2016 Intl IEEE Conferences on Ubiquitous Intelligence & Computing, Advanced and Trusted Computing, Scalable Computing and Communications, Cloud and Big Data Computing, Internet of People, and Smart World Congress

RPiDS: Raspberry Pi IDS
A Fruitful Intrusion Detection System for IoT

Alessandro Sforzin† and Mauro Conti
University of Padua
Via Trieste, 63, Padua, Italy
alessandro.sforzin@neclab.eu, conti@math.unipd.it

Félix Gómez Mármol and Jens-Matthias Bohli
NEC Laboratories Europe
Kurfürsten-Anlage, 36, Heidelberg, Germany
{felix.gomez-marmol,jens-matthias.bohli}@neclab.eu

***Abstract*—Our technology keeps advancing towards a future where everything is connected together. The Internet of Things (IoT) goal is to make every device accessible from the Internet. Even the most common electrical appliances, such as ovens and light bulbs, will have their own IP address, and will be reachable remotely. While this enhanced connectivity will definitely improve our quality of life, it also raises serious security, privacy and trustworthiness questions; the resource constrained nature of IoT entities makes traditional security techniques impractical.**

In this paper, we propose an intrusion detection architecture for the IoT. We discuss the feasibility of employing a commodity device as the core component of the architecture. In particular, we evaluated the performance of the Raspberry Pi, one of the most used commodity single-board computers, while running Snort, a widely known and open source Intrusion Detection System (IDS). Our experiments show that our proposed architecture based on resource constrained devices, such as the Raspberry Pi, can effectively serve as IDS in a distributed system such as IoT.

***Index Terms*—Intrusion Detection, IoT, Snort, Raspberry Pi.**

I. Introduction

A distributed system such as the Internet of Things (IoT) envisions a world in which every *thing* in our everyday lives is connected together. This implies a deep change in how we interact with our surroundings, and will require new strategies to manage the stream of information available to us at any given time.

The IoT vision does not come without serious challenges, though. IoT environments will be mostly comprised of small, battery powered, devices. For example, sensors are devices capable of performing monitoring tasks autonomously, but will avoid performing computationally expensive tasks (like certain cryptographic primitives), as these kinds of operation will deplete their battery very quickly.

Additionally, while this overwhelming connectivity provided by IoT will be crucial, there is a downside to it: the right to privacy is belittled. Nowadays, such privacy issue is becoming more and more relevant. Nonetheless, the advent of IoT technologies will probably accentuate the problem rather than provide a cure for it, because most of the time the exchange of information between IoT nodes will happen silently in the background unbeknownst to users.

† This work was carried out over the course of an internship at NEC Research laboratories in Heidelberg, Germany.

The prospect of an enormous amount of sensitive information transmitted between sensor nodes is very attractive to malicious third parties, which makes the IoT infrastructure a primary target of cyber-attacks. Research has shown vulnerabilities in a plethora of devices, including cars [1], baby monitors [2], medical devices [3], and even light bulbs [4]. Since IoT nodes mostly use wireless communication technology to exchange information, they are vulnerable to eavesdropping and man in the middle attacks. There is also the risk of tampering because, more often than not, IoT nodes are left unattended. Furthermore, conventional cryptography techniques such as public key cryptography are too costly, in terms of energy and bandwidth, to be implemented on IoT environments [5]. Although lightweight cryptography solutions are being researched [6], [7], they are not enough to protect the network from internal attackers (i.e., nodes that have been compromised), or from attacks the likes of DDoS, resource exhaustion, or selective forwarding.

In this paper our driving motivation was to find a robust and scalable security solution to protect *things* from so called cyber-attacks. We propose an IDS scheme applicable to a wide variety of IoT layouts, from city squares to users' smart homes. Our main contributions are:

- The proposal of an IDS architecture, which could be effectively applied to IoT environments (Section III).
- The discussion about an actual deployment of Snort [8] on a Raspberry Pi, a low powered device which could be found in a typical IoT environment.
- The evaluation of the Raspberry Pi's performance as the Snort's host (Section IV).

We outline future research lines and conclude the paper in Section V.

II. Related Work

There is a plethora of studies on Snort and its performance, but only few of them focus on assessing its impact on resource constrained devices.

One interesting work is [9], in which the authors investigate the performance of Snort and Bro on Wireless Mesh Networks (WMNs). Their study shows that those IDSs' modules plus deep packet inspection are too resource demanding for WMNs nodes, making them unsuitable as a security solution for WMNs. To address the problem, they present a lightweight

978-1-5090-2771-2/16 $31.00 © 2016 IEEE
DOI 10.1109/UIC-ATC-ScalCom-CBDCom-IoP-SmartWorld.2016.114

IDS for Wireless Mesh Networks that decreases memory consumption and packet drop rate in such resource constrained nodes. Their proposed solution, however, detects only few types of attacks, such as resource consumption attacks, IP spoofing, and spam email distribution.

In [10] the authors also argue about the infeasibility of deploying Snort in WMNs. They reach the same conclusion as the authors of [9]: using the full capabilities of Snort on WMNs nodes is impractical. They propose PRIDE (PRactical Intrusion DEtection in resource constrained wireless mesh network) as a solution that adapts Snort functionalities to WMNs by distributing them across the network. Each node will run a customized version of Snort that employs a different subset of signatures; different nodes will be in charge of detecting different types of attacks. This distribution of IDS functions is made in such a way that the network's detection coverage is maximized.

While these works highlight the performance of Snort on resource constrained devices, they do not target the Raspberry Pi. As we will see in Section IV-E, the causes of Snort inefficiency identified by the authors are the same we found in our study. However, in our case the Raspberry Pi hardware advantage mitigated the problems they faced in their experiments.

A study that focuses on the Raspberry Pi is [11], in which the authors compare the performance of Snort and Bro running on the small computer. They demonstrate that Snort has better performance than Bro. Our testing methodology differs from this work in that we tested with a wider spectrum of packet sizes and data rates. This gave us a better understanding of the performance of both the Raspberry Pi and Snort. Moreover, as we will show in Section IV-E, our findings suggest that the Raspberry Pi would be able to act as a single monitor node in a variety of network scenarios, whereas in [11] the authors come to the conclusion that the Raspberry Pi would not be able to fullfill such a role.

III. RPIDS

At the core of our intrusion detection architecture there is a small, portable device, pre-packaged with an IDS. The device could be seamlessly deployed anywhere (what we call *Plug & Protect*): users in a smart home could use it to protect their smart objects, city administrations could deploy it in streets, squares, university campuses, stadiums, or in other crowded areas, to monitor the network traffic of the surrounding area. It is effectively a portable, on demand, IDS that notifies the users, or the administrators of the network, whenever it detects an ongoing attack or suspicious network activities.

We will build this complex system with the following requirements in mind:

- *Portability*: since the goal is providing security and privacy on demand (i.e., everywhere, anytime), users must be able to carry the device with them, and network administrators must be able to relocate it effortlessly.
- *Minimum configuration*: the device should configure itself automatically for the most part, but power users will be able to further tailor it to their needs.
- *Ease of use*: users will only have to turn on the device to secure their communications.
- *Versatility*: the device could be used anywhere from stadiums to universities, public squares to streets, smart homes to users' pockets.

These requirements may sound familiar; they are the characteristics of a smartphone: it is portable, it requires very few configurations steps, and it is easy to use anywhere. This is not by chance; we propose a solution that users can get quickly familiar with. We moved away from smartphones because we found severe limitations that would prevent us from implementing our architecture on such platforms. First, we were unable to set the smartphone's Network Interface Controller (NIC) in promiscuous mode without root permissions. Second, some operative systems prevent applications and services from running continuously in the background, which is the main mode of operation of an IDS. Hence, we opted for the Raspberry Pi, a device free from these chains.

We put emphasis on the *portability* of the device[1]. In the IoT world connectivity is ubiquitous, therefore network security must be ubiquitous too. We propose a scalable framework which takes full advantage of the flexibility of the Raspberry Pi. Indeed, its main building block is a single Raspberry Pi equipped with Snort. At any time more Raspberry Pi devices could be added to or removed from a given area to tune its security level. In a likely scenario in which a user needs a single device to protect herself, intrusion detection is performed locally on the device, and the latter will notify the owner of any security threat detected.

These devices can also be used together to perform collaborative intrusion detection. Each device will still carry out intrusion detection locally, but will have the option of requesting traffic data from nearby nodes. Leveraging data from other nodes could help the detection task of each node, ease the monitoring job, and possibly reduce the false positives rate. Such a distributed and collaborative approach will give our security architecture the ability to detect attacks, such as network topology attacks, that are normally not detectable by a node working alone. In this scenario, users will be notified about irregularities in their proximity (e.g., ongoing cyber-attacks, or alerts triggered by one of their personal devices). For example, in a smart city there could be a cluster of Raspberry Pi devices monitoring the main city's square, and another cluster monitoring the city's stadium. Exchange of information between the two separate clusters would also be possible in order to further enhance detection.

Each device will collect attacks' statistics locally, and then send them to a remote server running a Security Information and Event Management (SIEM) software, from which network administrators can perform maintenance or emergency operations. Collecting data in this way may be useful for administrators to figure out the frequency and characteristics of attacks in a given area (e.g., the stadium in the previous example).

[1]From now on, when we write *device* or *node*, we refer to a hardware system comprised of, among other parts, a Raspberry Pi with Snort installed.

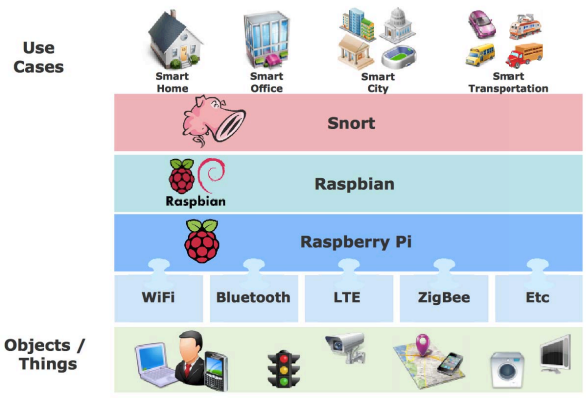

Fig. 1: High level structure of RPiDS

This in turn, would help in identifying the appropriate security measures to protect those areas [12].

Figure 1 shows a high level structure of the project. The device finds its place in a wide variety of IoT environments to monitor and communicate with a plethora of smart objects. One of the problems of IoT technologies is that there is no unique standard used by smart objects to "talk to each other". This severely limits the collaboration between them. Moreover, each major company develops its own standard and tries to encourage the rest of the world to use it. Therefore, our device will be able to employ different protocols while performing its duties. For example, Zigbee [13] is a widely used protocol for low powered hardware. Another example is the use of low power WiFi, which recent studies have shown to be a promising technology for IoT environments [14].

In the study we present in this paper, we tested the Raspberry Pi with a smart home scenario in mind. The simplest setting would be positioning the Raspberry Pi near the home network's router, and connect the former to the latter via Ethernet interface. Additionally, since it needs to inspect all the traffic incoming and outgoing from the router to protect the network, we configured the Raspberry Pi's NIC to listen in *promiscuous* mode. We stress here that this is only the first step in our roadmap; the final product will be able to perform its operations over wireless mediums. The choice between wired or wireless mode of operations will be left to the end user, and network administrators.

A. RPiDS Components

Following a step-by-step approach we, first, outlined the hardware characteristics of the base device; second, chose the IDS we wanted to use; third, assessed their compatibility. Concretely, we identified the Raspberry Pi as suitable base hardware, and Snort as the IDS for our system. Even though the Raspberry Pi does not have an embedded wireless card[2], we can extend its capabilities by using external hardware (e.g., USB antennas).

In this paper, we will focus on describing the methodology and results of our tests to evaluate the Raspberry Pi-Snort affinity. In Section IV we will show that the Raspberry Pi is a resilient computer, capable of handling Snort's operations flawlessly. Our main goal was to gain a better understanding of the minimum hardware requirements. Consequently, we investigated the feasibility of using a weaker than average machine as a network monitor by testing its performance while running intrusion detection functionalities. This is in stark contrast with most IDSs designs, in which a base station in charge of doing the heavy computations is assumed to be resource unconstrained.

1) Raspberry Pi: The first version of this credit card sized computer dates back to 2012. Since that time, a few increasingly powerful models have been released to the public. The one we used, called Raspberry Pi 2 Model B, was released in February 2015, and comes with a quad-core processor with 1GB of memory. We used such device because we thought that it is the only model that may be able to keep up with the speed of our networks, and the amount of traffic going through the wire.

Today, the Raspberry Pi is used in a wide variety of projects, from weather stations to home automation; it is also widely employed in academic environments around the world as a teaching instrument. People and organizations take advantage of its size, portability, cost, programmability, and connectibility. We feel that the Raspberry Pi is a solid choice for our system, given its proved flexibility and positive feedback.

2) Snort: To carry out intrusion detection, we installed Snort [8] on the Raspberry Pi. It is an IDS capable of real-time traffic analysis and packet logging. We chose Snort because it is the de facto standard of IDSs, open source, and always up to date with new signatures to match the latest security threats. Furthermore, it is employed both by private users and organizations as a security solution to monitor networks with speeds in the order of Gbit/s.

We initially considered Suricata [15] as a valid alternative; it is open source too, and its signatures database is comparable with Snort's one[3]. Suricata's most appreciated feature is its multi-threaded design, which takes full advantage of multi-core environments. In contrast, Snort's most criticized feature is its single-threaded design, which uses only one CPU to perform its operations, even in a multi-core environment. Nevertheless, "if the bandwidth being passed by the network interface associated with a Snort instance is greater than it can handle, more instances of Snort can be launched, and the traffic can be load balanced across the instances" [16]. In

[2]While the latest version of Raspberry Pi (version 3) is indeed equipped with a wireless interface, this work was conducted using a Raspberry Pi version 2 (https://www.raspberrypi.org/products/raspberry-pi-2-model-b)

[3]In practice, they fetch signatures from the same sources, but Snort uses an additional group of signatures called *Shared Object Rules*, a special type of rules used to extend Snort's detection capabilities.

this scenario, each Snort instance would run on a single CPU core, and a load balancer would redistribute the network traffic between them. Indeed testing of Suricata, as well as testing of multiple instances of Snort with a load balancer distributing the traffic, is worth being considered as a future work.

IV. Experimental Evaluation

Our main goal is assessing the Raspberri Pi's performance as the Snort's host. We judged its capabilities by determining at what data rates we would see packet drops, and which Snort's configuration the Raspberry Pi would be able to run steadily. We considered as "steady" Snort's executions that the Raspberry Pi sustained without running out of resources or without losing packets.

A. Hardware Setup

Our tests' setup consisted of a Raspberry Pi 2 Model B and a computer, connected together through their Ethernet interface. We kept both devices in an isolated environment; they were disconnected from the internet and from any other device. It is crucial to avoid any interference, because we might get inaccurate packet capturing results. Interferences might also cause the Raspberry Pi to lose some computational power because its resources are used elsewhere (i.e., serving other processes generated by unforeseen events in the network). Conducting the tests in such a controlled environment enhances the accuracy of the results.

We present here the full configuration of the two main entities involved in our experiments:

- The Raspberry Pi 2 Model B is a Quad-Core ARM Cortex-A7 with 1 GB of memory. We installed the operative system Raspbian (Debian Wheezy), and Snort 2.9.7.5. We equipped the Raspberry Pi with a 32 GB class 10 MicroSD card.
- The computer's hardware configuration was a 3.3 GHz Intel Core i5 with 8 GB of memory. We installed Ubuntu 15.04, and `tcpreplay` 4.1.0 to replay network traffic at different speeds [17].

The speed of the link between the two devices was 100 Mbit/s. It is worth noting, though, that the Raspberry Pi does not have a dedicated NIC (Network Interface Controller) like normal computers do; rather the NIC is part of the LAN chip, which consists of a USB 2.0 Hub and a 10/100 Ethernet Controller. Therefore, the USB Hub and the Ethernet Controller communicate with the System on a Chip (SoC) over the same USB 2.0 bus, which might cause the two components to interfere with each other.

B. Network Traffic Simulation

We carried out the tests with network traffic recorded in trace files. In [18] the authors argue that trace-driven simulation might not be the best option, because the timings of the packets are influenced by the network's condition when they where recorded. We used traces because our purpose was neither testing a specific protocol performance nor reproducing a specific network behavior. Employing traces allowed us to easily test Snort's behavior on the Raspberry Pi, by simply replaying traces storing both malicious traffic and regular traffic. A possible disadvantage is that there are a lot of different kinds of traffic patterns in today's Internet, each with its own unique characteristics, yet a trace file can represent only a few of them. Accordingly, we used multiple trace files from different sources of network traffic, to represent a wide range of network behaviors. We downloaded the traces from an online public repository [19], and divided them into three categories, based on the size of the packets they stored:

- *small*: traces in this category are mostly comprised of small-sized packets (less than 150 bytes).
- *long*: traces in this category mainly store packets of at least 1000 bytes.
- *mixed*: traces in this category are a good assortment of packets of all sizes; short packets and long packets are present in approximately the same amount, balancing each other out.

To be more precise, the first two categories contain packets of all sizes too, but they have one range of packet lengths that vastly outnumber the others (less than 150 bytes for the *small* category, and more than 1000 bytes for the *long* category). We focused on packet length because it influences both network overhead and throughput. The reasons are evident: in addition to the actual data, each packet contains some overhead information to perform control operations (such as routing or verifying the packets). As the packets shrink in size, that is, their payload data becomes smaller, the payload to overhead ratio favors the overhead information. This, in turn, means that the majority of the bandwidth is consumed to send overhead information rather than actual data. Another clear example is shown in Figure 2, which shows how packet size determines the packets per second rate in common Ethernet links speeds up to 1 Gbit/s. It is clear that small packet sizes require a much higher packet per second rate to reach the same throughput, defined in Mbit/s, as the bigger packet sizes. Moreover, although small packets have some advantages, such as better response times and less error rates, they have more overhead and cause more CPU utilization; the higher packet rate forces the receiver to process packets more quickly, thereby producing more CPU interrupts. As we expected that different packet sizes would produce very dissimilar results, we tested them separately by creating the aforementioned categories.

C. IDS Rules

Rules management is a crucial task for IDSs. In our experiments we used three basic rule sets, provided by Snort, which provide different levels of security. These are good starting points to create custom rule sets that satisfy the desired security level. Ordinary users may want to use them unmodified though, because deciding what subset of rules to enable is a complex procedure that requires full knowledge and understanding of the network's services and underlying

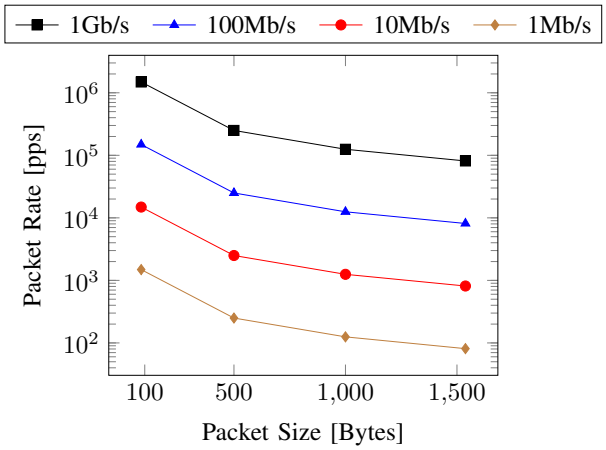

Fig. 2: Relationship between packet size and pps for Ethernet links of various speeds.

mechanisms. Enabling a lot of rules blindly is not the recommended approach; having more rules active does not mean more security, but rather it means more CPU and memory consumption for the IDS' host. We will show this effect in Section IV-E.

Here we list the three rule sets in order of increasing security:

- *Connectivity over Security*: the smallest set, and the less restrictive one. To minimize false positives, it will generate alerts with only well known malicious traffic. It will load ∼1k rules.
- *Balanced*: the "middle ground" set. As the name suggests, it is a good trade off between the less restrictive "Connectivity over Security", and the most restrictive "Security over Connectivity". It will load ∼7k rules.
- *Security over Connectivity*: the most secure and stringent set of the three, but also the most computationally expensive, because it enables more rules than the previous two. It will load ∼11k rules.

We used these three categories unmodified, because we wanted to test mostly whether the Raspberry Pi's hardware was able to handle a resource demanding software such as Snort. We also kept track of the number of alerts it generated for each test run.

D. Experiments Procedure

Algorithm 1 outlines the test strategy used for each of the three packet size categories we mentioned in Section IV-B. We wrote a python script to execute the whole procedure multiple times.

1) Input Parameters: The *for* loops variables, presented in Table I along with the range of values the algorithm assigns them, are the main features we selected. The outer loop iterates through the three packet size categories we mentioned in Section IV-B. The *DetectionEngine* and *RuleSets* loops refer to Snort's configuration. Since the detection engines have

Algorithm 1 Testing Algorithm

```
1:  procedure TEST
2:      for all pkt_size ∈ SizeCategories do
3:          for all engine ∈ DetectionEngines do
4:              for all ruleSet ∈ RuleSets do
5:                  for all speed ∈ ReplaySpeeds do
6:                      for i ← 1, iterations do
7:                          Start Snort and Resource Monitor
8:                          Replay Traces
9:                          Stop Snort and Resource Monitor
10:                     end for
11:                 end for
12:             end for
13:         end for
14:     end for
15: end procedure
```

Name	Description
SizeCategories	{small, long, mixed}
DetectionEngines	{lowmem, ac-bnfa, ac-split}
RuleSets	{connectivity, balanced, security}
ReplaySpeeds (Mbps)	{10, 20, 30, 40, 50, 60, 70, 80, 94}
Iterations	5

TABLE I: Input Variables

different computational requirements[4], they may play a part in the Raspberry Pi's capability to handle the incoming stream of traffic. The replay speed is a parameter we set in `tcpreplay`; we determined its maximum value by doing a series of tests with iperf [20], a tool for measuring the maximum achievable bandwidth on IP networks, which reported a maximum throughput of 94 Mbit/s. This upper bound is a good result, for it is stated in [21] that the maximum TCP throughput for a 100 Mbps Ethernet link, as the one between the Raspberry Pi and the computer, is 94.9 Mbit/s. As for Snort's rule sets, we have already introduced them in Section IV-C.

2) Monitored Statistics: We recorded several statistics for both the Raspberry Pi and Snort. Table II reports a brief description of those we found more relevant. We wrote a python script that computes the CPU and RAM usage twice per second. We used the percentage of packets successfully captured as an indicator of the Raspberry Pi load: as the replay speed increased, we expected to see smaller and smaller percentages of captured packets. Finally, the number of alerts generated by Snort was recorded for each experiment.

3) Inner Loop: The inner loop of the algorithm performs the crucial steps of our testing procedure. We repeated each test five times per configuration. We chose to iterate five times because we found it to be a good balance between meaningful test results and time needed for a test to complete.

Inside each inner loop's iteration, the algorithm performs the following steps:

[4]For a detailed description of Snort's available detection engines, refer to http://manual.snort.org/node16.html.

Name	Description
Avg. CPU	Average Raspberry Pi CPU usage for a given experiment
Avg. RAM	Average Raspberry Pi RAM usage for a given experiment
Pkt Capture Rate	% of packets dropped by Raspberry Pi for a given experiment
Alerts	No. alerts generated by Snort for a given experiment

TABLE II: Statistics monitored during the experiments

- *Start Snort and Resource Monitor*: the script, first, starts Snort with the detection engine and rule set selected in the outer loops; second, starts the python script that will gather information about hardware resource consumption (CPU and RAM).
- *Replay Traces*: the script sets up `tcpreplay` to replay all the traces in the *pkt_size* category at the given *speed* via the ethernet interface. As mentioned in section IV-B, we divided the traces in three different categories, based on packet size. Additionally, at each iteration `tcpreplay` replays the category's traces in a random order. For each category, the combined length of its traces is approximately a million packets.
- *Stop Snort and Resource Monitor*: after `tcpreplay` has finished replaying all the traces, the script stops Snort and the python script in charge of monitoring the CPU usage. We made it so that both Snort and the resource monitor program will print their final output, that is, the relevant statistics reported in table II, in a file.

E. Experimental Results

Figure 3 shows relevant relationships between input parameters and output statistics. We chose to emphasize the Raspberry Pi's behavior observed in response to a change in Snort's rule sets, rather than the one observed in response to a change in Snort's detection engine. With respect to the relationships we plotted, the difference between the three detection engines' performance was minor. Therefore, the plots in Figure 3 are the result of taking the mean of the data obtained from the various detection engines. The result is that each plot shows the mean curve of the detection engines. For example, Figure 3a shows the relationships between the data rate, in Mbit/s, and the CPU consumption of the Raspberry Pi with the *Connectivity* set loaded into Snort. Each curve in the plot refers to a different packet size, and each point of a curve is actually the mean of the values of the three detection engines for that point.

In the following sections, we will discuss the experimental results by analyzing the plots shown in Figure 3. We will analyze the Raspberry Pi's CPU and RAM consumption in sections IV-E1 and IV-E2 respectively. We will continue with the analysis of the Raspberry Pi's packet capture rate in Section IV-E3 and, lastly, discuss the varying number of alerts generated by Snort during our experiments.

1) CPU Usage: Figures 3a, 3b, and 3c show the relationship between the data rate, in Mbit/s, at which packets are transmitted from the computer, and the CPU consumption on the Raspberry Pi. The most striking result is that small packets cause substantial stress to the CPU, reaching 100% usage even at 10 Mbit/s, the lowest speed we used in our experiments. Though if we consider the tasks involved at the reception of a packet, this behavior is expected. When a packet arrives at a network card, the latter triggers an interrupt to notify the CPU about the presence of a new packet to be processed. As the number of packets per second (pps) increases, the number of interrupts starts to interfere with normal CPU operations; the interrupts become overwhelming, and since the CPU will try to serve them all, it will have little to no time to serve other processes. This phenomenon is called receive livelock [22]. Recall from figure 2 that for small packets to maintain the same throughput as large packets, they need a much higher rate. It follows that receive livelock is a sensible problem for small packets. For long packets the CPU reaches full utilization at much higher data rates, 70 Mbit/s in Figure 3a, against the 10 Mbit/s of small packets, with the mixed size case approximately in the middle of the two extremes.

2) RAM Usage: Figures 3d, 3e, and 3f show the effects on the RAM usage of the Raspberry Pi as the rule set changes. There is a clear difference in RAM consumption between the three rule sets, with the Connectivity rule set being the "lightest", and the Security rule set the most resource demanding of the three. Concretely, these plots show that switching to a bigger rule set results in higher RAM usage. The underlying reason is that Snort's detection engine loads into memory all the rules to perform its detection duties. Therefore, it's only natural that the detection engine needs more memory to load the Security rule set than the meager Connectivity ruleset.

3) Packet Capture Rate: Figures 3g, 3h, and 3i display the fraction of packets successfully processed at varying data rates. With respect to the traces with long and mixed packets, the capture rate is initially 100%, but then it starts decreasing. By comparing this set of graphs with the CPU Usage one, 3a, 3b and 3c, we can infer what is happening and why: the "breaking point", that is, when the packet capture rate starts decreasing, happens when the CPU approaches full utilization. This makes sense, and it agrees with the analysis of Section IV-E1: at high data rates the number of packets received, and consequently the number of interrupts generated, is more than the CPU can handle with the resources at its disposal, and will not be able to serve them all, hence new packets will be discarded. Regarding small packets, approximately half of them is discarded at the lowest speed. As the data rate increases, the situation gets worse, reaching a point where almost all of them are discarded. To complete the experiments, we did a small set of tests where we replayed the traces with small packets at data rates under 10 Mbit/s. It turns out that, with the traces we used, the highest data rate with observed 100% capture rate is 7 Mbit/s.

4) Alerts: Figures 3j, 3k, and 3l, show the number of alerts generated by Snort for the chosen traces. As we stated in Section IV-B, we employed some traces that stored malicious traffic. We would like to point out that the total expected

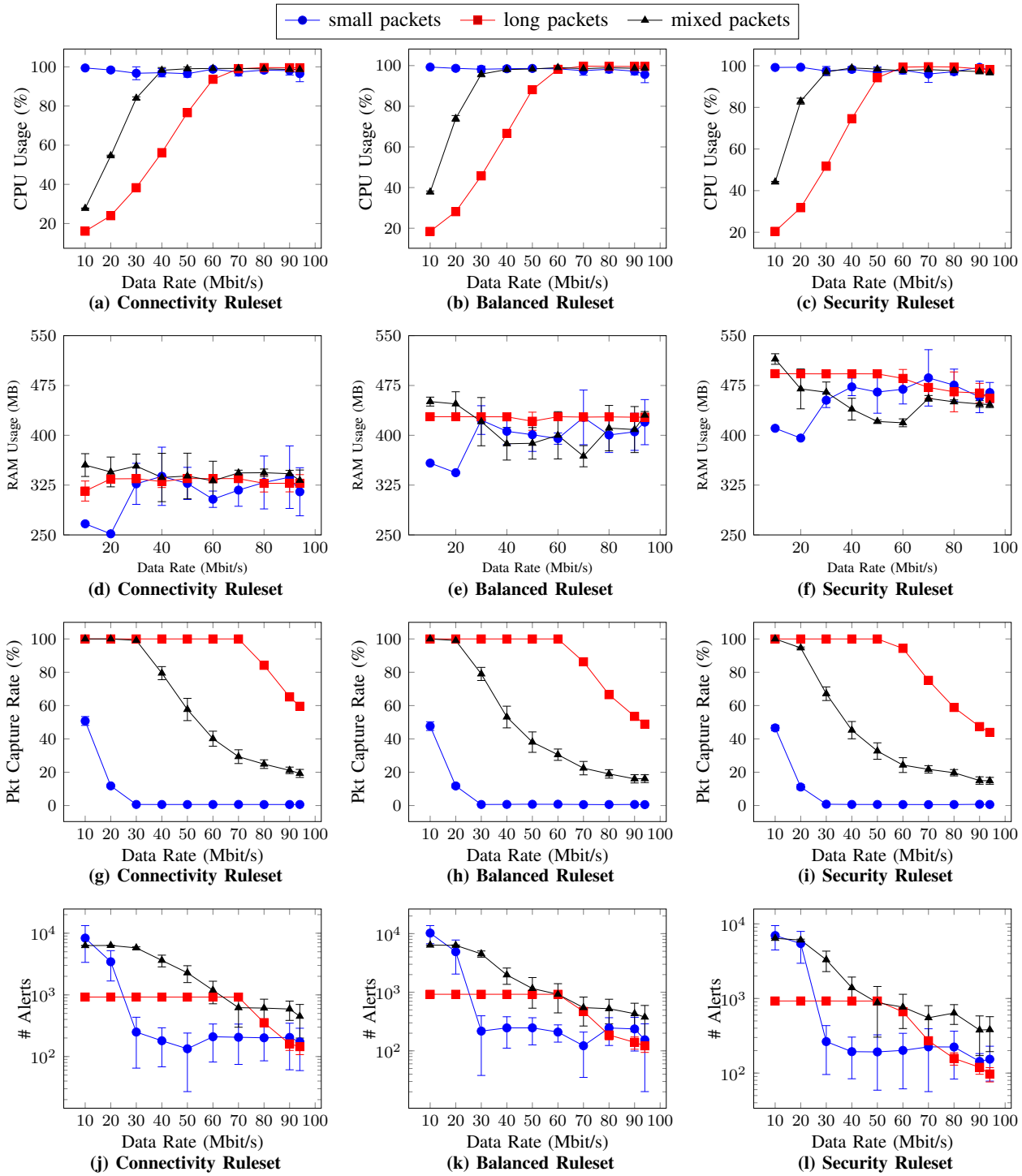

Fig. 3: Relationships between monitored variables as the signature set changes. a),b),c): Raspberry Pi's CPU usage variation as the data rate increases; d),e),f): Raspberry Pi's RAM usage variation as the data rate increases; g),h),i): Raspberry Pi's packet capture rate as the data rate increases; j),k),l): No. of Snort's generated alerts as the data rate increases.

number of alerts per packet size category was quite different, hence the difference in curve shapes observed in the plots. We determined the expected number of alerts by running Snort in pcap read mode, which allows Snort to analyze the traces offline. Concretely, the small packets category generated ∼20000 alerts, the mixed packets category generated ∼6000 alerts, and the long packets category generated ∼1000 alerts. These expectations are initially met by the large and mixed packets size categories, but as the data rate gets higher, the number of alerts gets lower. We can deduce the underlying reasons by joint analysis with the CPU and Packet Capture Rate plots: as the stressed CPU discards more and more packets, Snort receives less and less of them for inspection, which translated to less rules triggered, and therefore less alerts.

With respect to the small packets traces, we can see that at 10 Mbit/s Snort generates about 10000 alerts, which is quite far from the expected value. This is not surprising since figures 3g through 3i report that about 50% of the small packets are discarded for the same data rate. It follows that this lack of packets leads to the lower amount of alerts.

Finally, the random reordering of the traces at each iteration is likely the cause of the high error variability in these plots. First, the traces store different amounts of malicious traffic, and some of them store none; second, the Raspberry Pi's CPU discards packet as early as 10 Mbit/s. Therefore, at each iteration Snort will process only a small part of the traces' packets. For example, given a data rate of 50 Mbit/s, at a certain iteration Snort might process packets coming mostly from traces with little to no malicious traffic, the other packets being discarded by the CPU, therefore generating a low number of alerts. The next iteration might see a different order of traces, and so Snort might process mostly packets coming from traces with a high amount of malicious traffic, therefore producing a high number of alerts.

5) Connecting the Dots: In this section we will provide additional meaningful considerations on the outcome of our study.

The plots clearly show that switching to a bigger rule set leads to a slight decrease in performance. For example, from the CPU Usage plots we learn that going from the Connectivity set to the Balanced set causes the CPU to reach 100% usage 10 Mbit/s earlier for both mixed and long size packets. Changing to the Security set worsens the situation: the CPU consumption is very close to 100% as early as 50 Mbit/s for the long packets, which means "a loss" of approximately 20 Mbit/s for this packet size category. This confirms that activating signatures recklessly is not recommended. Although a bigger set of signatures carefully selected to match one's network needs might lead to more security, the resulting higher resources consumption of the IDS will take a toll on the system's resources.

The experiments' results suggest that the Raspberry Pi is capable of hosting Snort. We ran the tests for many consecutive days, during which we stressed the Rasbperry Pi with continuous streams of packets at various speeds, but it stood its ground very well. We think that it is certainly able to protect a smart home environment. In this scenario the number of *things* to protect is small, and the network load should be low or moderate. Given its versatility, it may also find its place in public transports; for example, it could be deployed on trains or on buses to protect their passengers. In these scenarios, the number of devices to monitor should be in the order of hundreds, which we think is something the Raspberry Pi can handle. A possible limitation is its Ethernet interface, which we think is outdated; in a world with Gigabit networks, a 100 Mbit/s interface seems inadequate.

Nevertheless, with the long packet size category the Raspberry Pi performed well until about 70 Mbit/s, when the first packet drops appeared. This, of course, is the best case scenario, but with small packets the little computer performed reasonably well too: in this case drops appeared around 8 Mbit/s, which is a speed in line with many home networks, and public WiFis. In very crowded areas the Raspberry Pi could still give some good results, though in those scenarios a Gigabit interface would greatly enhance its performance. However, as we stated in Section IV-A, the Raspberry Pi's Ethernet controller shares a USB 2.0 bus with the USB hub, meaning that even with a Gigabit Ethernet controller, the maximum theoretical speed would be limited to that of USB 2.0 (480 Mbit/s).

One should also consider that network load depends on the time of the day; network traffic during peak hours is much higher than network traffic during night hours. Consequently, estimations of the possible peak bandwidth consumption in each area are needed. Users in a public square, or in a library, will most likely use the bandwidth to do some simple operations, such as a quick internet search or checking their email, therefore the network may not even reach its full bandwidth usage. In such cases, RPiDS should perform well.

V. Conclusions

In this paper, we have proposed RPiDS, a novel IDS architecture for the IoT. At the core of our proposal stands a Raspberry Pi equipped with Snort, a full-fledged IDS. We argued that our architecture, based on a commodity device such as the Raspberry Pi, is suitable to perform intrusion detection in an IoT environment.

To support our claims, we carried out extensive tests involving the Raspberry Pi acting as a monitor node to analyze different types of network traffic. Additionally, we experimented with different Snort configurations, namely detection engine and number of rules loaded, because they influence CPU and RAM demands of Snort. Our experiments' results show that the Raspberry Pi is capable of hosting Snort, making RPiDS a feasible solution. Surprisingly, Snort resource demands did not overwhelm the Raspberry Pi: the main cause of full CPU usage were the high data rates, and memory never reached 100% utilization. Still, experiments with networks bigger than a smart home are needed to better grasp the true limits of the Raspberry Pi.

In the future, we plan to run experiments with intrusion detection over wireless protocols such as Wi-Fi and Bluetooth. We also plan to study a collaborative scenario, in which multiple RPiDS cooperate to perform intrusion detection. Additionally, we intend to analyze the possibility of running multiple instances of Snort, plus a load balancer, in a multi-core environment like the Raspberry Pi. Finally, we will also investigate alternatives to Snort; we are considering experimenting with Suricata, and Snort 3.

VI. Acknowledgments

Mauro Conti is supported by a Marie Curie Fellowship funded by the European Commission (agreement PCIG11-GA-2012-321980).

This work is also partially supported by the EU TagItSmart! Project (agreement H2020-ICT30-2015-688061), the EU-India REACH Project (agreement ICI+/2014/342-896), the EU SMARTIE Project (agreement FP7-ICT-609062), the Italian MIUR-PRIN TENACE Project (agreement 20103P34XC), and by the projects "Tackling Mobile Malware with Innovative Machine Learning Techniques", "Physical-Layer Security for Wireless Communication", and "Content Centric Networking: Security and Privacy Issues" funded by the University of Padua.

References

[1] A. Wright, "Hacking cars," *Communications of the ACM*, vol. 54, no. 11, pp. 18–19, 2011.

[2] K. Albrecht and L. McIntyre, "Privacy nightmare: When baby monitors go bad [opinion]," *Technology and Society Magazine, IEEE*, vol. 34, no. 3, pp. 14–19, 2015.

[3] W. B. Glisson, T. Andel, T. McDonald, M. Jacobs, M. Campbell, and J. Mayr, "Compromising a medical mannequin," *arXiv preprint arXiv:1509.00065*, 2015.

[4] N. Dhanjani, "Hacking lightbulbs: Security evaluation of the philips hue personal wireless lighting system," 2013.

[5] L. Atzori, A. Iera, and G. Morabito, "The internet of things: A survey," *Computer networks*, vol. 54, no. 15, pp. 2787–2805, 2010.

[6] M. Feldhofer, S. Dominikus, and J. Wolkerstorfer, "Strong authentication for RFID systems using the AES algorithm," in *Cryptographic Hardware and Embedded Systems-CHES 2004*. Springer, 2004.

[7] L. Eschenauer and V. D. Gligor, "A key-management scheme for distributed sensor networks," in *Proceedings of the 9th ACM conference on Computer and communications security*. ACM, 2002, pp. 41–47.

[8] M. Roesch, "Snort: Lightweight intrusion detection for networks," in *LISA*, vol. 99, no. 1, 1999, pp. 229–238.

[9] F. Hugelshofer, P. Smith, D. Hutchison, and N. J. Race, "OpenLIDS: a lightweight intrusion detection system for wireless mesh networks," in *Proceedings of the 15th annual international conference on Mobile computing and networking*. ACM, 2009, pp. 309–320.

[10] A. Hassanzadeh, Z. Xu, R. Stoleru, G. Gu, and M. Polychronakis, "PRIDE: Practical intrusion detection in resource constrained wireless mesh networks," in *Information and Communications Security*. Springer, 2013, pp. 213–228.

[11] A. K. Kyaw, Y. Chen, and J. Joseph, "Pi-IDS: evaluation of open-source intrusion detection systems on Raspberry Pi 2," in *2015 Second International Conference on Information Security and Cyber Forensics (InfoSec)*. IEEE, 2015, pp. 165–170.

[12] K. Scarfone and P. Mell, "Guide to intrusion detection and prevention systems (IDPS)," *NIST special publication*, vol. 800, no. 2007, p. 94, 2007.

[13] P. Baronti, P. Pillai, V. W. Chook, S. Chessa, A. Gotta, and Y. F. Hu, "Wireless sensor networks: A survey on the state of the art and the 802.15.4 and ZigBee standards," *Computer communications*, vol. 30, no. 7, pp. 1655–1695, 2007.

[14] S. Tozlu, M. Senel, W. Mao, and A. Keshavarzian, "Wi-Fi enabled sensors for internet of things: A practical approach," *Communications Magazine, IEEE*, vol. 50, no. 6, pp. 134–143, 2012.

[15] Open Information Security Foundation. (2010, July) Suricata: Open source IDS/IPS/NSM engine. [Online]. Available: http://suricata-ids.org/

[16] N. Houghton. (2010, June) Single threaded data processing pipelines and the Intel architecture. [Online]. Available: http://vrt-blog.snort.org/2010/06/single-threaded-data-processing.html

[17] F. Klassen. (2013, December) Tcpreplay: Pcap editing and replaying utilities. [Online]. Available: http://tcpreplay.appneta.com/

[18] S. Floyd and V. Paxson, "Difficulties in simulating the internet," *IEEE/ACM Transactions on Networking (TON)*, vol. 9, no. 4, pp. 392–403, 2001.

[19] NETRESEC. Publicly available PCAP files. [Online]. Available: http://www.netresec.com/?page=PcapFiles

[20] J. Dugan, S. Elliott, B. A. Mah, J. Poskanzer, and K. Prabhu. (2014, January) iPerf: The network bandwidth measurement tool. [Online]. Available: http://iperf.fr

[21] B. Constantine, R. Geib, R. Schrage, and G. Forget. (2011, August) RFC 6349: Framework for TCP throughput testing. [Online]. Available: http://tools.ietf.org/html/rfc6349

[22] J. C. Mogul and K. Ramakrishnan, "Eliminating receive livelock in an interrupt-driven kernel," *ACM Transactions on Computer Systems*, vol. 15, no. 3, pp. 217–252, 1997.