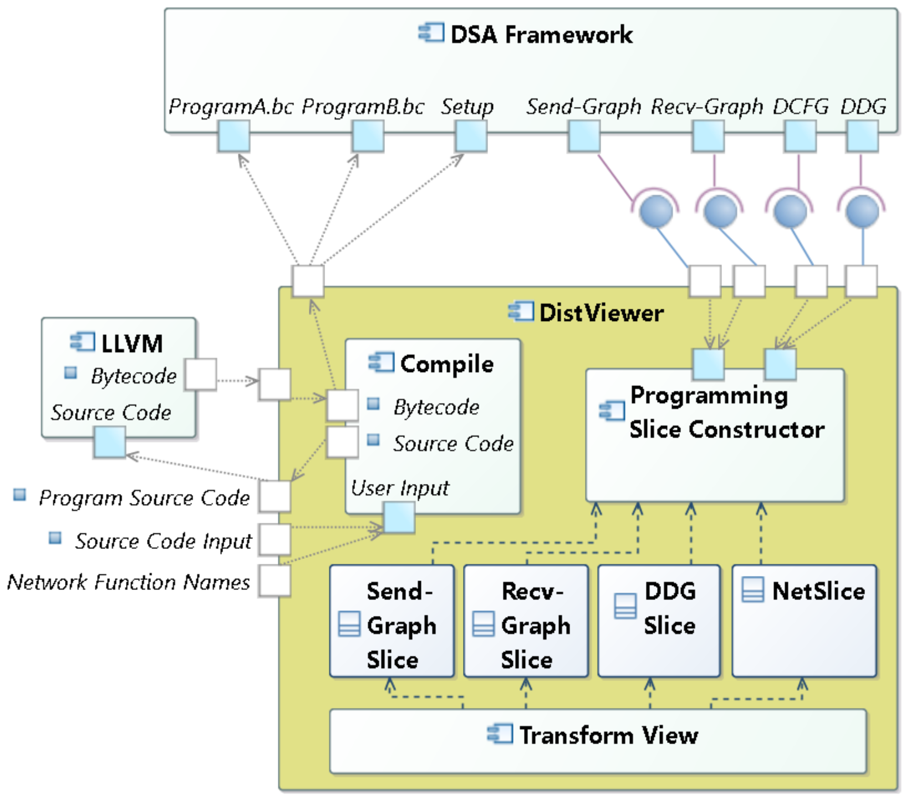


Figure 4.10. DistViewer Architecture.

files which are therefore displayed as output to the user via *web* interface.

4.4 Chapter Summary

In this chapter we present the architecture and implementation aspects of our DSA framework. The architecture of the DSA framework is organized in three layers to enable the development of extensions of the framework as a whole or only part of it.

The first layer, *Merging*, provides as output a program that is the junction of the distributed system programs. This output is a program that represents the distributed system as a whole. Through this output, it is possible to extend conventional static analysis structures (as we did for the CFG and the DG) or create new structures (as we did with the Send-Graph and Receive-Graph).

The second layer, *Linking*, provides structures used in conventional static analysis extended to the context of distributed systems. Such structures can be used as a basis to extend conventional analyzes for treating distributed systems. For example, the

DDG and DCFG, supplied as the output of the second layer, can be used to develop a distributed integer overflow analysis or a distributed range analysis. Another example: our DistViewer extends the first and second layers to provide the programmer an overview of programs focusing on the interaction over a network.

The third layer, *DSA Instance*, is composed by specific static analysis like the SIoT. SIoT extends the DSA and performs a tainted flow analysis in order to detect buffer overflows. This layer can be extended or used for other types of analysis such as, for example, analysis of leakage of confidential information.

We have implemented the DSA as an extension of the LLVM compiler. Therefore, the DSA can be directly applied to distributed systems made in C. No change is required in the source code of the programs that make up the distributed system, even to include annotations in the code (as is required on many static analysis solutions). The source code is available online.

In next Chapter (Chapter 5) we show that, to validate our solution, we have applied the SIoT and DistViewer on six applications present in Contiki OS. The results show that SIoT is more energy-efficient than existing solutions. The results also show that DistViewer can provide graphs of sending and receiving messages of the programs and the interrelationship between them, and provide program slices of distributed system with size equivalent to 5% of the original program.

ï»¿

Chapter 5

Experiments

In the present section we describe the results of experimental evaluation of our proposal. We start (Section 5.1) by introducing the applications used in the experiments and the cost of the static analysis that we have implemented in terms of memory and computational time.

We then (Section 5.2) compares the number of ABCs eliminated by SIoT versus other approaches. Firstly (Section 5.2.1) we present a case study: the implementation of `udp-ipv6` that ContikiOS uses. This application lets us illustrate how false positives can be avoided by our approach. After that (Section 5.2.2), we discuss the results of experiments about the number of ABCs that we can reduce using our approach.To demonstrate that each ABC that we eliminate represents a small saving in terms of energy, we present (Section 5.3) the energy budget of the code that SIoT helps us to produce due to reduction of ABCs.

Next (Section 5.4), we descibe experiments and results about Asymptotic Complexity of the Elevator algorithm that we propose in Section 3.1. We have run the Elevator on hundreds of C programs that we have generated randomly.

Finally (Section 5.5), we present results about the ability of DistViewer to process real programs and the reduction of graphs sizes through the DistViewer slices.

5.1 Benchmark – Static Analysis Time and Memory

To conducted our experiment we choose Contiki applications as benchmark. Contiki is an open source operating system for the Internet of Things [Dunkels et al., 2004]. Contiki connects tiny low-cost, low-power microcontrollers to the Internet. Contiki applications are written in standard C, with the Cooja simulator Contiki networks can be emulated before burned into hardware, and Instant Contiki provides an entire development environment in a single download. There are plenty of applications examples in the Contiki source code tree. Some examples show how to program network code, others show how to interact with the platform hardware, yet others demonstrate different aspects of the Contiki system.

Contiki provides a full IP network stack, with standard IP protocols such as UDP, TCP, and HTTP, in addition to the new low-power IPv6 networking standards, including the 6lowpan adaptation layer, the RPL IPv6 multi-hop routing protocol, and the CoAP RESTful application-layer protocol. The Contiki IPv6 stack, developed by and contributed to Contiki by Cisco, is fully certified under the IPv6 Ready Logo program.

Contiki runs on a wide range of tiny platforms, ranging from 8051-powered systems-on-a-chip through the MSP430 and the AVR to a variety of ARM devices.

Among the applications of Contiki we choose six distributed applications (12 programs). We priorize applications that interacts with network and represent standard protocols for IoT or have academic documentation. The applications used in our experiment was:

- NetDB – client and server
- Ping6 and New-ipv6
- Udp-ipv6 – client and server
- Ipv6-rpl-collect – udp-sender and sink
- Ipv6-rpl-udp – client and server
- Coap-client and Rest-server

`NetDB` consists of a client and a set of servers using Antelope [Tsiftes and Dunkels, 2011]. Antelope is a database management system for resource-constrained sensors. Antelope provides a dynamic database system that enables run-time creation and deletion of databases and indexes. The `NetDB client` runs on a sink node, and has a command-line interface through which an operator can submit queries to any server. The `NetDB server` forwards incoming queries from the radio to Antelope, and packs the query result in packets that are streamed back to the client. `NetDB` uses the Mesh module in the Rime communication stack, which provides best-effort, multi-hop communication.

`Ping6`, `New-ipv6` and `Udp-ipv6` are examples of Contiki OS applications that exercices the use of IPv6 on IoT. IPv6 application to the IoT has been being researched since many years [Zanella et al., 2014]. The research community has developed a compressed version of IPv6 named 6LoWPAN [Hui et al., 2010]. It is a simple and efficient mechanism to shorten the IPv6 address size for constrained devices, while border routers can translate those compressed addresses into regular IPv6 addresses. In parallel, tiny stacks have been developed, such as Contiki, which takes no more than 11.5 Kbytes. IPv6 provides an address self-configuration mechanism (Stateless mechanism). The nodes can define their addresses in very autonomous manner. This enables to reduce drastically the configuration effort and cost. IPv6 is fully Internet compliant. In other words, it is possible to use a global network to develop oneâs own network of smart things or to interconnect oneâs own smart things with the rest of the World. Thanks to its large address space, IPv6 enables the extension of the Internet to any device and service. Experiments have demonstrated the successful use of IPv6 addresses to large scale deployments of sensors in smart buildings, smart cities and even with cattle.

`Coap-client` and `Rest-server` are Contiki applications that demonstrate the Constrained Application Protocol (CoAP) protocol. CoAP enables the constrained devices to behave as web services easily accessible and fully compliant with REST architecture [Shelby et al., 2014]. CoAP has been designed as a generic protocol for Lowpower and Lossy Networkss (LLNs) taking into account the features of the underlying architecture [Bormann et al., 2012]. The CoAP has defined by IETF Constrained RESTful Environments (CORE) working group to address specialized requirements such as: multicast support, very low overhead, and simplicity for constrained environments. The CORE working group, instead of blindly making a compression of HTTP, defined a subset of the RESTful specification, making it interoperable with HTTP but also specializing it for so constrained environments. Main features addressed by CoAP are:

- Constrained web protocol specialized to M2M requirements.
- Stateless HTTP mapping through the use of proxies or direct mapping of HTTP interfaces to CoAP.
- UDP transport with application layer reliable unicast and best-effort multicast support.
- Asynchronous message exchanges.

Table 5.1. Size of applications and the DDG analyzed.

Application	Instructions	DDG Vertices	DDG Edges
netdb client/server	57.877	95,656	139,763
ping / new-ipv6	47,422	75,899	113,900
ipv6-rpl-collect udp-sender/sink	48,800	78,365	117,416
ipv6-rpl-udp client/server	48,226	77,128	115,770
udp-ipv6 client/server	48,800	78,365	117,416
coap-client / rest-server	51,258	82,647	123,879

- Low header overhead and parsing complexity.
- URI and Content-type support.
- Simple proxy and caching capabilities.
- Optional resource discovery.

`Ipv6-rpl-collect` (`upd-sender` and `sink`) and `Ipv6-rpl-udp` are Contiki applications that use the IPv6 Routing Protocol for Lowpower and Lossy Networks (RPL). RPL is an IPv6 routing protocol for LLN. Routing issues are very challenging for 6LoWPAN, given the low-power and lossy radio-links, the battery supplied nodes, the multi-hop mesh topologies, and the frequent topology changes due to mobility. Successful solutions should take into account the specific application requirements, along with IPv6 behavior and 6LoWPAN mechanisms. RPL was proposed by the IETF "Routing Over Low power and Lossy (ROLL) networks" working group. ROLL has proposed the leading IPv6 Routing Protocol for LLN based on a gradient-based approach [Winter, 2012; Vasseur et al., 2011]. RPL can support a wide variety of different link layers, including ones that are constrained, potentially lossy, or typically utilized in conjunction with host or router devices with very limited resources, as in building/home automation, industrial environments, and urban applications. It is able to quickly build up network routes, to distribute routing knowledge among nodes, and to adapt the topology in a very efficient way.

Table 5.1 shows the size of applications used in SIoT experiments in terms of instructions and DDG graph size (vertices and edges). Each of these applications has more than 45,000 instructions, including code in the client and server side. On average, our DDGs had approximately 75,000 nodes and 110,000 edges. These numbers demonstrate that the applications have a considerable size to be used as benchmark of our soluction.

SIOT statically analyze programs and pinpoint vulnerable loads and stores. Table 5.2 shows that SIoT's static analysis took, on average, 66 seconds at compile time

Table 5.2. Time and memory spent in SIoT's static analysis.

Application	Instructions	Time (s)	Memory (MB)
netdb client/server	57.877	66.24	210.03
ping / new-ipv6	47,422	63.58	167.36
ipv6-rpl-collect udp-sender/sink	48,800	80.08	173.37
ipv6-rpl-udp client/server	48,226	66.31	169.90
udp-ipv6 client/server	48,800	80.08	167.39
coap-client / rest-server	51,258	54.36	179.68

and used 170MB when applied onto our benchmarks. We conducted experiments on a laptop Intel Core i7 2.2GHz. Memory measurements have been obtained through Valgrind [Nethercote and Seward, 2007]. These results demonstrate that SIoT takes around 1 minute on average for programs of 50 thousand instructions which is a resanable time to compile programs of this size. Moreover, it's important to highlight that this analysis is done offline, in a workstation, therefore no means represents a burden for embedded devices.

5.2 Array-Bound Checks

Array-Bound Checks – ABCs – are an effective technique to prevent attacks that exploit buffer overflows [Chess and West, 2007]. ABCs are tests performed at runtime to ensure that a particular array access is safe. Buffer overflows tools first scan programs' assembly representation to find code snippets containing vulnerabilities; in a second step, they return to the potential vulnerabilities and insert ABCs. While effective in preventing out-out-bound memory accesses from taking place, these proposals impose a significant overhead on compiled programs because ABCs have a cost, in terms of code size, runtime and energy consumption. We have used the SIoT instance (described in Section 4.2) to eliminate part of these checks from applications taken from the ContikiOS operating system [Dunkels et al., 2004]. In the present section we describe these results.

Section 5.2.1 describes a case study: the implementation of `udp-ipv6` that ContikiOS uses. This application lets us illustrate how false positives can be avoided by our approach. Section 5.2.2 compares the number of ABCs eliminated by SIoT versus other approaches.

```
 1 static void tcpip_handler(void){
 2   char *str;
 3   if(uip_newdata()){
 4     str = uip_appdata;
 5     str[uip_datalen()] = '\0';
 6     printf("Response from the
 7            server: '%s' \n", str);
 8   }
 9 }
10 static void timeout_handler(void){
11  static int seq_id;
12  printf("Client sending to: ");
13  PRINT6ADDR(&client_conn->ripaddr);
14  sprintf(buf, "Hello %d from
        the client", ++seq_id);
15  printf(" (msg: %s)\n", buf);
16  uip_udp_packet_send(client_conn,
        buf, UIP_APPDATA_SIZE);
17  ...
18 }
```

```
 1 static void tcpip_handler(void) {
 2  static int seq_id;
 3  char buf[MAX_PAYLOAD_LEN];
 4  if(uip_newdata()) {
 5    ((char*)uip_appdata)[uip_datalen()]=0;
 6    PRINTF("Server received: '%s' from ",
         (char *)uip_appdata);
 7    ...
 8    uip_ipaddr_copy(&server_conn->ripaddr,
          &UIP_IP_BUF->srcipaddr);
 9    PRINTF("Responding with message: ");
10    sprintf(buf, "Hello from the server!
         (%d)", ++seq_id);
11    PRINTF("%s\n", buf);
12    uip_udp_packet_send(server_conn,
         buf, strlen(buf));
13    ...
14  }
15 }
```

Figure 5.1. ContikiOS udp-ipv6 client (left-hand-side) and server (right-hand-side) code snippets.

5.2.1 Case Study: udp-ipv6

Udp-ipv6[1] is an open source application that runs in ContikiOS. It implements a UDP 6LoWPAN[2] client and server. In the UDP Server, (Fig. 5.1 – left-hand-side), messages are received via network by function `uip_newdata()` (line 4), and are used to index the `uip_appdata` array. Traditional tools are likely to consider this array vulnerable, in which case they must introduce an ABC to protect it. Similar approaches would also lead to the insertion of ABCs to protect the arrays accesses of the client module (line 4 and 5, Fig. 5.1 – right-hand-side).

On the other hand, once we analyze these two programs as a single body, we can observe that none of these array accesses is vulnerable. This observation stems from the fact that messages prepared by the client have no dependency from any external data source (Fig. 5.1 – left-hand-side, lines 10-14). Because such messages are not tainted at their origin, according to our attack model (Section 2.4.2), they are not vulnerable at their destination. Similarly, the messages that the server sends to the client do not depend on any input data (Fig. 5.1 – left-hand-side, lines 8-11). Thus, it

[1]http://github.com/contiki-os/contiki/tree/master/examples/udp-ipv6
[2]http://tools.ietf.org/html/rfc6282

is not necessary to insert ABCs on the client side. In other words, any alarm triggered by a traditional static analysis would be a false positive in this example.

5.2.2 Number of ABCs that we insert

The inter-program view that SIoT gives to a static analysis lets it consider network channels as links between modules instead of input operations. Therefore, all the ABCs that depend exclusively on the network and do not reach user inputs can be eliminated. The end result of this extra precision is a more efficient executable code. To validate this claim, we have used SIoT to improve the code that AddressSanitizer (ASan) [Serebryany et al., 2012] generates. To give the reader some perspective on our results, we compare SIoT to a hypothetical traditional tainted flow analysis, i.e., a technique that treats distributed system as separate programs, and not as a single entity. Henceforth, we refer to this technique as Baseline.

We perform the experiments in six pairs of ContikiOS applications (Table 5.1). Each pair consists of a client and a server. For each of these pairs, we compare the number of ABCs that ASan inserts without any optimization against the number of ABCs that the Baseline and the SIoT-based approaches insert. We compare performance of SIoT versus Baseline (Table 5.3) and ASan (Table 5.4) using the number of ABCs necessaries in each approach.

Table 5.4 shows that ASan introduces between 3,736 ABCs (*udp-ipv6 client/server*) and 7,453 ABCs (*ping6 / new-ipv6*). This number is less than the total number of memory accesses because LLVM, the compiler on top of which ASan exists, already eliminates some redundant guards as a result of classic code optimization.

The Baseline approach reduces ASan's numbers substantially (Table 5.3, because in this case we are eliminating every guard that is not influenced by data coming from an external function. In this case, the number of ABCs varies between 170 (*ipv6-rpl-udp client/server*) and 214 (*rest-server/coap-client*). SIoT can further reduce this number one order of magnitude more. In this case, contrary to what is done by the Baseline approach, network functions are no longer marked as dangerous, unless they read data that comes from genuine inputs. Notice that both, the Baseline and the SIoT approaches are a form of tainted flow analysis, as we explain in Section 2. We conclude from these experiments that the automatic inference of links between distributed programs improves the precision of static analyses tools in non-trivial ways.

Table 5.3. ABCs inserted by Baseline versus SIoT.

Applications	Memory Accesses	ABCs inserted		% ABCs Reduction
		Baseline	SIoT	SIoT vs Baseline
netdb client/server	22,819	172	16	90.70%
ping6 / new-ipv6	16,871	166	14	91.57%
ipv6-rpl-collect udp-sender / sink	17,301	168	14	91.67%
ipv6-rpl-udp client/server	17,162	170	14	91.76%
udp-ipv6 client/server	16,945	212	14	93.40%
coap-client / rest-server	18,693	214	14	93.46%

Table 5.4. ABCs inserted by ASan versus SIoT.

Applications	Memory Accesses	ABCs inserted		% ABCs Reduction
		ASan	SIoT	SIoT vs ASan
netdb client/server	22,819	4,641	16	99.66%
ping6 / new-ipv6	16,871	7,453	14	99.81%
ipv6-rpl-collect udp-sender / sink	17,301	3,831	14	99.63%
ipv6-rpl-udp client/server	17,162	3,787	14	99.63%
udp-ipv6 client/server	16,945	3,736	14	99.63%
coap-client / rest-server	18,693	4,032	14	99.65%

5.3 Energy Saving

Each ABC that we eliminate represents a small saving in terms of energy consumption. To back up this observation with data, we performed an experiment with six ContikiOS applications. We have modified the client side of each one of these applications to initiate execution, send 6,000 messages within an interval of one minute, and then stop. We tested three versions of each application: (i) without ABCs; (ii) with the ABCs inserted by the Baseline; and (iii) with the ABCs inserted by SIoT. To carry out this experiment, we have installed the applications in IRIS XM2110 sensors[3] and have measured the amount of energy that they consume. Each application was executed 10 times. This number of repetitions is enough to give us a confidence interval of 95% in every sample.

To determine the amount of energy consumed, we rely on a simple, yet robust methodology, which we adapted from the work of Singh *et al.* [Singh and Kaiser, 2010]. The main difference between our approach and Singh *et al.*'s is in terms of electronics: we probe small sensors; they work on Intel's Atom board. We use a DAQ[4] (NI USB-6009) power meter to measure the instantaneous current between the load and the ground ports (Fig. 5.2). We then send the signal to a software that runs on a separate PC. Since the voltage in the embedded system is constant, this software is able to

[3]`http://www.memsic.com/wireless-sensor-networks/`
[4]`http://en.wikipedia.org/wiki/Data_acquisition`

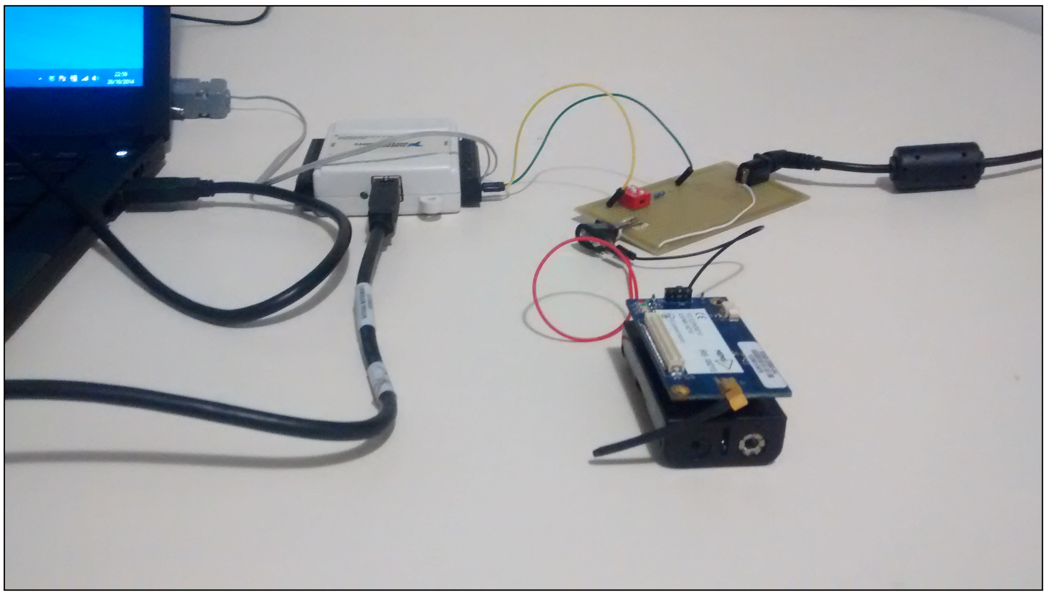


Figure 5.2. Energy Measurement Setup.

calculate the instantaneous power and, by integrating it, the total amount of energy that the program consumes.

A shunt resistor is used to probe the current. To determine the shunt resistance, R_{sen}, it is necessary to calculate the load resistance. It is important that R_{sen} does not influence the total system load. To ensure this non-interference, R_{sen}'s value needs to be 1% of R_{Load}'s value. The DAQ actually measures a tension value that is linearly proportional to the current ($V_{sen} = I_{sen}R_{sen}$). Since the R_{sen} value is known, it is possible to calculate the current ($I_{sen} = V_{sen}/R_{sen}$) and the consumed energy.

The consumed energy is finally calculated with Equation 5.3. FilterData, included on *AutomatedDataAnalysis*, implements a second order low-pass filter (Butterworth) to filter the measured signal noise. It detects the cut-off frequency automatically.

$$E = \int_{t_i}^{t_f} p(t)dt = \int_{t_i}^{t_f} v(t)i(t)dt = V\int_{t_i}^{t_f} i(t)dt \tag{5.1}$$

$$R_{Load} = \frac{P_{Mean}}{I_{Max}^2} \tag{5.2}$$

$$E = V\int_{t_i}^{t_f} i(t)dt = V\frac{1}{R_{sen}}\int_{t_i}^{t_f} v_{sen}(t)dt \tag{5.3}$$

We manipulate this data using a signal processing software of our own craft. This tool filters the electric signal, identifies the time when execution starts and ends, and calculates the energy consumed by the application with a given confidence interval.

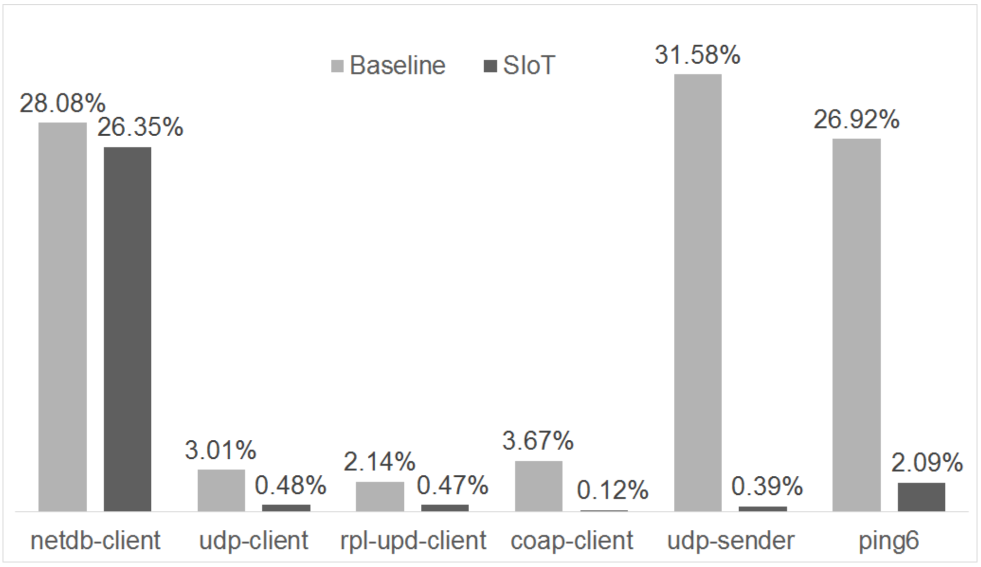


Figure 5.3. SIoT vs Baseline – energy overhead.

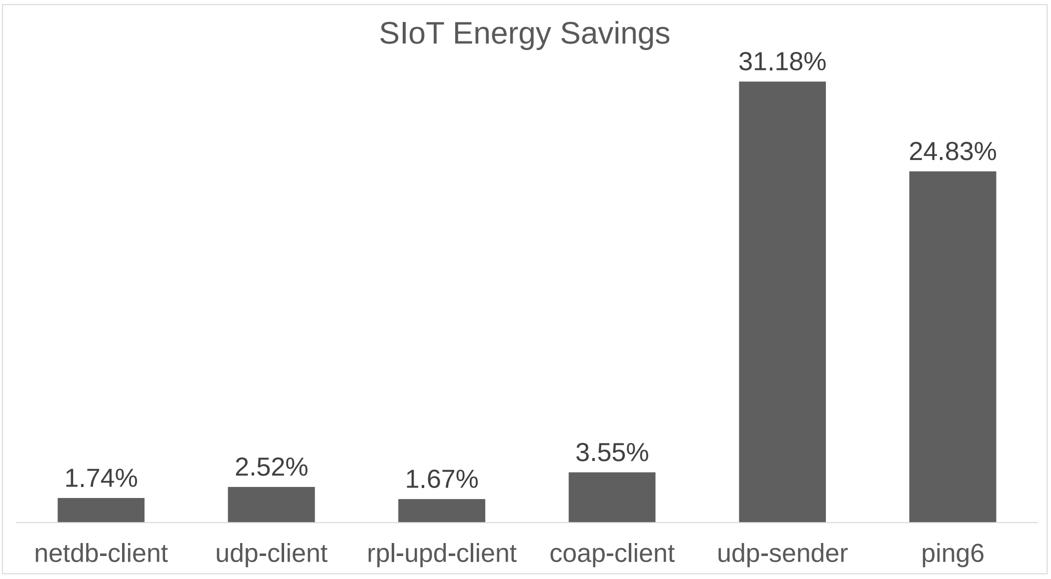


Figure 5.4. SIoT vs Baseline – energy savings.

Our tool is able to discriminate multiple executions of the same application by sending signals to the sensor that we are sampling. These signals mark the moment when each execution starts and finishes.

Our results (Table 5.5, Fig. 5.3 and Fig. 5.4) show that SIoT outperforms Baseline for all applications. The SIoT's savings range from 1.67% (*rpl-udp-client*) to 31.58%

Table 5.5. Energy consumption for the unprotected (Plain) and protected (SIoT and Baseline) versions of applications. CI: Confidence Interval.

Application	Plain		SIoT		Baseline	
	Energy (J)	CI	Energy (J)	CI	Energy (J)	CI
netdb client	1.941	0.025	2.452	0.014	2.486	0.011
ping6	0.109	0.001	0.111	0.001	0.151	0.002
ipv6-rpl-collect udp-sender	2.277	0.043	2.286	0.043	2.996	0.029
ipv6-rpl-udp udp-client	3.062	0.029	3.076	0.016	3.127	0.005
udp-ipv6 client	3.842	0.019	3.860	0.011	3.958	0.029
coap-client / rest-server	4.856	0.020	4.861	0.037	5.034	0.041

(*ipv6-rpl-collect udp-sender*). On average SIoT is 18% more energy efficient than Baseline. The amount of energy consumed is proportional to ABCs inserted that are executed. This energy overhead is due to the fact that for each ABC the programs need to execute at least six extra instructions. Moreover, while the original instruction correspond a simple load or store, the new instructions sets has jumps and conditional logic that increase the computational cost. In an embedded device where the program is small and the instructions are optimized to save energy, this extra processing has a significant impact per inserted ABC. As the difference of inserted ABCs in Baseline and SIoT approaches is significant, the final difference of energy overhead is high.

5.4 Asymptotic Complexity

To estimate the asymptotic complexity of the algorithm that we propose in Section 3.1, i.e., the “Elevator", we have run it on hundreds of C programs that we have generated randomly. We generate these programs by successive applications of three rewriting rules. These rules work on *program points*. Each program point contains either a SEND or a RECV function.

Our rewriting rules may transform a program point into: (i) a sequence of two program points; (ii) an “if-then-else" with a point in the “then" path and a point in the “else" path; (iii) a “while" loop with one program point in its body.

We always generate two programs in tandem, a client and a server, in such a way that there exists a valid path linking SENDs and RECVs. The randomized pieces of code with RECVs come in nests of “while" and “if-then-else" blocks. We can produce programs arbitrarily large by varying the number of program points. Therefore, the worst case was approximated.

We run the Elevator 10 times on each synthetic program, reporting the average of these samples. From these points, i.e., size versus runtime, we produced the chart

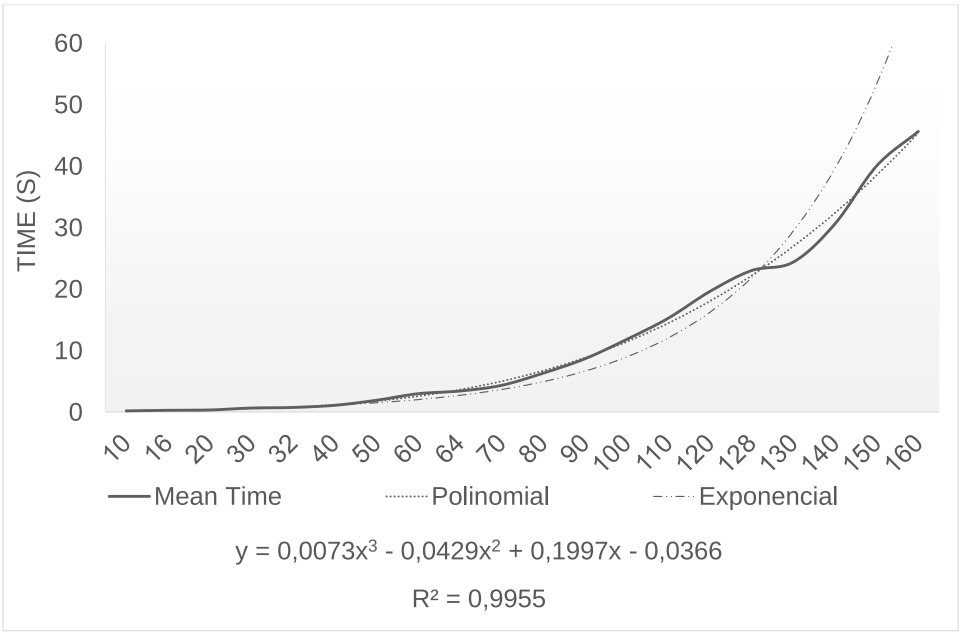


Figure 5.5. Elevator – Runtime as a function of program size.

in (Fig. 5.5). This figure suggests that our algorithm has cubic complexity, as its coefficient of cubic regression, given a universe of 160 samples, is 0.9955; hence, too close of 1.0. Thus, even though we have determined an exponential upper bound to the asymptotic complexity of our algorithm (Section 3.4), in practice it has polynomial behavior, at least for programs with around 200 pairs of sends and receives. We have not found a single benchmark among the actual applications that we have analyzed with more than 20 such pairs.

These complexity results are directly related to the numbers that we have observed when analyzing actual programs. Thus, the techniques that we describe in this work seem feasible in practice.

5.5 DistViewer Graph Size Test

In order to evaluate the ability of DistViewer to process real programs we have conducted tests on six *ContikiOS* [Dunkels et al., 2004] applications. The Table 5.6 summarizes the results obtained. In addition to demonstrating the viability of running DistViewer using real programs it was observed that the NetSlice was reduce the number of DDG vertices to around 5%. This reduction is made possible by the focus on vertices of network-related operations (SENDs and RECVs) and vertex that manipulate memory data.

Table 5.6. DDG Vertices: Baseline versus DistViewer NetSlice graph

Program	**Baseline**	**DistViewer**	**%**
netdb client/server	95,656	5,070	5.30%
ping / new-ipv6	75,899	3,719	4.90%
ipv6-rpl-collect udp-sender/sink	78,365	3,918	5.00%
ipv6-rpl-udp client/server	77,128	3,934	5.10%
udp-ipv6 client/server	78,365	3,918	5.00%
coap-client / rest-server	82,647	4,711	5.70%

5.6 Chapter Summary

In this chapter we have shown, as an example, the results of experimental evaluation of our proposal. We describe the Contiki OS applications used as benchmarks in our experiments and the cost of the static analysis that we have implemented in terms of memory and computational time.

We have compared the number of ABCs eliminated by SIoT versus other approaches. We have presented a case study using the implementation of Contiki OS `udp-ipv6`. This application lets us illustrate how false positives can be avoided by our approach. We have discussed the results of experiments about the number of ABCs that we can reduce using our approach. We then demonstrated that each ABC that we eliminate represents a small saving in terms of energy, we present the energy budget of the code that SIoT helps us to produce due to reduction of ABCs.

We also have descibed experiments and results about Asymptotic Complexity of our Elevator algorithm. We have run the Elevator on hundreds of C programs that we have generaterandomly and the results has shown that the Elevator is polynomial in practice.

Finally, we have present results about the ability of DistViewer to process real programs and the reduction of graphs sizes through the DistViewer slices.

In next chapter (Chapter 6) we present a review of related works.

Chapter 6

Related Work

In this chapter,we present a review of related works. We start explaining the literature review methodology adopted (Section 6.1). We then (Section 6.2) discuss related work breaking them into three categories: (i) Networked Embedded Systems Code Analysis (Section 6.2.1), (ii) Inference of communication links (Section 6.2.2), and (ii) Runtime Analyses (Section 6.2.3).

6.1 Literature Review Methodology

Our work permeates different areas of research on computing such as Compilers (code optimization, security of code), Distributed Systems, and System Security. Thus, our research methodology was divided into two parts: (i) Compilers Literature Review; and (ii) Systems Literature Review. Initially, we have reviewed the compilers literature search for works related to code optimization security applied to computer networks, parallel or distributed systems. We then reviewed the literature related to Systems (distributed, parallel, and security) search of work related to code analysis. We then obtained a list of works related to distributed systems' code analysis.

Related works of this research have been collected from electronic databases (IEEE Xplore, ACM Digital Library, USENIX and Elsevier Science Direct) including peer reviewed articles and papers describing research on Distributed System Code Analysis published in journals, magazines or conferences with recognized reputation in publishing high quality content and excluding short papers, technical reports and posters.

We have defined our period of interest as Jan. 2004 to Aug. 2015, but we have included some classical works with the fundamental compiler concepts used in this work. In order to select the jobs directly related to this work, we define selection criteria relevant to systems security code, networked embedded systems, distributed embedded systems, and inference links, as listed below:

1. Stand Alone Program Analysis: e.g., Buffer Overflows, Tainted Flow Analysis, Memory Safety;

2. Network Embedded Systems Code Analysis: e.g., Code analysis of Embedded Systems, Internet of Things, Wireless Sensor Network (WSN);

3. Distributed Systems Analysis: e.g., Static Analysis of Distributed and Parallel System, Communication Links Inference, Message Passing Interface (MPI), MPI Interprocedural CFG (MPI-ICFG).

As keywords used in the first stage of search we can cited as example: "Network" & "Control Flow Graph", "Distributed" & "Control Flow Graph", "Distributed Systems Compiler", "Distributed Systems" & "Static Analysis ", "WSN" & "Static Analysis", "IoT" & "Static Analysis", "Embedded Systems" & "Static Analysis","MPI-CFG", "MPI-ICFG", "Petri Nets & Static Analysis", "Security & Embedded Systems", "Security & IoT", "Security" & WSN", "Buffer Overflow & Network Embedded Systems".

Using the search criteria we have retrieved the works with matches in title, abstract or keywords using the search engine of each publisher (e.g., IEEE Xplore, ACM Digital Library, USENIX and Elsevier Science Direct) or search via Google Scholar[1]. For each keyword search we choose by title around 10 works. The selected papers, around 350 works, were then filtered by abstract. In order to validate the search, we have conducted an investigative review in order to eliminate papers that match only one criteria but is not direct related to our work. Moreover, some works were found through the citations of works previously found. The related work were, then, evaluated in depth and report in this work.

We split each category into groups of related work. The fig 6.1 shows the categories, groups and some examples of works of each group. We also have insert part of our contributions to show where our work fits in this scenario. The first category, *Stand Alone Program Analysis*, includes the works with the fundamental concepts of code analyses and has been subdivided in traditional and recent works. (For a discussion of Stand Alone Code Analysis please see Chapter 2). The second category, *Networked Embedded System Code Analysis*, includes works about buffer overflow and tainted flow analysis applied to embedded systems or sensor networks (Section 6.2.1).

[1]`http://scholar.google.com`

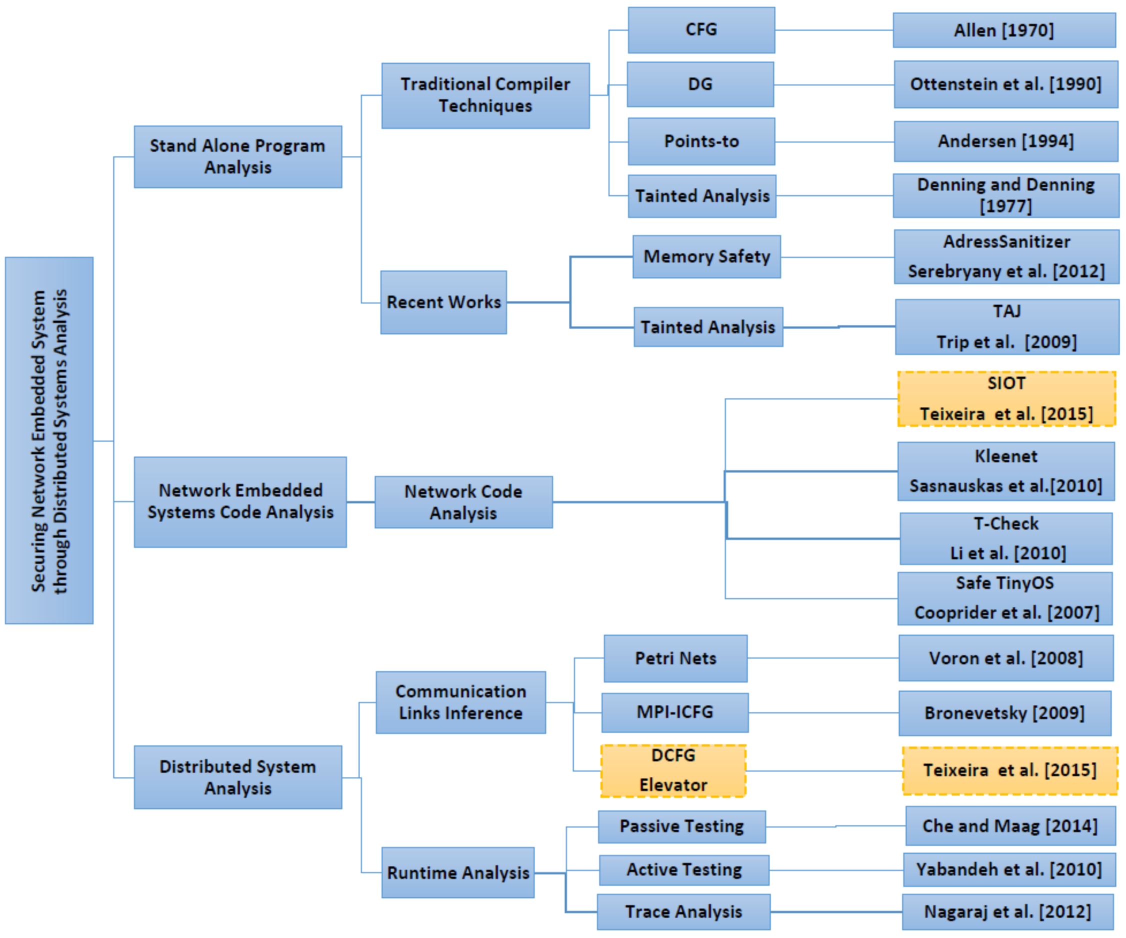


Figure 6.1. Related Works Categories. We have cited one work for each category as an example. We also have inserted part of our contributions to show where our work fits (orange boxes with dashed lines).

Finally (Section 6.2), the third category, groups the works focus on Distributed or Parallel Systems and has been divided in Static Analysis and Runtime Analysis (analysis of deployed systems and trace analysis).

6.2 Distributed Systems Analysis

In what follows, we first describe works on Network Embdded Systems Code (Section 6.2.1) and then works on Inference Communication Links (Section 6.2.2).

6.2.1 Network Embedded Systems Code Analysis

Cooprider *et al.* propose a memory safety solution for sensor nodes named Safe TinyOS [Cooprider et al., 2007]. It handles array and pointer errors before they can corrupt the RAM. Safe TinyOS check array bound for sensor nodes to detect memory bugs for TinyOS 2 applications running on the Mica2, MicaZ, and TelosB platforms. Safe TinyOS uses Deputy to insert annotations and cXprop as its static analyzer and source-to-source optimizer for embedded C programs. Safe TinyOS is light enough to be embedded in sensor nodes. However it also sees each program of a distributed system individually, and the user needs to insert annotations in the code.

There are also proposals for constrained networks that analyze codes of distributed systems based on test generation. Among these, we highlight Kleenet [Sasnauskas et al., 2010] and T-Check [Li and Regehr, 2010].

Kleenet is a debugging environment for testing of sensor network applications before deployment. Its goal is to enable the detection of bugs that result from interactions of multiple nodes, nondeterministic events in the network, and unpredictable data inputs. Kleenet is built on the symbolic virtual machine KLEE [Cadar et al., 2008]. It considers symbolic input values from the environment and generates execution paths of participating nodes at high-coverage. KleeNet injects symbolic, nondeterministic events such as loss, duplication and corruption of packets and node failures automatically. If an execution path violates an assertion, KleeNet automatically generates a test case to reproduce the bug.

The goal of T-Check is similar to Kleenet, i.e., find safety and liveness errors in sensor network applications but its focus on applications running on TinyOS. T-Check employs model checking and random exploration, it uses random walks and explicit state model checking to look for violations of safety and liveness properties in TinyOS sensor network applications. T-check is building upon TOSSIM (the TinyOS simulator). T-Check gains enough scalability to detect distributed errors such as a collection tree protocol's failure to properly repair when a node dies.

These tools generates tests based on symbolic execution (Kleenet) or model checking (T-check) to find software defects. Their goal is to explore automatically most of the execution paths within programs. If an assertion fails, then the tool registers the test case for repeatability. However in these solutions, the developer needs to add annotations to the code. This step is manual, and requires knowledge of the application logic. Moreover, the solution's complexity depends on symbolic inputs, assertions and the number of nodes. The authors of Kleenet, for example, report that even with relatively small-sized symbolic inputs and few nodes, some applications have thousands of execution paths. SIoT is complementary to Kleenet and T-Check, and we consider our link inference engine could be used to improve the accuracy of those tools.

Lai et al. [2008] propose a framework to test nesC applications based on construction of Inter-Context Flow Graphs. They firstly model the potential execution orders of tasks in a nesC application as task graphs. Based on task graphs, they propose Inter-Context Flow Graphs to model the behaviors of nesC applications for testing purposes. These graphs capture not only control transfers in individual subroutines, but also the interactions among subroutines in concurrent calling contexts initiated by interrupts. We further propose control-flow and data-flow test adequacy criteria based on ICFGs to measure the coverage of test suites for testing the interactions among subroutines. They have evaluate proposed testing criteria using an open-source structural health monitoring application. Experimental results show that their criteria are on average 21 percent more effective in exposing faults than their conventional counterparts. This proposal is close to ours in terms of to extract the CFG. Lai et al. [2008] proposal has as advantage the fact to handle the behavior of nesC application as parallel tasks. However it also sees each program of a distributed system individually and so do not handle the communication among programs.

6.2.2 Inference of communication links

The inference of communication links between different modules of a distributed system is not a new problem, and there are solutions in literature. However, previous approaches were either too costly or semi-automatic. For instance, Pascual and Hascoët [Pascual and Hascoët, 2012] have defined a system of annotations which the user can employ to point out to the compiler implicit communication channels in a distributed system. This approach, although precise – as long as the user correctly understands the application – has the main disadvantage of burdening programmers with a task that, in our understanding, should be solved by the compiler.

6.2.2.1 Petri Nets

Petri Nets is also used to find represent the communication links between different programs. Petri nets (PNs) are formal models developed for modeling of concurrent systems [Murata, 1989; Iordache and Antsaklis, 2009]. They are first introduced by Carl Adam Petri in 1962. Petri Nets are based on strong mathematical foundation and are very similar to State Transition Diagrams. Petri Nets are largely used as a visual communication aid to model the system behavior [Murata, 1989]. Petri Nets can be used a diagrammatic tool to model concurrency and synchronization in distributed systems [Murata, 1989].

Many works uses Petri Nets to analyze the distributed or parallel systems; see, e. g., Balbo et al. [1992], Voron and Kordon [2008], Liao et al. [2013], and Fan et al. [2012]. A review of the application of Petri Nets to computer programming is presented by Iordache and Antsaklis [2009].

Although the Petri Nets can represent the distributed systems with more precision and more details than our approach, the construction of Petri Nets need a significant burden placed on the analyst in order to specify complex models and the graphical representation may become too complex to be useful. Another disadvantage is that their representation of priorities or ordering is hard to manage.

For instance, Voron and Kordon [2008] propose an approach to perform transformation of source code (C programs) into Petri nets, as a suitable specification for model checking. They use the CFG to derive Petri Nets. However, the authors report that to overcome the complexity of the resulting specification they need to focus on specific aspects of the program. Several transformations must be performed to verify each aspect of the processed program.

In the rest of this section we describe only automatic approaches to tackle with the inference of communication links problem.

6.2.2.2 MPI-ICFG

Message passing via MPI is widely used in parallel programs [Strout et al., 2006; Friedley et al., 2013; Gopalakrishnan et al., 2011; Bronevetsky, 2009]. Some works proposes a way to construct a MPI-ICFG which is an ICFG augmented with communication edges between possible send and receive pairs and partial context sensitivity [Strout et al., 2006; Bronevetsky, 2009; Pellegrini, 2011].

Among the fully automatic solutions, the work that is the closest to ours is Bronevetsky's analysis, which finds a matching between sends and receives in an MPI program [Bronevetsky, 2009]. His analysis is more precise than ours, for it takes the semantics of MPI into consideration. It executes the program symbolically, separating processes by their IDs.

This precision has a cost: the channel inference may take too long to converge, as loops, for instance, may lead to the generation of many different symbolic sets. As a consequence of this cost, Bronevestsky's analysis has not, thus far, being applied on large code bases. Our technique, on the other hand, trades precision for speed. Hence, as we have demonstrated in Section 5.4, our asymptotic complexity in practice is cubic on the number of sends and receives present in the target system.

Pellegrini, in his PhD dissertation [Pellegrini, 2011], has expanded Bronevestsky's ideas to deal with features of MPI programs that the latter could not handle. He relies on the polyhedron model [Feautrier, 1996] to divide processes into matching sets, again relying on the process ID as a symbol with semantic value within the programming language. Pellegrini evaluates his technique on a suite of small MPI programs.

We believe that his technique is even more precise than Bronevestsky's; however, we speculate that similar to it, Pellegrini's analysis may not scale up to very large code bases. The difficulty is the same: the more processes we may have, and the more complicated is the program's CFG, the higher the number of matches that are possible.

Instead of precise results, we provide an approximation of the possible communication links in a distributed program. Our results may present more false positives than Bronevestsky's or Pellegrini's approaches, but we run faster. Additionally, contrary to these works, we bring forth formal proofs that our algorithm terminates, and we describe an empirical study of its complexity.

6.2.3 Runtime Analyses

In addition to static analyses such as Bronevestsky's and Pellegrini's, the literature also contains works that infer communication links between programs by studying the traces of instructions that these programs produce during execution [Chen et al., 2006; Shende and Malony, 2006; Preissl et al., 2008; Wu and Mueller, 2011]. We call such approaches *runtime analyses.*

The solution can be subdivided in passive testing [Che and Maag, 2014; Sunyé et al., 2014; Mouttappa et al., 2013], active testing of deployed applications [Yabandeh et al., 2010; Liu et al., 2008] and dynamic analyses based on the evaluation of previous executions traces [Nagaraj et al., 2012; Preissl et al., 2008; Ayers et al., 2005].

The main advantage of these approaches is precision: they never produce false positives, as every link inferred over execution traces represents an actual exchange of messages. On the other hand, post-mortem methods have a number of disadvantages. In particular, they are unsound, given that they may not point out every implicit communication link in a distributed system. In other words, their precision depends on the inputs that are used to test a program, and these inputs may not cover every possible path within the program's CFG. A second disadvantage is their computational cost: programs can generate very large traces, which are difficult to store and process.

6.3 Chapter Summary

In this chapter, we present our literature review and discuss that even solutions for network systems generally analyze the distributed programs separately.

We have described some studies that analyze the distributed systems as whole through test generation and symbolic execution, however the strategy we propose requires less computational resources.

We also have discussed works aimed at messaging passing in parallel systems. These works are related to an important part of our work that is analyze different programs as one using inference of communication links between programs.

Finally, we have discussed works that infer communication links between programs in a distributed system during runtime. These works analyze traces of instructions that these programs produce during execution.

In the next chapter (Chapter 7) we present our final remarks and proposal of future work.

Chapter 7

Conclusion

This work has presented a general framework for analysis of distributed systems programs, which has been customized for implement: (i) an efficient solution to counter buffer-overflow attacks in Networked Embedded Systems, and (ii) a tool to build programming slices of distributed systems.

Our key insight is to look at a distributed system as a single entity, and not as separate programs that exchange messages. By doing so, we can crosscheck information inferred from different programs. This crosschecking increases the precision of traditional static analyses.

To validate this claim, we have implemented our framework on top of the LLVM compiler, and have developed two instances. Firstly, we use our framework to instantiate a version of tainted flow analysis that points out which memory accesses need to be guarded against buffer overflow attacks. Our experiments have demonstrated that our approach is energy-effective and useful to make programs running over a network safer. The other instance uses the graphs generated by the framework to generate a code view that highlights the part of code that has dependency of network.

This work brings forth both theoretical and practical contributions, more specifically:

1. We propose an extension to the standard Control Flow Graphs (CFGs) [Allen, 1970], called Distributed Control Flow Graph (DCFG), that is expressive enough to model the control flow spanning multiple programs that communicate over a network.

2. We propose an algorithm that infers communication links between different programs from a distributed system, and prove that the algorithm (i) never misses

possible communication paths between programs; and (ii) always reaches a fixed point, and hence always terminates.

3. We have implemented our algorithm and its companion distributed tainted flow analysis in the LLVM compiler [Lattner and Adve, 2004].
4. We have applied this analysis on six applications present in ContikiOS [Dunkels et al., 2004], and the results show that our proposal is 18% more energy-efficient than existing solutions.
5. We have developed a tool to generate a code view (programming slices) that highlights the part of code that has dependency of network.

7.1 Future Work

We believe that our framework and the techniques propose here can be used to enable other security analysis, compiler optimizations and defects detection in distributed system or network protocols.

In particular, we are interested in using it to secure programs against errors caused by Integer Overflow (IOF) [Rodrigues et al., 2013]. Saggioro et al. [2015] and Paisante et al. [2014] have done the first steps. They use our framework to extend the traditional range analysis to a new distributed range analysis. This is an important step to adapt traditional IOF static analysis to distributed system programs.

Moreover, we believe that Paisante et al. [2014] segmentation messages also can be used to improve our SIoT Tainted Flow Analysis. If we can segment the buffer using a distributed range analysis, we can use this information to reduce the number of tainted flows or the number of ABC in each tainted flow.

We also want to use our framework to enable compiler optimizations. As an example, if we go back to Figure 2.3, we see that the conditional test at line 2 of our server is unnecessary if you have a inter-program view of the system.

Finally, we believe that the Send-Graph and Receive-Graph discussed in Chapter 3 and Chapter 4 can be used to automatically pinpoints bugs in network protocols implementation. For instance, with DistViewer programming slices is possible to warn the protocol developer about RECV without SEND and vice-versa.

Bibliography

Allen, F. E. (1970). Control flow analysis. *ACM Sigplan Notices*, 5:1--19.

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

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

Ayers, A., Schooler, R., Metcalf, C., Agarwal, A., Rhee, J., and Witchel, E. (2005). Traceback: first fault diagnosis by reconstruction of distributed control flow. *ACM SIGPLAN Notices*, 40(6):201--212.

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.

Balbo, G., Donatelli, S., and Franceschinis, G. (1992). Understanding parallel program behavior through petri net models. *Journal of Parallel and Distributed Computing*, 15(3):171--187.

Balzarotti, D., Cova, M., Felmetsger, V., Jovanovic, N., Kirda, E., Kruegel, C., and Vigna, G. (2008). Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *Symposium on Security and Privacy (S&P)*. IEEE.

Bhatkar, E., Duvarney, D. C., and Sekar, R. (2003). Address obfuscation: an efficient approach to combat a broad range of memory error exploits. In *USENIX Security*, pages 105--120.

Borgia, E. (2014). The internet of things vision: Key features, applications and open issues. *Computer Communications*, 54:1--31.

Bormann, C., Castellani, A. P., and Shelby, Z. (2012). Coap: An application protocol for billions of tiny internet nodes. *IEEE Internet Computing*, 16(2):62.

Boukerch, A., Xu, L., and El-Khatib, K. (2007). Trust-based security for wireless ad hoc and sensor networks. *Computer Communications*, 30(11):2413--2427.

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

Buchanan, E., Roemer, R., Shacham, H., and Savage, S. (2008). When good instructions go bad: generalizing return-oriented programming to RISC. In *CCS*, pages 27--38. ACM.

Cadar, C., Dunbar, D., Engler, D. R., et al. (2008). Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209--224.

Che, X. and Maag, S. (2014). Passive performance testing of network protocols. *Computer Communications*, 51:36--47.

Chen, H., Chen, W., Huang, J., Robert, B., and Kuhn, H. (2006). MPIPP: An automatic profile-guided parallel process placement toolset for smp clusters and multiclusters. In *International Conference on Supercomputing.* ACM.

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

Cooprider, N., Archer, W., Eide, E., Gay, D., and Regehr, J. (2007). Efficient memory safety for tinyos. In *Conference on Embedded Networked Sensor Systems (SenSys).* ACM.

Cowan, C., Wagle, F., Pu, C., Beattie, S., and Walpole, J. (2000). Buffer overflows: attacks and defenses for the vulnerability of the decade. In *DARPA Information Survivability Conference and Exposition, (DISCEX).* DARPA.

Criswell, J., Geoffray, N., and Adve, V. (2009). Memory safety for low-level software/hardware interactions. In *SSYM*, pages 83--100. USENIX Association.

Criswell, J., Lenharth, A., Dhurjati, D., and Adve, V. (2007). Secure virtual architecture: a safe execution environment for commodity operating systems. In *SOSP*, pages 351--366. ACM.

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. *Transactions on Programming Languages and Systems, (TOPLAS)*, 13(4):451–490.

Delicato, F. C., Pires, P. F., and Batista, T. (2013). *Middleware solutions for the Internet of Things.* Springer.

Denning, D. E. and Denning, P. J. (1977). Certification of programs for secure information flow. *Communications ACM*, 20:504--513.

Devietti, J., Blundell, C., Martin, M. M., and Zdancewic, S. (2008). Hardbound: architectural support for spatial safety of the c programming language. *ACM SIGOPS Operating Systems Review*, 42(2):103--114.

Dhurjati, D., Kowshik, S., and Adve, V. (1996). SAFECode: enforcing alias analysis for weakly typed languages. In *Conference on Programming Language Design and Implementation, (PLDI).* ACM.

Dunkels, A., Gronvall, B., and Voigt, T. (2004). Contiki - a lightweight and flexible operating system for tiny networked sensors. In *International Conference on Local Computer Networks (LCN).* IEEE.

Engoulou, R. G., Bellaïche, M., Pierre, S., and Quintero, A. (2014). Vanet security surveys. *Computer Communications*, 44:1--13.

Fan, L., Wang, Y., Cheng, X., and Jin, S. (2012). Quantitative analysis for privacy leak software with privacy petri net. In *Proceedings of the ACM SIGKDD Workshop on Intelligence and Security Informatics*, page 7. ACM.

Feautrier, P. (1996). Automatic parallelization in the polytope model. In *The Data Parallel Programming Model.* Springer.

Friedley, A., Bronevetsky, G., Hoefler, T., and Lumsdaine, A. (2013). Hybrid mpi: efficient message passing for multi-core systems. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 18. ACM.

Ghose, S., Gilgeous, L., Dudnik, P., Aggarwal, A., and Waxman, C. (2009). Architectural support for low overhead detection of memory violations. In *Design, Automation & Test in Europe (DATE).* IEEE.

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

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.

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.

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.

Hui, J., Thubert, P., et al. (2010). Compression format for ipv6 datagrams in 6lowpan networks. *draft-ietf-6lowpan-hc-13 (work in progress)*.

Intelligence, A. B. (2013). More than 30 billion devices will wirelessly connect to the internet of everything in 2020. *ABI research news*, 9.

Iordache, M. V. and Antsaklis, P. J. (2009). Petri nets and programming: A survey. In *Proceedings of the 2009 American control conference*, pages 4994--4999.

Jabeen, F. and Nawaz, S. (2015). In-network wireless sensor network query processors: State of the art, challenges and future directions. *Information Fusion*, 25:1--15.

Jovanovic, N., Kruegel, C., and Kirda, E. (2006). Pixy: A static analysis tool for detecting web application vulnerabilities. In *Security and Privacy, 2006 IEEE Symposium on*, pages 6--pp. IEEE.

Kermani, M. M., Zhang, M., Raghunathan, A., and Jha, N. K. (2013). Emerging frontiers in embedded security. In *VLSI Design and 2013 12th International Conference on Embedded Systems (VLSID), 2013 26th International Conference on*, pages 203--208. IEEE.

Kinney, S. L. (2006). *Trusted platform module basics: using TPM in embedded systems*. Newnes.

Kothmayr, T., Hu, W., Schmitt, C., Bruenig, M., and Carle, G. (2011). Poster: Securing the internet of things with DTLS. In *Conference on Embedded Networked Sensor Systems, (SenSys)*. ACM.

Lai, Z., Cheung, S.-C., and Chan, W. K. (2008). Inter-context control-flow and dataflow test adequacy criteria for nesc applications. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 94--104. ACM.

Lam, M. S., Martin, M., Livshits, B., and Whaley, J. (2008). Securing web applications with static and dynamic information flow tracking. In *Proceedings of the 2008 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 3--12. ACM.

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.

Levy, E. (1996). Smashing the stack for fun and profit. *Phrack*, 7(49).

Li, P. and Regehr, J. (2010). T-check: bug finding for sensor networks. In *International Conference on Information Processing in Sensor Networks (IPSN)*. ACM.

Li, W., Delicato, F. C., Pires, P. F., Lee, Y. C., Zomaya, A. Y., Miceli, C., and Pirmez, L. (2014). Efficient allocation of resources in multiple heterogeneous wireless sensor networks. *Journal of Parallel and Distributed Computing*, 74(1):1775--1788.

Liao, H., Wang, Y., Cho, H. K., Stanley, J., Kelly, T., Lafortune, S., Mahlke, S., and Reveliotis, S. (2013). Concurrency bugs in multithreaded software: Modeling and analysis using petri nets. *Discrete Event Dynamic Systems*, 23(2):157--195.

Liu, X., Guo, Z., Wang, X., Chen, F., Lian, X., Tang, J., Wu, M., Kaashoek, M. F., and Zhang, Z. (2008). D3S: Debugging deployed distributed systems. In Crowcroft, J. and Dahlin, M., editors, *NSDI*, pages 423--437. USENIX Association.

Macedo, D. F., de Oliveira, S., Teixeira, F. A., Aquino, A. L. L., and Rabelo, R. A. (2012). CIA2-ITS: Interconnecting mobile and ubiquitous devices for Intelligent Transportation Systems. In *IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom)*, pages 447--450. IEEE.

Marwedel, P. (2010). *Embedded system design: Embedded systems foundations of cyber-physical systems.* Springer Science & Business Media.

Mezghani, F., Dhaou, R., Nogueira, M., and Beylot, A.-L. (2014). Utility-based forwarder selection for content dissemination in vehicular networks. In *IEEE International Conference on Personal, Indoor and Mobile Radio Communications (PIMRC).*

Mouttappa, P., Maag, S., and Cavalli, A. R. (2013). Using passive testing based on symbolic execution and slicing techniques: Application to the validation of communication protocols. *Computer Networks*, 57(15):2992--3008.

Murata, T. (1989). Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541--580.

Nagaraj, K., Killian, C., and Neville, J. (2012). Structured comparative analysis of systems logs to diagnose performance problems. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 26--26. USENIX Association.

Nagarakatte, S., Martin, M. M., and Zdancewic, S. (2012). Watchdog: Hardware for safe and secure manual memory management and full memory safety. *Computer Architecture News*, 40(3):189--200.

Nagarakatte, S., Martin, M. M., and Zdancewic, S. (2014). Watchdoglite: Hardware-accelerated compiler-based pointer checking. In *International Symposium on Code Generation and Optimization (CGO)*. IEEE.

Nazaré, H., Maffra, I., Santos, W., Barbosa, L., Gonnord, L., and Quintão Pereira, F. M. (2014). Validation of memory accesses through symbolic analyses. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, pages 791--809. ACM.

Nethercote, N. and Seward, J. (2007). Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Conference on Programming language design and implementation, (PLDI)*. ACM.

Nielson, F., Nielson, H. R., and Hankin, C. (1999). *Principles of program analysis*. Springer Science & Business Media.

Oliveira, L., Scott, M., Lopez, J., and Dahab, R. (2008). Tinypbc: Pairings for authenticated identity-based non-interactive key distribution in sensor networks. In *International Conference on Networked Sensing Systems,(INSS).*, pages 173--180. IEEE.

Oliveira, S. D., Teixeira, F. A., and Macedo, D. F. (2013). Sistema de Coleta e Disseminação de Dados de Trânsito (SBRC). In *Brazilian Symposium on Computer Networks and Distributed Systems*, pages 1092--1099.

Ottenstein, K. J., Ballance, R. A., and MacCabe, A. B. (1990). The program dependence web: a representation supporting control-, data-, and demand-driven interpretation of imperative languages. In *Conference on Programming Language Design and Implementation, (PLDI)*. ACM.

Paisante, V. M., Saggioro, L. F. Z., Rodrigues, R. E., Oliveira, L. B., and Pereira, F. M. Q. (2014). Prevençao de ataques em sistemas distribuídos via análise de intervalos. In *XIV Simpósio Brasileiro em Segurança da Informação e Sistemas Computacionais (SBSeg)*.

Pascual, V. and Hascoët, L. (2012). Native handling of Message-Passing communication in Data-Flow analysis. In *Springer Recent Advances in Algorithmic Differentiation*. Springer.

Pellegrini, S. (2011). *On Simplifying and Optimizing Message Passing Programs: A Compiler and Runtime-Based Approach*. PhD thesis, University of Innsbruck.

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.

Perrig, A., Szewczyk, R., Wen, V., Culler, D., and Tygar, J. D. (2002). SPINS: Security protocols for sensor networks. *Wireless Networks*, 8(5):521--534. Also in MobiCom'01.

Pires, P. F., Cavalcante, E., Barros, T., Delicato, F. C., Batista, T., and Costa, B. (2014). A platform for integrating physical devices in the internet of things. In *Embedded and Ubiquitous Computing (EUC), 2014 12th IEEE International Conference on*, pages 234--241. IEEE.

Preissl, R., Köckerbauer, T., Schulz, M., Kranzlmüller, D., Supinski, B. R. d., and Quinlan, D. J. (2008). Detecting patterns in mpi communication traces. In *International Conference on Parallel Processing (ICPP)*. ICPP.

Raveneau, P., Chaput, E., Dhaou, R., and Beylot, A.-L. (2014a). Freak dtn: Frequency routing, encounters and keenness for dtn. In *Wireless Days (WD), 2014 IFIP*, pages 1--3. IEEE.

Raveneau, P., Dhaou, R., Chaput, E., and Beylot, A.-L. (2014b). Dtns back: Dtns broadcasting ack. In *Global Communications Conference (GLOBECOM), 2014 IEEE*, pages 2789--2794. IEEE.

Ravi, S., Raghunathan, A., Kocher, P., and Hattangady, S. (2004). Security in embedded systems: Design challenges. *ACM Transactions on Embedded Computing Systems (TECS)*, 3(3):461--491.

Rimsa, A. A., D'Amorim, M., and Pereira, F. M. Q. (2010). Efficient static checker for tainted variable attacks. In *SBLP*. SBC.

Rivera, J. and van der Meulen, R. (2013). Gartner says the internet of things installed base will grow to 26 billion units by 2020. *Stamford, conn., December*, 12.

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)*. IEEE.

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. *IEEE Latin America Transactions*, 14(April).

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 *International Conference on Information Processing in Sensor Networks (IPSN)*. ACM.

Schwartz, E. J., Avgerinos, T., and Brumley, D. (2010). All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Symposium on Security and Privacy (S&P)*. IEEE.

Serebryany, K., Bruening, D., Potapenko, A., and Vyukov, D. (2012). AddressSanitizer: a fast address sanity checker. In *Annual Technical Conference (ATA)*. USENIX.

Seshadri, A., Perrig, A., Van Doorn, L., and Khosla, P. (2004). Swatt: Software-based attestation for embedded devices. In *Security and Privacy, 2004. Proceedings. 2004 IEEE Symposium on*, pages 272--282. IEEE.

Shacham, H. (2007). The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *CCS*, pages 552--561. ACM.

Shacham, H., Page, M., Pfaff, B., Goh, E.-J., Modadugu, N., and Boneh, D. (2004). On the effectiveness of address-space randomization. In *CSS*, pages 298--307. ACM.

Shelby, Z., Hartke, K., and Bormann, C. (2014). The constrained application protocol (coap).

Shende, S. S. and Malony, A. D. (2006). The tau parallel performance system. *Int. J. High Perform. Comput. Appl.*, 20(2):287--311.

Silva, B., Cecilio, D., Souza, E. M., Teixeira, F. A., Wong, H. C., and Nazaré, H. (2013a). Segurança de Software em Sistemas Embarcados: Ataques & Defesas. In *Minicursos do XIII Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais (SBSEG)*, pages 101--155.

Silva, C., Teixeira, F. A., Oliveira, S. D., and Aquino, A. L. L. (2013b). PMCP : Uma Heurística Probabilística para Otimizar a Instalação de Pontos de Disseminação em Redes Veiculares. In *Simpósio Brasileiro de Computação Ubíqua e Pervasiva (SBCUP)*.

Silva, M. J., Teixeira, F. A., and Rabelo, R. A. (2013c). RouteSpray : Um algoritmo de roteamento de múltiplas cópias baseado em rotas de trânsito. In *Simpósio Brasileiro de Computação Ubíqua e Pervasiva (SBCUP)*.

Silva, M. J., Teixeira, F. A., and Rabelo, R. A. (2014). Combining the spray technique with routes to improve the routing process in VANETS. In *16th International Conference on Enterprise Information Systems (ICEIS)*.

Singh, D. and Kaiser, W. J. (2010). The atom LEAP platform for energy-efficient embedded computing. Technical report 88b146bk, UCLA.

Sridharan, M., Artzi, S., Pistoia, M., Guarnieri, S., Tripp, O., and Berg, R. (2011). F4F: taint analysis of framework-based web applications. In *Conference on Object-Oriented Programming (OOPSLA)*. ACM.

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

Stroustrup, B. (2007). Evolving a language in and for the real world: C++ 1991-2006. In *HOPL*, pages 1–59. ACM.

Strout, M. M., Kreaseck, B., and Hovland, P. D. (2006). Data-flow analysis for mpi programs. In *Parallel Processing, 2006. ICPP 2006. International Conference on*, pages 175--184. IEEE.

Sunyé, G., De Almeida, E. C., Le Traon, Y., Baudry, B., and Jézéquel, J.-M. (2014). Model-based testing of global properties on large-scale distributed systems. *Information and Software Technology*, 56(7):749--762.

Teixeira, F. A., e Silva, V. F., Leoni, J. L., Macedo, D. F., and Nogueira, J. M. (2014a). Vehicular networks using the ieee 802.11 p standard: an experimental analysis. *Vehicular Communications*, 1(2):91--96.

Teixeira, F. A., e Silva, V. F., Leoni, J. L., Macedo, D. F., and Nogueira, J. M. S. (2014b). Vehicular networks using the IEEE 802.11p standard: An experimental analysis. *Vehicular Communications*, 1:91--96. ISSN 22142096.

Teixeira, F. A., e Silva, V. F., Leoni, J. L., Santos, G. C. E., and Macedo, D. F. (2013). Análise Experimental de Redes Veiculares Utilizando o ao IEEE 802 . 11p. In *Simpósio Brasileiro de Computação Ubíqua e Pervasiva (SBCUP)*.

Teixeira, F. A., Machado, G. V., Fonseca, P. M., Pereira, F. M. Q., Wong, H. C., Nogueira, J. M. S., and Oliveira, L. B. (2014c). Defending Code from the Internet of Things against Buffer Overflow. In *Brazilian Symposium on Computer Networks and Distributed Systems (SBRC)*, pages 293--301.

Teixeira, F. A., Machado, G. V., Fonseca, P. M., Pereira, F. M. Q., Wong, H. C., Nogueira, J. M. S., and Oliveira, L. B. (2015a). Defending Internet of Things against Exploits. *IEEE Latin America Transactions*, 14(April).

Teixeira, F. A., Machado, G. V., Pereira, F. M., Wong, H. C., Nogueira, J., and Oliveira, L. B. (2015b). 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.

Tripp, O., Pistoia, M., Fink, S. J., Sridharan, M., and Weisman, O. (2009). TAJ: Effective taint analysis of web applications. In *Conference on Programming Language Design and Implementation (PLDI)*. ACM.

Tsiftes, N. and Dunkels, A. (2011). A database in every sensor. In *Proceedings of the 9th ACM Conference on Embedded Networked Sensor Systems*, pages 316--332. ACM.

Vasseur, J., Agarwal, N., Hui, J., Shelby, Z., Bertrand, P., and Chauvenet, C. (2011). Rpl: The ip routing protocol designed for low power and lossy networks. *Internet Protocol for Smart Objects (IPSO) Alliance*, 36.

Voron, J.-B. and Kordon, F. (2008). Transforming sources to petri nets: A way to analyze execution of parallel programs. In *Proceedings of the 1st international conference on Simulation tools and techniques for communications, networks and systems & workshops*, page 13. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

Wassermann, G. and Su, Z. (2007). Sound and precise analysis of web applications for injection vulnerabilities. In *PLDI*, pages 32--41. ACM.

Winter, T. (2012). Rpl: Ipv6 routing protocol for low-power and lossy networks.

Wu, X. and Mueller, F. (2011). Scalaextrap: Trace-based communication extrapolation for spmd programs. In *Symposium on Principles and Practice of Parallel Programming (PPoPP)*. ACM.

Yabandeh, M., Knezevic, N., Kostic, D., and Kuncak, V. (2010). Predicting and preventing inconsistencies in deployed distributed systems. *ACM Trans. Comput. Syst*, 28(1).

Zanella, A., Bui, N., Castellani, A., Vangelista, L., and Zorzi, M. (2014). Internet of things for smart cities. *Internet of Things Journal, IEEE*, 1(1):22--32.

Attachment A

History and publications

This section describes the publications produced during this doctorate. This doctorate can be divided into two phases: (i) Research in Vehicular Networks with heterogeneous wireless access (VANETs); and (ii) Security Code of Distributed Systems.

The first phase was carried out in parallel with the disciplines, stage in teaching and qualifying examinations. Part of the second phase has been performed in the context of Energy-Efficient Instrumentation to Secure Systems-on-a-Chip Devices (eCoSoC)[1] project in partnership with Intel.

From December 2012 the doctorate student started to work on the eCoSoC project and the focus of research became the security code of embedded systems. This is an interdisciplinary project involving security mechanisms, compilers and computer networks.

In this phase the problem was defined, the first solution was proposed and initial experiments have been performed to demonstrate that our approach had advantages over state of the art approaches in terms of false positives and energy savings.

The problem and first idea of solution was published as part of the work [Silva et al., 2013a]. Then, a first version of the SIoT with preliminary experiments was published in SBRC 2014 [Teixeira et al., 2014c].

An extension of this work was published as an article at Latin American Transactions [Teixeira et al., 2015a]. On November 2014, we present our solution in ISRA workshop – Intel Strategic Research Alliance – sponsored by Intel in SBSeg 2014. Recently, the work was published at the IPSN[2] [Teixeira et al., 2015b]. The list of these publications can be seen below:

- Silva, B., Cecilio, D., Souza, E. M., Teixeira, F. A., Wong, H. C., and Nazaré, H..

[1]eCoSoC project – `http://www.ecosoc.dcc.ufmg.br/`

[2]CAPES – Qualis A1, H-index = 88

Segurança de Software em Sistemas Embarcados: Ataques & Defesas. In Minicursos do XIII Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais. SBSEG'2013.

- Teixeira, F. A., Machado, G. V., Fonseca, P. M., Pereira, F. M. Q., Wong, H. C., Nogueira, J. M. S., and Oliveira, L. B. . **Defending Code from the Internet of Things against Buffer Overflow**. In Brazilian Symposium on Computer Networks and Distributed Sys- tems. SBRC'2014.

- Teixeira, F. A., Machado, G. V., Fonseca, P. M., Pereira, F. M. Q., Wong, H. C., Nogueira, J. M. S., and Oliveira, L. B. . **Defending Internet of Things against Exploits**. IEEE Latin America Transactions, 2015.

- Teixeira, F. A., Machado, G. V., Fonseca, P. M., Pereira, F. M. Q., Wong, H. C., Nogueira, J. M. S., and Oliveira, L. B. . **SIoT: Securing the Internet of Things through Distributed System Analysis**. In 14th International Conference on Information Processing in Sensor Networks. IPSN'2015.

From our work was derived two undergraduate projects [3] and one master degrees[4]. The undergraduate projects have finished in 2014. The Saggioro's master project uses our framework to create a Distributed Range Analyis, and it was conclude in the first semester of 2015. Part of Saggioro's work was published at SBSeg 2014 [Paisante et al., 2014] and at Latin American Transactions [Saggioro et al., 2015].

Before december 2012, the goal of the doctorate project was to propose protocols and algorithms for the dissemination and data collection in Vehicular Ad Hoc Network (VANET). In this phase, we have reviewed works about VANET and IoT in order to propose new strategies to disseminate the information in this kind of network.

For instance, we have started to study the best way to interconnect a vehicle to another vehicle or to distribution point in an urban center with the technologies that are being made available in IoT.

In this subject, the doctorade student has obtained four national publications [Oliveira et al., 2013; Silva et al., 2013b,c; Teixeira et al., 2013] and three international publications [Teixeira et al., 2014b; Macedo et al., 2012; Silva et al., 2014]. These publications are listed below:

- Macedo, D. F., de Oliveira, S., Teixeira, F. a., Aquino, A. L. L., and Rabelo, R. A. **$(CIA)^2$ –ITS: Interconnecting mobile and ubiquitous devices for**

[3]Gustavo Vieira Machado (Automation and Control Engineering – UFMG) and Pablo Marcondes Mendes (Computer Science – UFMG)

[4]Luis Felipe Zafra Saggioro PPGCC – UFMG

Intelligent Transportation Systems. In IEEE International Conference on Pervasive Computing and Communications Workshops. PERCOM'2012.

- Oliveira, S. D., Teixeira, F. A., and Macedo, D. F. . **Sistema de Coleta e Disseminação de Dados de Trânsito**. In Brazilian Symposium on Computer Networks and Distributed Systems. SBRC'2013.
- Silva, C., Teixeira, F. A., Oliveira, S. D., and Aquino, A. L. L. . **PMCP : Uma Heurística Probabilística para Otimizar a Instalação de Pontos de Disseminação em Redes Veiculares**. In Simpósio Brasileiro de Computação Ubíqua e Pervasiva. SBCUP'2013.
- Silva, M. J., Teixeira, F. A., and Rabelo, R. A. **RouteSpray : Um algoritmo de roteamento de múltiplas cópias baseado em rotas de trânsito**. In Simpósio Brasileiro de Computação Ubíqua e Pervasiva. SBCUP'2013.
- Silva, M. J., Teixeira, F. A., and Rabelo, R. A.. **Combining the spray technique with routes to improve the routing process in VANETS**. In 16th International Conference on Enterprise Information Systems. ICEIS'2014.
- Teixeira, F. A., e Silva, V. F., Leoni, J. L., Macedo, D. F., and Nogueira, J. M. S.. **Vehicular networks using the IEEE 802.11p standard: An experimental analysis**. Vehicular Communications, 2014. ISSN 22142096.

The doctorade student spent three months in France as a Visitor PhD Student (sandwich doctorate), between December 2014 and March 2015. In this period, the doctorade student interacted with local research group, studied local works [Raveneau et al., 2014a; Mezghani et al., 2014; Raveneau et al., 2014b], and analyzed the [Raveneau et al., 2014a] protocol with SIoT.