# Securing Networked Embedded Systems through Distributed Systems Analysis

**Fernando A. Teixeira**[1], **José Marcos S. Nogueira, Leonardo B. Oliveira**

[1] UFSJ – Ouro Branco, MG, Brazil. UFMG – Belo Horizonte, MG, Brazil

teixeira@ufsj.edu.br,{jmarcos,leob}@dcc.ufmg.br

***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) [Borgia 2014]. 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.

IoT faces a plethora of security problems [Heer et al. 2011]. It suffers from the same security issues as traditional Internet-based and/or wireless systems. 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 [Chess and West 2007]. Its semantics allows out-of-bound memory accesses. This type of accesses are dangerous because they give room to Buffer Overflow (BOF) attacks. 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 [Dhurjati et al. 1996] and AddressSanitizer (ASan) [Serebryany et al. 2012]). 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) [Balzarotti et al. 2008] on the model we proposed, and sanitized C programs against out-of-bound memory accesses. TFA tracks potentially malicious data (i.e., data that can be influenced by attackers) flows across the program. Memory indexed by tainted data can then be guarded against invalid access during runtime using ABC. 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 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) [Allen 1970], 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 [Lattner and Adve 2004]. **2.** We have applied this analysis on six applications present in ContikiOS [Dunkels et al. 2004], and the results show that our proposal is 18% more energy-efficient than existing solutions.

**Publications.** The problem and first idea of solution was published as part of an SBSeg13 tutorial. 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. In 2015, a complete version of SIoT was published at the IPSN [Teixeira et al. 2015][1]. 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.*

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 and at Latin American Transactions 2015.

## 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 [Chess and West 2007], dynamic analysis [Serebryany et al. 2012], 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).

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

 

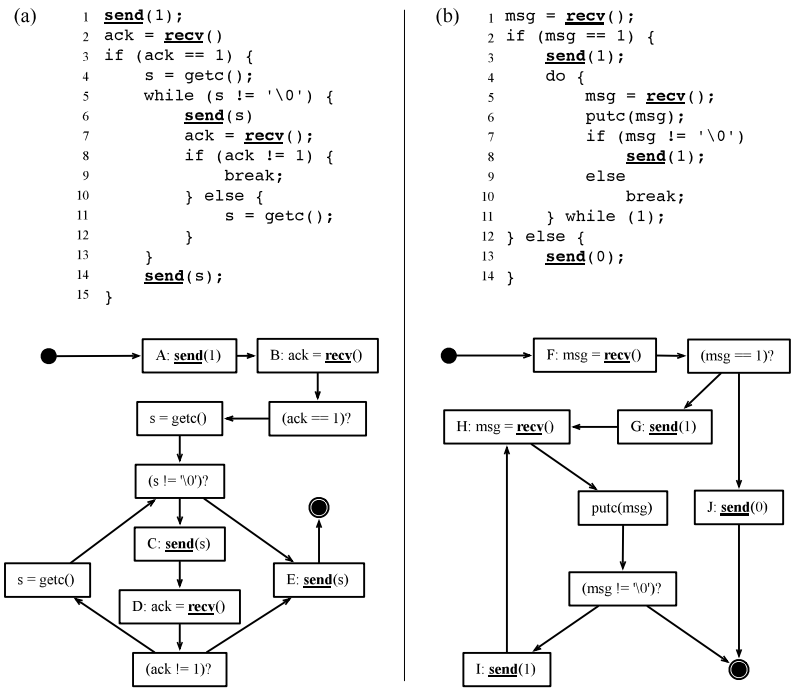


**Figure 1. Echo application's programs and their respective CFGs. (a) Echo client. (b) Echo server.**

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) [Kinney 2006] can be employed to ensure the integrity of nodes systems and cryptographic solutions like [Perrig et al. 2002, Kothmayr et al. 2011] 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 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.

## 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 $D$ 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, 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.

 

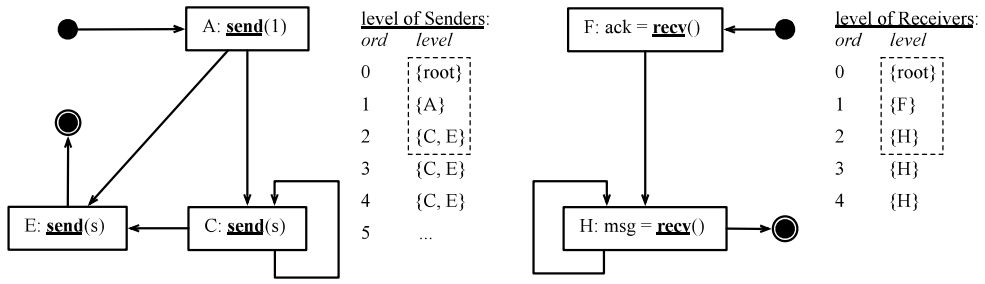


**Figure 2. Levels for echo client's SENDs (left-hand-side) and echo server's RECVs (right-hand-side).**

**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 ∈ {S_1, S_2} ∪ {R_1, R_2} do
    n ← 0
    L_{G_i,n} ← {root}
    ▷ While the new generated set L_{G_i,n} is unique
    while L_{G_i,n} ≠ L_{G_i,0..n-1} do
        foreach vertex v in L_{G_i,n} do
            S_succs ← successors of v
            L_{G_i,n+1} ← L_{G_i,n+1} ∪ S_succs
        n ← n + 1

▷ Link SENDs and RECVs of the same level
D ← C_1 ∪ C_2
for k ← 1 to n do
    foreach v_s ∈ L_{S_1,k} and v_r ∈ L_{R_2,k} do
        add an edge from v_s to v_r in D
    foreach v_s ∈ L_{S_2,k} and v_r ∈ L_{R_1,k} do
        add an edge from v_s to v_r in D
```

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

 

**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 |

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/.

**Acknowledgment.** We thank Fernando Pereira and Hao Chi Wong for their valuable contributions. This work was partially supported by Intel Corporation, CNPq and FAPEMIG.

## References

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

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

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

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

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

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

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

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

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

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

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

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

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

 