Securing Networked Embedded Systems Code through Distributed Systems Analysis

Fernando A. Teixeira, José M. S. Nogueira *Member, IEEE,* Leonardo B. Oliveira *Member, IEEE*

◆

Abstract—The growing importance of Internet of Things (IoT) calls for tools able to provide users with secure systems. Traditional approaches to analyze distributed systems are not expressive enough to address this challenge. As a solution, we present a framework to analyze networked embedded systems. Our key insight is to look at a distributed system as a single body, and not as separate programs that exchange messages. We then can crosscheck information inferred from different nodes. 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 an inter-program view, 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. We have implemented our solution on top of the LLVM compiler, and have used it to secure six ContikiOS applications against buffer overflow attacks. Our solution produces code that is as safe as code secured by more traditional analyses; however, our binaries are on average 18% more energy-efficient.

1 INTRODUCTION

The emergence of the Internet of Things (IoT) has increased importance of Networked Embedded Systems (NES) [1]. In fact, 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. 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 [2], [3], [4].

Ensuring the security of such systems is a problem of increasing relevance [5]. DARPA has elected security the central point in its Cyber Grand Challenges call[1]. Two main factors make IoT security even more critical: (i) things act as bridges between user's physical and cyber worlds, and (ii) 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], it had targets like home routers, smart TVs, and smart appliances.

IoT faces a plethora of security problems [5]. It suffers from the same security issues as traditional Internet-based and/or wireless systems, including jamming, spoofing, replay, and eavesdropping [5], [6]. In addition, it is more prone to other issues such as out-of-bound memory accesses due to a few factors: 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, an inherently unsafe language [7]. Its semantics allows out-of-bound memory accesses. This type of accesses are dangerous because they give room to Buffer Overflow (BOF) attacks [8]. A BOF takes place whenever a system allows data to be accessed out of the bounds of an array.

Much work has already been done to turn C into a safer language (e.g. SAFECode [9] and AddressSanitizer (ASan) [10]). Existing proposals resort to Array-Bound Checks (ABC), which are tests done at runtime to ensure that a particular array access is safe. 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, ASan 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 work 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 IoT. 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 BOF attacks. We call our solution Securing Internet of Things (SIoT). More specifically, we applied Tainted Flow Analysis (TFA) [11] 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 ABC. Because the analysis has a holistic view of

- *F. A. Teixeira are with Universidade Federal de São João Del Rei, Brazil E-mail:teixeira@ufsj.edu.br*
- *J. M. S. Nogueira and L. B. Oliveira are with Universidade Federal de Minas Gerais, Brazil E-mail: jmarcos@dcc.ufmg.br,leob@dcc.ufmg.br*

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

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 work is just an instance of it.

Our contribution. 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: **1.** We propose an extension to the standard Control Flow Graph (CFG) [12], called Distributed Control Flow Graph (DCFG), 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 IoT against BOF, and can do so more efficiently than traditional approaches. More specifically: **1.** We have implemented our algorithm and its companion distributed TFA in the LLVM compiler [13]. **2.** We have applied this analysis on six applications present in ContikiOS [14], and the results show that our proposal is 18% more energy-efficient than existing solutions.

Publications. The problem and first idea of solution was published as part of an SBSeg13 tutorial [15]. Then, a first version of the SIoT with preliminary results was published in SBRC14. An extension of this work was published at Latin American Transactions [16]. In 2015, a complete version of SIoT was published at the IPSN [17][3]. Finally, we have published a thesis summary paper at the Security Thesis and Dissertation Competition (CTDSeg) of SBSeg 2016 [18]. The list of these publications can be seen below:

- Segurança de Software em Sistemas Embarcados: Ataques & Defesas. *SBSeg'2013*.
- Defending Code from the Internet of Things against Buffer Overflow. *SBRC'2014*.
- Defending Internet of Things against Exploits. *IEEE Latin America Transactions, 2015*.
- SIoT: Securing the Internet of Things through Distributed System Analysis. *IPSN'2015*.
- Securing Networked Embedded Systems through Distributed Systems Analysis. *SBSeg'2016*.

From our work was derived two undergraduate projects and one master degree dissertation. The undergraduate projects have finished in 2014. The master dissertation uses our framework to create a Distributed Range Analysis. It was conclude in 2015 and was published at SBSeg14 [19] and at Latin American Transactions 2015 [20].

Organization. Section 2 describes programming language concepts. Section 3 presents the attack model used in this paper. Section 4 presents our solution. Section 5 evaluates SIoT in terms of energy-efficiency. We conclude in Section 6.

---

3. IPSN is a flagship conference on networked embedded systems – http://ipsn.acm.org/

2 LANGUAGE-BASED TECHNIQUES FOR ADDRESSING BOF VULNERABILITIES

Software code can harbor different types of security vulnerabilities, and those susceptible to BOF attacks are the most exploited. Solutions to address this class of vulnerabilities have long existed, and are largely based on static analysis [7], dynamic analysis [10], or a combination of both. In static analysis, analysis is performed without actually running the program; instead either the source code or the object code is inspected, and vulnerabilities flagged. Dynamic analysis, 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. 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.

Code Analysis using CFG. The CFG is used to model the control flow of computer programs (Fig. 1). 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.

One class of potential BOF 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 3 for a discussion of these assumptions), then we would see the assignments in lines 1 and 5 (Fig. 1b) 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. 1a), which could provide malicious data from attackers (msg has been used in buffer access). The first assignment in server program (line 1, Fig. 1b) is not vulnerable because the data being assigned is a hard-coded constant from the client program (line 1, Fig. 1a).

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. 1b) and one false-positive (line 1, Fig. 1b). Our proposal addresses this shortcoming (Section 4).

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 Platform Module (TPM) [21] can be employed to ensure the integrity of nodes systems and cryptographic solutions like [22], [23] can be used to establish a secure communication channel.
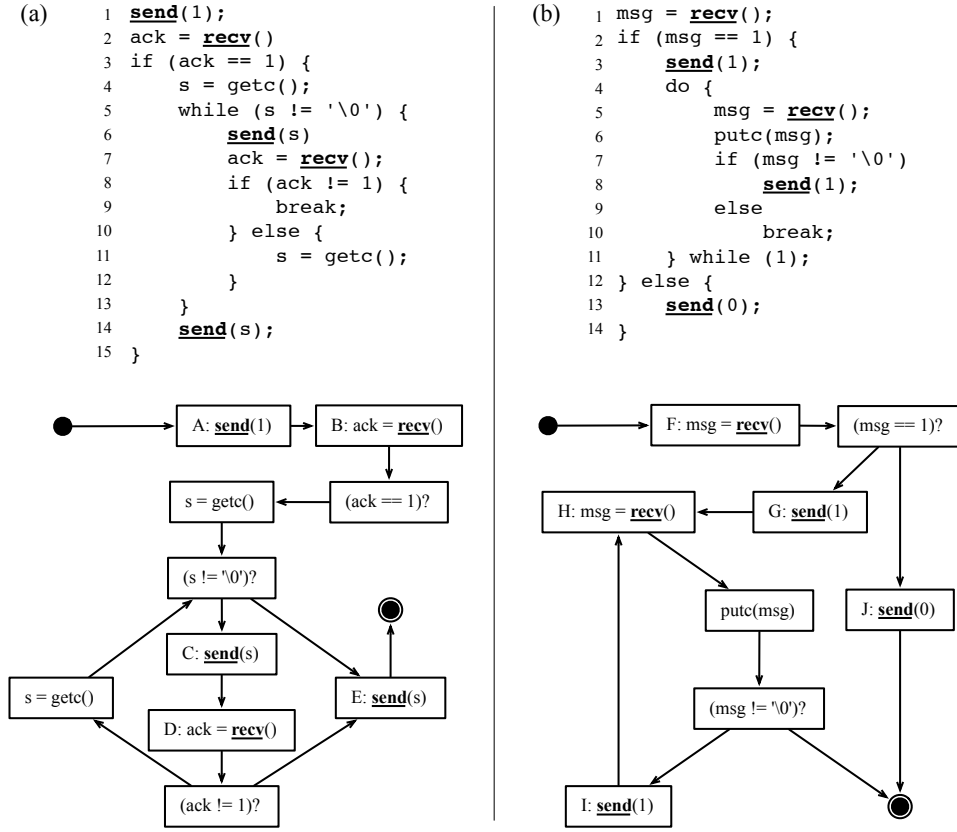
(a)
```
1  send(1);
2  ack = recv()
3  if (ack == 1) {
4      s = getc();
5      while (s != '\0') {
6          send(s)
7          ack = recv();
8          if (ack != 1) {
9              break;
10         } else {
11             s = getc();
12         }
13     }
14     send(s);
15 }
```

(b)
```
1  msg = recv();
2  if (msg == 1) {
3      send(1);
4      do {
5          msg = recv();
6          putc(msg);
7          if (msg != '\0')
8              send(1);
9          else
10             break;
11     } while (1);
12 } else {
13     send(0);
14 }
```

Fig. 1. Echo application's programs and their respective CFGs. (a) Echo client. (b) Echo server.

4 COMMUNICATION LINKS INFERENCE

CFGs model the control flow of individual programs. To analyze an entire distributed system, we need to work with CFGs 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 D

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 BOF. Attackers can then manipulate the environment (to produce spurious sensor readings) or provide spurious user input to launch a BOF 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.

the resulting DCFG. D contains C_1 and C_2 as a subgraph. Inter-program edges connecting C_1 and C_2 are then added to 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 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 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. 1, 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 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 S and Receive-Graph R as follows. For each vertex $v \in \mathcal{C}$ labeled with a send operation, we add a vertex v' to S. We also add $start'$ and $exit'$ vertices, which correspond to start and exit in $\mathcal{C}$. Edges in 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 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 R in a similar way, replacing sends by recvs in the procedure described above.

Next, we move on to the concept of *level*. Given a Send-Graph,

---

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 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
$D \leftarrow \mathcal{C}_1 \cup \mathcal{C}_2$
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 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 D

---

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. 2. Consider echo client program Send-Graph (Fig. 2 left-hand-side). 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. 1, their resulting DCFG can be built by linking the SEND vertices in one CFG with the RECV 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. The steps describe above are captured in the *Elevator Algorithm* (Algorithm 1). (See Section 3.4, in the Thesis, for formalization of Elevator algorithm, with a prove of its correctness and termination.)

5 EVALUATION

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 ASan 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 six pairs of ContikiOS applications (Table 1). 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. This table shows that ASan introduces between 3,736 ABCs (*udp-ipv6 client/server*) and 7,453 ABCs (*ping6 / new-ipv6*). The Baseline approach reduces ASan's numbers substantially, between

170 (*ipv6-rpl-udp client/server*) and 214 (*rest-server/coap-client*), because it is eliminating every guard that is not influenced by data coming from an external function. 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. 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.

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 six ContikiOS applications. 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 on this experiment, we have installed the applications in *IRIS XM2110* sensors and have measured the amount of energy that they consume. Our results (Table 2) show that SIoT outperforms Baseline for all applications. On average SIoT is 18% more energy efficient than Baseline.

6 CONCLUSION

This work has presented a framework for analysis and optimization of distributed system, which we have used to protect IoT systems against BOF attacks. Our framework 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 created SIoT to instantiate a version of TFA that points out which memory accesses need to be guarded against BOF 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 our framework 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 our framework to enable compiler optimizations. As an example, if we go back to Figure 1, 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.

Thesis Links. The Thesis text is publicly available at http://homepages.dcc.ufmg.br/~teixeira/teixeira-thesis.pdf. The SIoT code is publicly available at http://cuda.dcc.ufmg.br/siot/.

REFERENCES

[1] E. Borgia, "The internet of things vision: Key features, applications and open issues," *Computer Communications*, vol. 54, pp. 1–31, 2014.

[2] L. Atzori, A. Iera, and G. Morabito, "The Internet of Things: A survey," *Computer Networks*, vol. 54, no. 15, pp. 2787–2805, 2010.

[3] D. F. Macedo, S. de Oliveira, F. A. Teixeira, A. L. L. Aquino, and R. A. Rabelo, "CIA2-ITS: Interconnecting mobile and ubiquitous devices for Intelligent Transportation Systems," in *PerCom*. IEEE, 2012.

[4] F. A. Teixeira, V. F. e Silva, J. L. Leoni, D. F. Macedo, and J. M. Nogueira, "Vehicular networks using the ieee 802.11 p standard: an experimental analysis," *Vehicular Communications*, vol. 1, no. 2, pp. 91–96, 2014.

[5] 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*, vol. 61, no. 3, pp. 527–542, 2011.
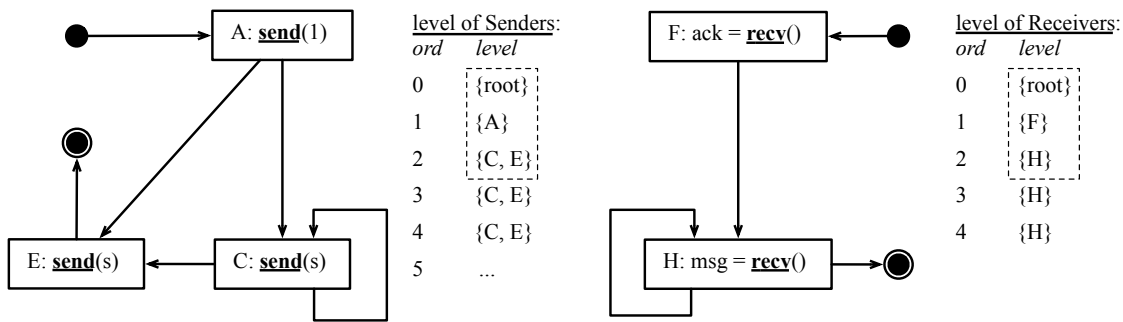
Fig. 2. Levels for echo client's SENDs (left-hand-side) and echo server's RECVs (right-hand-side).

TABLE 1
ABCs inserted by ASan, Baseline, and SIoT.

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%

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

[6] 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.

[7] B. Chess and J. West, *Secure Programming with Static Analysis*, 1st ed. Addison-Wesley Professional, 2007.

[8] C. Cowan, F. Wagle, C. Pu, S. Beattie, and J. Walpole, "Buffer overflows: attacks and defenses for the vulnerability of the decade," in *(DISCEX)*. DARPA, 2000.

[9] D. Dhurjati, S. Kowshik, and V. Adve, "SAFECode: enforcing alias analysis for weakly typed languages," in *PLDI*. ACM, 1996.

[10] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Address-Sanitizer: a fast address sanity checker," in *Annual Technical Conference (ATA)*. USENIX, 2012.

[11] 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 *(S&P)*. IEEE, 2008.

[12] F. E. Allen, "Control flow analysis," *ACM Sigplan Notices*, vol. 5, pp. 1–19, 1970.

[13] C. Lattner and V. S. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *CGO*. IEEE, 2004.

[14] A. Dunkels, B. Gronvall, and T. Voigt, "Contiki - a lightweight and flexible operating system for tiny networked sensors," in *LCN*. IEEE, 2004.

[15] B. Silva, D. Cecilio, E. M. Souza, F. A. Teixeira, H. C. Wong, and H. Nazaré, "Segurança de Software em Sistemas Embarcados: Ataques & Defesas," in *Minicursos do XIII SBSeg*, 2013, pp. 101–155.

[16] F. A. Teixeira, G. V. Machado, P. M. Fonseca, F. M. Q. Pereira, H. C. Wong, J. M. S. Nogueira, and L. B. Oliveira, "Defending Internet of Things against Exploits," *IEEE Latin America Transactions*, vol. 14, no. April, 2015.

[17] F. A. Teixeira, G. V. Machado, F. M. Pereira, H. C. Wong, J. Nogueira, and L. B. Oliveira, "SIoT: Securing the Internet of Things through Distributed System Analysis," in *Proceedings of 14th IPSN*. ACM, 2015, pp. 310–321.

[18] F. A. Teixeira, J. M. S. Nogueira, and L. B. Oliveira, "Securing networked embedded systems through distributed systems analysis," in *Concurso de Teses e Dissertações (CTDSeg), SBSeg*, 2016.

[19] V. M. Paisante, L. F. Z. Saggioro, R. E. Rodrigues, L. B. Oliveira, and F. M. Q. Pereira, "Prevençao de ataques em sistemas distribuídos via análise de intervalos," in *XIV SBSeg*, 2014.

[20] L. F. Z. Saggioro, V. M. Paisante, R. E. Rodrigues, L. B. Oliveira, and F. M. Q. Pereira, "Crosschecking distributed data to detect integer overflow," *IEEE Latin America Transactions*, vol. 14, no. April, 2015.

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

[22] A. Perrig, R. Szewczyk, V. Wen, D. Culler, and J. D. Tygar, "SPINS: Security protocols for sensor networks," *Wireless Networks*, vol. 8, no. 5, pp. 521–534, 2002.

[23] T. Kothmayr, W. Hu, C. Schmitt, M. Bruenig, and G. Carle, "Poster: Securing the internet of things with DTLS," in *SenSys*. ACM, 2011.