SIoT: Securing the Internet of Things through Distributed System Analysis

Fernando A. Teixeira
UFMG, Brazil
teixeira@dcc.ufmg.br

Gustavo V. Machado
UFMG, Brazil
gvieira@dcc.ufmg.br

Fernando M. Q. Pereira
UFMG, Brazil
fernando@dcc.ufmg.br

Hao Chi Wong
Intel Corporation, CA
hao-chi.wong@intel.com

José M. S. Nogueira
UFMG, Brazil
jmarcos@dcc.ufmg.br

Leonardo B. Oliveira
UFMG, Brazil
leob@dcc.ufmg.br

ABSTRACT

The Internet of Things (IoT) is increasingly more relevant. This growing importance calls for tools able to provide users with correct, reliable and secure systems. In this paper, we claim that traditional approaches to analyze distributed systems are not expressive enough to address this challenge. As a solution to this problem, we present SIoT, 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, we can crosscheck information inferred from different nodes. This crosschecking increases 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 efficiently. Such links lets us build a holistic view of the entire network, a knowledge that we can thus forward to a traditional tool. We prove that our algorithm always terminates and that it correctly models the semantics of a distributed system. To validate our solution, we have implemented SIoT on top of the LLVM compiler, and have used one instance of it to secure 6 ContikiOS applications against buffer overflow attacks. This instance of SIoT produces code that is as safe as code secured by more traditional analyses; however, our binaries are on average 18% more energy-efficient.

Categories and Subject Descriptors

C.2.4 [**Computer Communication Networks**]: Distributed Systems—*Distributed applications*; D.4.6 [**Software**]: Operating Systems—*Security and Protection*; D.4.6 [**Software**]: Software Engineering—*Testing and Debugging*

General Terms

Security

Keywords

Internet of Things, Software Security, Buffer Overflow, Distributed System Analysis.

1. INTRODUCTION

The Internet of Things (IoT) may be thought of as the epithet of Ubiquitous Computing. In IoT, the user environment is replete of devices working collaboratively [2,3]. These devices range from computing elements such as RFID tags and biochip transponders on farm animals to smartphones and automobiles with built-in sensors. Equipments like these are known by the name of "things" [2, 3].

Ensuring the security of such systems is a problem of increasing relevance [17]. In fact, DARPA has elected security the central point in its Cyber Grand Challenges call[1]. Two main factors, however, make IoT 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.

IoT faces a plethora of security problems [17]. It suffers from the same security issues as traditional Internet-based and/or wireless systems, including jamming, spoofing, replay, and eavesdropping [4,17]. In addition, it is more prone (as compared to traditional systems) to other issues such as out-of-bound memory accesses. IoT's increased vulnerability to out-of-bound memory accesses is due to a few factors. Notably, IoT 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 IoT are commonly developed using lightweight languages such as C. This strategy turns out to be a double-edged sword. 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.

C is an inherently unsafe language [8]. 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 [10]. 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-

[1]http://cgc.darpa.mil
[2]http://slashdot.org/topic/datacenter/100k-thingbot-net-shows-risk-of-smart-devices/
[3]http://en.wikipedia.org/wiki/Morris_worm
[4]http://en.wikipedia.org/wiki/Heartbleed

```
#define BUFSIZE 512

int main(int argc,
         char **argv) {
  int buffer[BUFSIZE];
  int a;
  int  i,j;
  ...
  for(i;i<j;i++){
    ...
    buffer[i] = a;
    ...
  }
  ...
}
```

```
#define BUFSIZE 512

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

Figure 1: Vulnerable (left-hand-side) and ABC protected (right-hand-side) C code.

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. SAFECode [13] and AddressSanitizer [35]). Existing proposals resort to Array-Bound Checks (ABCs), which are tests done at runtime to ensure that a particular array access is safe – Fig. 1. 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 IoT. 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 IoT.

The goal of this paper is to describe a general framework for analysis and optimization of distributed systems, which we can use to implement an efficient solution to counter buffer-overflow attacks in IoTs. We call our framework SIoT, short for *Secure Internet of Things*. Our **key insight** is to look at a distributed system as a single entity, rather than as multiple separate message-exchanging programs. Using a novel algorithm, we can infer the communication links between different programs that converse through a network. This knowledge lets us model how data flows across distributed programs; hence, it gives us a holistic view of the entire system. Such a view can be coupled with traditional static analysis tools to improve their precision.

To validate our claims, we have used our framework to protect IoT systems against buffer overflow attacks. More specifically, we applied tainted flow analysis [5] on the model we proposed, and sanitized C programs against out-of-bound memory accesses. Tainted flow analysis 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 holistic view of the entire system, it 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. Notice that this framework is general enough to support a wider range of analysis. The solution for out-of-bound memory accesses presented in this paper is just an instance of it.

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

1. We propose an extension to the standard Control Flow Graphs (CFGs) [1], 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.

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

1. We have implemented our algorithm and its companion distributed tainted flow analysis in the LLVM compiler [20]. (Our implementation is publicly available[5].)

2. We have applied this analysis on 6 applications present in ContikiOS [14], and the results show that our proposal is 18% more energy-efficient than existing solutions.

Organization. The remainder of this paper is structured as follows. Section 2 describes programming language concepts, security definitions, and the attack model used in this paper. Section 3 presents our solution. Section 4 describes the implementation of SIoT. Section 5 evaluates SIoT in terms of energy-efficiency. In Section 6 we discuss related work. We conclude in Section 7.

2. BACKGROUND AND ASSUMPTIONS

In this section, we describe the fundamental concepts of system security and the attack model used in this paper. We start (Section 2.1) with a brief description of language-base techniques to deal with BOFs. We then (Section 2.2) describe two data structures used to code analyses – CFG and Dependence Graph (DG). Finally, we present (Section 2.3) the assumptions and the attack model used in this paper.

2.1 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 [8], dynamic analysis [35], or a combination of both.

In static analysis [8] (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 runtime overhead. Its downside is that the analysis is not able to use information that is only available at runtime, which can determine more 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 [35], 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

[5]https://code.google.com/p/ecosoc

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.

2.2 Code Analysis Using CFG and DG

The Control Flow Graph (CFG) [1] is 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 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 2.3 for a discussion of these assumptions), then we would see the assignments in lines 1 and 5 (Fig. 2) differently. 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. 2a), which could provide malicious data from attackers (`msg` has been used in buffer access). The first assignment in server program (line 1, Fig. 2b) is not vulnerable because the data being assigned is a hard-coded constant from the client program (line 1, Fig. 2a).

In their standard form, CFGs are 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. 2b) and one false-positive (line 1, Fig. 2b). We present a proposal that addresses this shortcoming in Section 3.

The Dependence Graph (DG) [27] is a data structure frequently used with the CFG. While CFGs model the control flow of a program, DGs focus on the data flow, i.e., dependences between instructions and data. Given a program in a format known as Static Single Assignment form [11] (where each variable has a single definition site), a DG has a node 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. Just like standard CFGs, standard DGs are unable to model the data flow in programs that span multiple distributed processes. This is another issue that this paper solves.

2.3 System Assumptions & Attack Model

For the purpose of this work, we assume networks of distributed and embedded nodes, organized as IoT systems. Each node can interact with its environment through sensors, actuators, or user interfaces. 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 Plataform Module (TPM) [18] can be employed to ensure the integrity of nodes systems and cryptographic solutions like [19, 26, 30] can be used to establish a secure communication channel.

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

Though limited in the type of attacks they can launch, such attackers can potentially 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.

3. SIoT

In this section, we present our proposal on analysis of distributed programs, and then show how this framework can be used to mitigate buffer overflow attacks against IoT systems implemented in C. We start (Section 3.1) by introducing the concepts of DCFG and DDG, two data structures that would enable us to model control flow and data flow across different programs in a distributed system. We then (Section 3.2) formalize the level assignment algorithm briefly described in Section 3.1, and prove that the algorithm terminates, and is correct.

3.1 DCFG and DDG

CFGs model the control flow of individual programs. To analyze an entire distributed system, we need to work with control flow graphs that transcend program boundaries. We propose the notion of DCFG for this purpose, and describe how they are built below. Each DCFG models the communication between two programs in a system. If the system contains more than two programs, a DCFG is necessary to model the relationship between each pair of them. 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 a subgraph. 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 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, 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)

```
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);
}
```

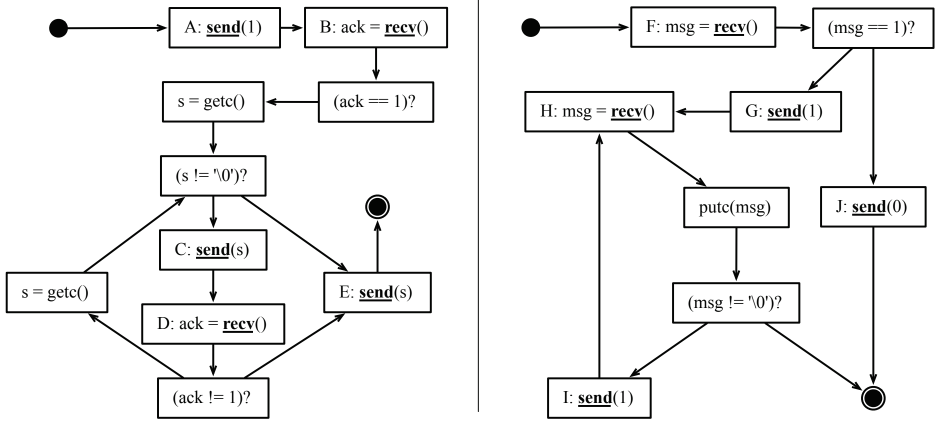


Figure 2: Echo application's programs and their respective CFGs. (a) Echo client. (b) Echo server.

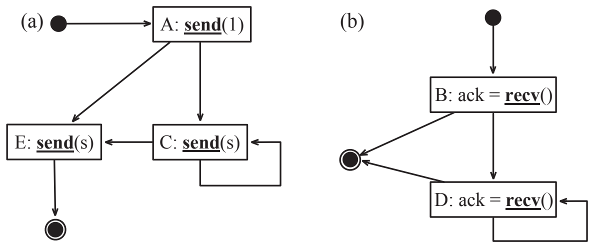


Figure 3: (a) Send-Graph and (b) Receive-Graph for echo client (Fig. 2a).

a DCFG that better model the workings of a system, we introduce the notions of Send-Graph, Receive-Graph, and levels.

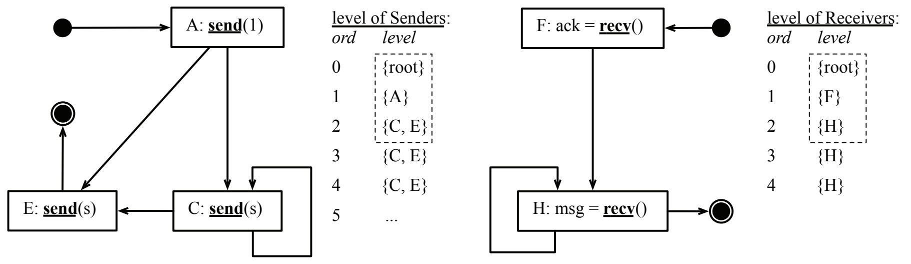
Given a CFG $\mathcal{C}$ of a program, we define its associated Send-Graph $\mathcal{S}$ and Receive-Graph $\mathcal{R}$ as follows. For each vertex $v \in \mathcal{C}$ labeled with a send operation, we add a vertex v' to $\mathcal{S}$. We also add $start'$ and $exit'$ vertices, which correspond to 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 shows the $\mathcal{S}$ and $\mathcal{R}$ derived from the CFG in Fig. 2a.

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 Fig. 4.

Consider echo client program Send-Graph (Fig. 4 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, 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. 5 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 describe above are captured in the *Elevator Algorithm* (Algorithm 1).

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 the similar way previous described. Thus, the final graph con-



level of Senders:

ord	*level*
0	{root}
1	{A}
2	{C, E}
3	{C, E}
4	{C, E}
5	...

level of Receivers:

ord	*level*
0	{root}
1	{F}
2	{H}
3	{H}
4	{H}

Figure 4: 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. 6.

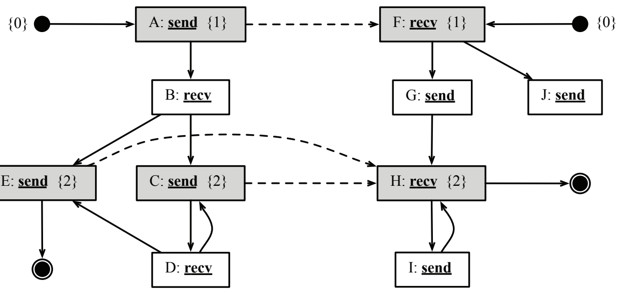


Figure 5: Links between echo client's SENDs and echo sever's RECVs of our running example. Numbers next to vertices denote their respective levels.

tains the dependences of all the programs of the distributed system as if it were a single program system. The DDG can be used for different security analysis like the detection of buffer overflow or integer overflow [32] vulnerabilities. For instance, in Section 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.

3.2 Formalization of Elevator Algorithm

Equation 1 defines the levels of either a Send-Graph or Receive-Graph graph, named here as Message-Graph mg as follows:

$$\begin{array}{lcl} level(mg, 0) & = & \{start\} \\ level(mg, n) & = & \{v \mid \overrightarrow{uv} \in mg \wedge u \in level(mg, n-1)\} \end{array} \quad (1)$$

Equation 1 gives us a way to generate levels, which we formalize in Fig. 6. 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 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 of producing levels to messages graphs. According to this rule, we stop generating levels as soon as we produce a set of nodes that we had generated before. As we prove later, in Lemma 3.6, this algorithm always terminates.

If $solution(L)$ is true, we number the levels of a program ac-

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: DCFG – $\mathcal{D}$
▷ Set the SEND levels
foreach $\mathcal{S}_i$ do
    $n \leftarrow 0$
    $\mathcal{S}_i L_n \leftarrow \{root\}$
    while $\mathcal{S}_i L_n \neq \emptyset$ and not repeat a level set do
        foreach node $s$ in $\mathcal{S}_i L_n$ do
            $S_{succs} \leftarrow$ successors of s
            $\mathcal{S}_i L_{n+1} \leftarrow \mathcal{S}_i L_{n+1} \cup S_{succs}$
        $n \leftarrow n + 1$
▷ Set the RECV levels likewise
▷ Build the DCFG
$\mathcal{D} \leftarrow \mathcal{C}_1 \cup \mathcal{C}_2$
for $k \leftarrow 1$ in $n$ do
    foreach SEND $\in \mathcal{S}_1 L_k$ and RECV $\in \mathcal{R}_2 L_k$ do
        add an edge from SEND to RECV in $\mathcal{D}$
    foreach RECV $\in \mathcal{R}_1 L_k$ and SEND $\in \mathcal{S}_2 L_k$ do
        add an edge from SEND to RECV in $\mathcal{D}$
```

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

[LVR] $$level([start])$$

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

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

Figure 6: 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).

cording to the rules below:

[ZER] $$ord([\,], 0)$$

[NUM] $$\frac{ord(L, N)}{ord([L_1 \mid L], N+1)}$$

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 1. Fig. 4 (a) shows the sets of levels for the Send-Graph in the echo client of our running example. Fig. 4 (b) shows the levels for the Receive-Graph of the echo server.

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. 7 shows the semantics of a simple language [25], 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.

Essential Properties.

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

DEFINITION 3.1. *Given a Send-Graph (or a Receive-Graph) mg, a node v has distance d if there exists a path from* start *to v passing through $d-1$ nodes. The same node might have several distances.*

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 node that corresponds to send operation i in P, we shall refer to them as the pair i/s_i. The distance of s_i determines how many messages have been sent by P once i is evaluated. We state this property in Lemma 3.2. There exists an analogous result for receivers, which we omit, for the sake of brevity.

LEMMA 3.2. *Let mg_s be the Send-Graph of program P, such that i/s_i is a pair. If s_i has distance 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: then $v_s = start$, which corresponds to the point before any instruction of P. Because there is no send before the first instruction of P, the lemma is vacuously true.
Induction hypothesis : the lemma is true until distance d.
Induction step : If a node s has distance $d+1$, then it is preceded by a node s' of distance d, by the definition of distance. 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 distances. Lemma 3.3 makes this relationship explicit. According to this lemma, if two nodes belong into the same level, then there exist paths of same length linking these nodes back to the start vertex. In other words, the idea of levels group vertices that share common distances. Notice that the same node can have different distances. As an example, node C in Fig. 3 (a) has infinite different distances, as it is part of a loop. Thus, it is possible to have the same node in different levels.

LEMMA 3.3. *If $v \in level(mg, n)$, then there is a path of distance n from v to* start.

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

The concept of distance is key to prove the correctness of our method to build distributed graphs, because only nodes with the same distance can communicate, as we state in Lemma 3.4.

LEMMA 3.4. *Given a Send-Graph mg_s, a Receive-Graph mg_r, node $s \in mg_s$ and node $r \in mg_r$, 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 distance.*

Proof: From Lemma 3.2, we know that the distance 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 distance, 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 distance 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, we have that two nodes can communicate only if they belong into the same level. We state formally this property in Theorem 3.5.

THEOREM 3.5. *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 3.3, we know that two nodes at the same level have a common distance, and by Lemma 3.4 we know that they can communicate.
Necessity: still by Lemma 3.4, if two nodes do not share a common distance, then they cannot communicate. $\square$

The Computation of Levels Terminates.

Equation 1 gives us an algorithmic way to compute the set of levels of messages graphs. We can construct a level n from the

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

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

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

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

Figure 7: The semantics of our message passing language.

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 3.6.

LEMMA 3.6. *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 1, $level(k+1)$ is totally defined by $level(k)$. □

Lemma 3.6 gives us an interactive way to build levels for a Send-Graph (or a Receive-Graph) graph: it is enough to solve Equation 1 successively, until we find two levels that are the same. This process is guaranteed to terminate, as we prove in Theorem 3.7.

THEOREM 3.7. *The iterative construction of levels terminate.*

Proof: The number of levels is finite, because a graph with N nodes might have at most 2^N different levels. Thus, successive applications of Equation 1 will eventually produce two levels that are the same. From Lemma 3.6, we know that we have found a cycle, and no new level will be discovered. □

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. 6 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.

4. IMPLEMENTATION DETAILS

SIoT is a template for deriving static analyses tools, which we have implemented on top of the LLVM compiler [20]. A tool derived from SIoT is composed of two parts: one application independent, and other application dependent. Fig. 8 shows this architecture. The part of SIoT that is application independent is the same for any static analysis. We call it the SIoT *Core*. The second part contains libraries that a user of SIoT must create to implement a specific static analysis – we call it the SIoT *instance*. The bridge between these two worlds, core and instance, is the Distributed Dependence Graph – DDG. SIoT always builds the DDG for a program in the same way, independent on how this structure will be used later. Each instance process this graph in a particular way.

The Architecture of the SIoT Core.

SIoT process files written in the LLVM IR. 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, bitvectors and arrays. From a set of files, in this format, SIoT builds the DDG, through a process that we further divide into two phases: *merging* and *linking*. Merging joints several bytecode files into a single file. During this phase we must resolve name conflicts, i.e., different files may define the same names. We solve such conflicts through renaming. In the linking phase we extract 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.

Merging different bytecode files only requires the names of SEND and RECV functions. 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. Based on the network functions information, SIoT adds special tags to the LLVM bytecode files, identifying SEND and RECV. The annotated bytecodes are then merged into a single unit to facilitate subsequent analyses.

During the linking stage, *Elevator* assigns levels to functions marked as SEND or RECV. This step is described in Section 3.1. After determining levels, SIoT creates the DCFG (Distributed Control Flow Graph). From the DCFG, a third pass, *DistDepGraph*, builds the DDG. This data structure is then passed to possible instances of SIoT.

Case Study: Buffer Overflow Vulnerabilities.

As a case study, we analyze the dependences between inputs and memory operations in order to detect potential vulnerabilities to buffer overflow attacks. We say that an array is vulnerable if it can be indexed through data that is a function of some untrusted input. An untrusted input is any function that may interact with the user or the environment, with the file system, sensors or with

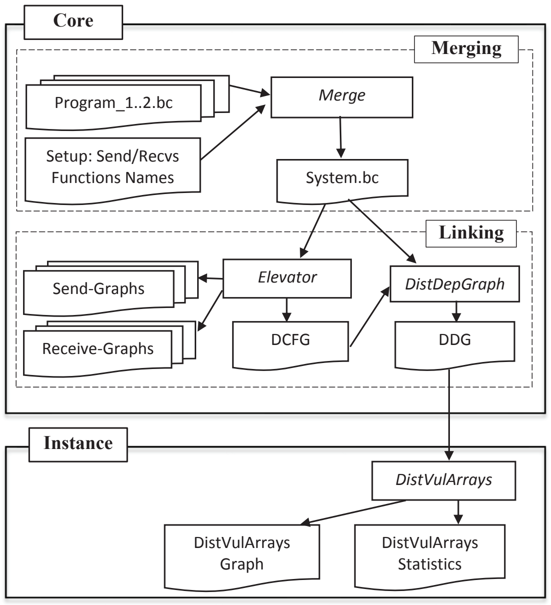


Figure 8: SIoT Architecture.

the serial port. A malicious user that controls an untrusted input can try to force out-of-bounds memory accesses. The DDG lets us track the flow of information throughout the program, giving us a way to point out vulnerabilities. To detect program vulnerabilities, we feed the DDG and the input list to *DistVulArrays*, the LLVM pass that implements our analysis. This pass searches for paths, within the DDG, between untrusted inputs and memory access operations. After analyzing a program, *DistVulArrays* produces the following outputs: (i) the number of true-positives, and potential false-positives (if we had analyzed each program of a system independently); and (ii) the graph of vulnerable paths in the program.

5. EVALUATION

Array-Bound Checks – ABCs [8] – are an effective technique to prevent attacks that exploit buffer overflows. However, ABCs have a cost, in terms of code size, runtime and energy consumption. We have used the instance of SIoT described in Section 4 to eliminate these checks from applications taken from the ContikiOS operating system [14]. In the present section we describe these results. Section 5.1 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 compares the number of ABCs eliminated by SIoT versus other approaches. Section 5.3 discusses the energy budget of the code that SIoT helps us to produce. Finally, Section 5.4 deals with the cost of the static analysis that we have implemented in SIoT and the complexity of the Elevator algorithm.

5.1 Case Study: udp-ipv6

Udp-ipv6[6] is a public example of application that runs in ContikiOS. It implements a UDP 6LoWPAN[7] client and server. In the UDP Server, (Fig. 9 – 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. 9 – 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 are vulnerable. This observation stems from the fact that messages prepared by the client have no dependency from any external data source (Fig. 9 – right-hand-side, lines 10-14). Because such messages are not tainted at their origin, according to our attack model (Section 2.3), 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. 9 – left-hand-side, lines 8-11). Thus, it 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 On the Number of ABCs that we insert

The holistic 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 more efficient executable code. To validate this claim, we have used SIoT to improve the code that AddressSanitizer (ASan) [35] 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 separate programs, and not as a single entity. Henceforth, we refer to this technique as Baseline.

We perform the experiments into 6 pairs of ContikiOS applications (Table 1). Each pair consists on 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. Table 1 compares performance of ASan, TFA and SIoT using the number of ABCs necessaries in each approach. This table 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, 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 6. 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.

5.3 On Energy Saving

Each ABC that we eliminate represents a small saving in terms of energy consumption. To back up this observation with actual data, we performed an experiment with 6 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 applica-

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

```
static void tcpip_handler(void) {
  static int seq_id;
  char buf[MAX_PAYLOAD_LEN];
  if(uip_newdata()) {
    ((char *)uip_appdata)[uip_datalen()] = 0;
    PRINTF("Server received: '%s' from ",
         (char *)uip_appdata);
    ...
    uip_ipaddr_copy(&server_conn->ripaddr,
          &UIP_IP_BUF->srcipaddr);
    PRINTF("Responding with message: ");
    sprintf(buf, "Hello from the server!  (%d)",
         ++seq_id);
    PRINTF("%s\n", buf);
    uip_udp_packet_send(server_conn, buf,
         strlen(buf));
    ...
  }
}
```

```
static void tcpip_handler(void) {
  char *str;
  if(uip_newdata()) {
    str = uip_appdata;
    str[uip_datalen()] = '\0';
    printf("Response from the server: '%s'\n",
         hilightstr);
  }
}
static void timeout_handler(void) {
  static int seq_id;
  printf("Client sending to: ");
  PRINT6ADDR(&client_conn->ripaddr);
  sprintf(buf, "Hello %d from the client", ++seq_id);
  printf(" (msg: %s)\n", buf);
  uip_udp_packet_send(client_conn, buf,
       UIP_APPDATA_SIZE);
  ...
}
```

Figure 9: ContikiOS udp-ipv6 server (left-hand-side) and client (right-hand-side) code snippets.

Table 1: ABCs inserted by ASan, Baseline, and SIoT as function of applications characteristics.

Applications	Arrays	Memory Accesses	ABCs inserted			% ABCs Reduction	
			ASan	Baseline	SIoT	SIoT vs ASan	SIoT vs Baseline
netdb client/server	6,181	22,819	4,641	172	16	99.66%	90.70%
ping6 / new-ipv6	4,683	16,871	7,453	166	14	99.81%	91.57%
ipv6-rpl-collect udp-sender / sink	4,786	17,301	3,831	168	14	99.63%	91.67%
ipv6-rpl-udp client/server	4,760	17,162	3,787	170	14	99.63%	91.76%
udp-ipv6 client/server	4,701	16,945	3,736	212	14	99.63%	93.40%
coap-client / rest-server	5,195	18,693	4,032	214	14	99.65%	93.46%

tion: (i) without ABCs; (ii) with the ABCs inserted by the Baseline; and (iii) with the ABCs inserted by SIoT. To carry on this experiment, we have installed the applications in IRIS XM2110 sensors[8] 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* [36]. 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[9] (NI USB-6009) power meter to measure the instantaneous current between the load and the ground ports (Fig. 10). We then send the signal to a software that runs on a separate PC. Since the tension in the embedded system is constant, this software is able to calculate the instantaneous power and, by integrating it, the total amount of energy that the program consumes.

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

Our results (Table 2 and Fig. 11) show that SIoT outperforms Baseline for all applications. The SIoT's savings range from 2.14% (*ping*) to 31.58% (*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.

[8]http://www.memsic.com/wireless-sensor-networks/

[9]http://en.wikipedia.org/wiki/Data_acquisition

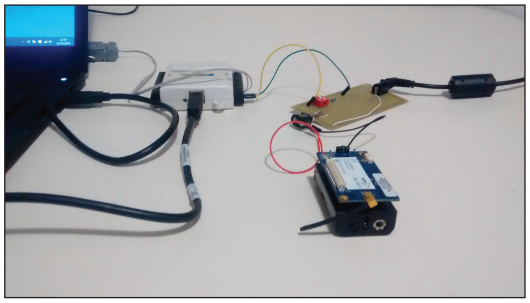


Figure 10: Energy Measurement Setup.

5.4 On Asymptotic Complexity

To estimate the asymptotic complexity of the algorithm that we propose in Section 3, 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, report-

Table 2: 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

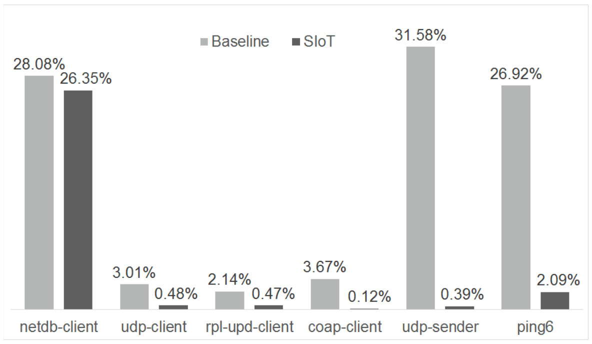


Figure 11: SIoT vs Baseline – energy overhead.

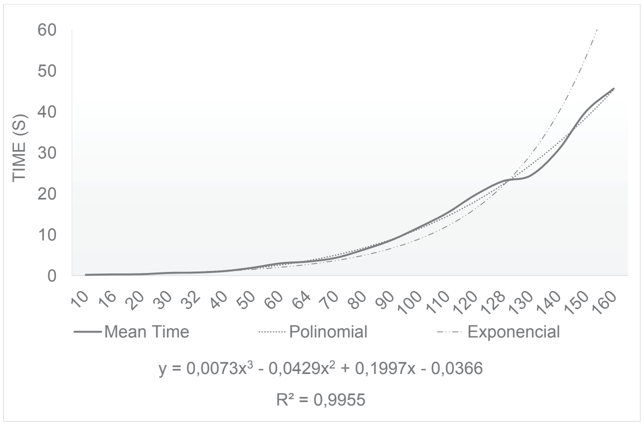


Figure 12: Elevator – Runtime as a function of program size.

ing the average of these samples. From these points, i.e., size versus runtime, we produced the chart in (Fig. 12). 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 3.2), 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 paper are feasible in practice. Table 3 shows that SIoT's static analysis took, on average, 66 seconds at compile time [10] and used 170 MB[11] when applied onto our benchmarks. Once again, note that these time and memory are used in compile time and in turn they do not degrade the code performance once deployed. 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.

6. RELATED WORK

Code Analyses of Sensor Networks. The instance of SIoT described in Section 4 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. [10,16,35]), hardware-based (e.g. [12,22]), or hybrid (e.g. [23]). 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.

This paper uses distributed control flow graph as a way to improve the precision of taint analysis that tracks buffer overflow vulnerabilities. Taint analysis is not new. For an overview of the field, see Schwartz *et al.* [34]. There are many tools and frameworks to perform taint analyses in actual programs, including distributed systems such as web applications [38]. 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 [38] and F4F [37].

Cooprider *et al.* propose a memory safety solution for sensor nodes named Safe TinyOS [9]. It handles array and pointer errors before they can corrupt the RAM. 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 limited networks that analyze codes of distributed systems based on test generation. Among these, we highlight Kleenet [33] and T-Check [21]. 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. But 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.

[10]We conducted experiments on a laptop Intel Core i7 2.2GHz.

[11]Memory measurements have been obtained through Valgrind [24].

Table 3: Time and memory spent in SIoT's static analysis as a function of application characteristics.

Application	Instructions	DDG Nodes	DDG Edges	Time (s)	Memory (MB)
netdb client/server	57.877	95,656	139,763	66.24	210.03
ping / new-ipv6	47,422	75,899	113,900	63.58	167.36
ipv6-rpl-collect udp-sender/sink	48,800	78,365	117,416	80.08	173.37
ipv6-rpl-udp client/server	48,226	77,128	115,770	66.31	169.90
udp-ipv6 client/server	48,800	78,365	117,416	80.08	167.39
coap-client / rest-server	51,258	82,647	123,879	54.36	179.68

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 [28] 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 efficient and precise – as long as the user correctly understand the application – has the main disadvantage of burdening programmers with a task that, in our understanding, should be solved by the compiler. In the rest of this section we describe only automatic approaches to tackle with the inference of communication links problem.

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 [6]. 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 [29], has expanded Bronevestsky's ideas to deal with features of MPI programs that the latter could not handle. He relies on the polyhedron model [15] 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.

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 [7, 31, 39]. We call such approaches *dynamic post-mortem analyses*.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.

7. CONCLUSION

This paper has presented SIoT, a framework to Secure the internet of Things. SIoT provides typical static analyses with a holistic view of a distributed system. This view improves the precision of such analyses. To validate this claim, we have used SIoT 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 effective and useful to make programs running over a network safer.

As future work, we plan to use SIoT to enable other kinds of program analyses. In particular, we are interested in using it to secure programs against errors caused by integer overflows. We also want to use SIoT to enable compiler optimizations. As an example, if we go back to Figure 2, we see that the conditional test at line 2 of our server is unnecessary. Such an observation requires SIoT's global view of a distributed system.

Acknowledgment: We thank the IPSN referees for suggestions and comments. This paper is supported by the Intel Corporation, CNPq and FAPEMIG.

Software: The SIoT implementation is publicly available at https://code.google.com/p/ecosoc.

8. ADDITIONAL AUTHORS

Pablo M. Fonseca, UFMG, Brazil – pablom@dcc.ufmg.br.

9. REFERENCES

[1] F. E. Allen. Control flow analysis. *ACM Sigplan Notices*, 5:1–19, 1970.

[2] K. Ashton. That 'Internet of Things' Thing. *RFiD Journal*, 22:97–114, 2009.

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

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

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

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

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

[8] B. Chess and J. West. *Secure Programming with Static Analysis*. Addison-Wesley Professional, first edition, 2007.

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

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

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

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

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

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

[15] P. Feautrier. Automatic parallelization in the polytope model. In *The Data Parallel Programming Model*. Springer, 1996.

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

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

[18] S. L. Kinney. *Trusted platform module basics: using TPM in embedded systems*. Newnes, 2006.

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

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

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

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

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

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

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

[26] L. Oliveira, M. Scott, J. Lopez, and R. Dahab. 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, 2008.

[27] K. J. Ottenstein, R. A. Ballance, and A. B. MacCabe. 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, 1990.

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

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

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

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

[32] R. E. Rodrigues, V. H. S. Campos, and F. M. Q. Pereira. A fast and low overhead technique to secure programs against integer overflows. In *International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2013.

[33] R. Sasnauskas, O. Landsiedel, M. H. Alizai, C. Weise, S. Kowalewski, and K. Wehrle. Kleenet: discovering insidious interaction bugs in wireless sensor networks before deployment. In *International Conference on Information Processing in Sensor Networks (IPSN)*. ACM, 2010.

[34] E. J. Schwartz, T. Avgerinos, and D. Brumley. 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, 2010.

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

[36] D. Singh and W. J. Kaiser. The atom LEAP platform for energy-efficient embedded computing. Technical Report 88b146bk, UCLA, 2010.

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

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

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