IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received January 30, 2019, accepted February 16, 2019, date of publication February 22, 2019, date of current version March 20, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2900939

IoTsafe, Decoupling Security From Applications for a Safer IoT

JORGE DAVID DE HOZ DIEGO, (Member, IEEE), JOSE SALDANA, (Member, IEEE), JULIÁN FERNÁNDEZ-NAVAJAS, AND JOSÉ RUIZ-MAS
I3A, University of Zaragoza, 50018 Zaragoza, Spain

Corresponding author: Jorge David de Hoz Diego (dhoz@unizar.es)

This work was supported in part by the Túneles Inteligentes y Seguros para Flujos IP con Baja Eficiencia Government of Spain under Grant TIN2015-64770-R, in part by the Spanish Ministry of Economy, Industry and Competitiveness, European Development Fund under Grant TIN2016-76770-R, in part by the Government of Aragon, in part by the Fondo Europeo de Desarrollo Regional Aragón Operative Programme 2014-2020 "Building Europe from Aragon" (Research Group T31_17R), and in part by the Consejo Nacional de Ciencia y Tecnología de México under Grant PEI 682/2014.

ABSTRACT The use of robust security solutions is a must for the Internet of Things (IoT) devices and their applications: regulators in different countries are creating frameworks for certifying those devices with an acceptable security level. However, even for already certified devices, security protocols have to be updated when a breach is found or a certain version becomes obsolete. Many approaches for securing IoT applications are nowadays based on the integration of a security layer [e.g., using transport layer security, (TLS)], but this may result in difficulties when upgrading the security algorithms, as the whole application has to be updated. This fact may shorten the life of IoT devices. As a way to overcome these difficulties, this paper presents IoTsafe, a novel approach relying on secure socket shell (SSH), a feasible alternative to secure communications in IoT applications based on hypertext transfer protocol (HTTP and HTTP/2). In order to illustrate its advantages, a comparison between the traditional approach (HTTP with TLS) and our scheme (HTTP with SSH) is performed over low-power wireless personal area networks (6loWPAN) through 802.15.4 interfaces. The results show that the proposed approach not only provides a more robust and easy-to-update solution, but it also brings an improvement to the overall performance in terms of goodput and energy consumption. Core server stress tests are also presented, and the server performance is also analyzed in terms of RAM consumption and escalation strategies.

INDEX TERMS SSH, IoT, TLS, HTTP, HTTP/2.

I. INTRODUCTION

In the last years, the emergence of Machine to Machine communications has led to predictions talking of billions of devices connected to the Internet of Things (IoT) [1]. The main features traditionally considered when developing new products for this field, i.e. *functionality*, *cost* and *energy consumption*, have been surpassed by *connectivity*, and also by *security*: the future development of the IoT may be compromised if security is not seriously addressed. In fact, in recent surveys carried out by the IEEE IoT Initiative [2], [3], *security* was identified as the first concern between the developers of IoT solutions. Indeed, the lack of security may incur hidden costs that technology and software business have to face.

At present, as the amount of IoT *devices* is increasing exponentially in time, they can easily become a useful tool for hackers to perform massive Distributed Denial of Service (DDoS) attacks on sensible business facilities [4]. This should force these devices to either get patched, disconnected from the Internet or replaced by new ones, due to the impossibility of upgrading them. However, this is unlikely to happen, as their owners may not be noticing any side effect [5]. As a consequence, governments will be forced to legislate in nearby future, as the lack of security in IoT *devices* may represent a global threat [6]. Periodical software patching should be mandatory, as well as the possibility of IoT devices "*upgrading themselves on a regular basis, in a painless manner, without all the fear and loathing that accompanies software upgrades today*" [7].

New disruptive threats as those compromising the supply security chain [8] can be rendered into huge exploits particularly in servers. As an example, Python services can become unsecure since their conception, as developers might inadvertently use compromised libraries, or even include fake

The associate editor coordinating the review of this manuscript and approving it for publication was Gaurav Somani.

software packages into their designs through old techniques such as typo squatting [9].

As a result, it is acknowledged that a major cause of nowadays' most common security flaws lies with user's and programmer's lack of sensitivity towards security best practices [10]. Mitigating this trend can be a difficult path. Therefore, in addition to relieving the user's responsibility over security in IoT devices, it would be advisable to simplify the work of the developers by means of security decoupling. This would reduce software design requirements, leaving the security supply chain out of consideration.

The implementation of preventive measures against security breaches should become a major concern, as TLS security mitigations (or any other) cannot be easily deployed, particularly once elevated privileges are achieved in Linux servers *"since they are at the same privilege level as the kernel, and thus hard to defend against*" [11]. Regarding SSH, "*There are no mitigating factors for this*" either (see section 9.5.1. *End Point Security* of [12]).

Concerning IoT devices security, it is clear that highly constrained hardware has the advantage of its difficulty of being hacked, as optimized hardware cannot be a target of generic automated attacks. However, communication data might still be compromised: a recent publication by the National Institute of Standards and Technology (NIST) [13] states that a product life cycle may be limited by the weakness of its security protocols and algorithms, forcing to upgrade hardware before its deadline or to perform software updating procedures which might be tedious and expensive. This is mainly because hardware is usually deeply optimized to accomplish its original designed functionality, in order to save costs in mass production and to extend batteries' life. However, this fact limits software updating capabilities, forcing end users to acquire new up-to-date products.

In addition, certain regulation trends in Europe such as General Data Protection Regulation (GDPR) [14], require systems providing "*privacy by design*", which inherently requires the "*security by design*" approach stated in the "*Baseline Security Recommendations for IoT*" addressed by the European Union Agency for Network and Information Security [15]. This could make unfeasible the use of highly constrained devices in some scenarios, as they are usually impossible to be upgraded with enough flexibility. In the next section we include a summary of ongoing regulation efforts aimed at providing security to the Internet of Things.

According to different surveys [2], [3], it can be deduced that the development time for IoT devices is shortening as a business policy. The most demanded application protocols are currently HTTP (Hypertext Transfer Protocol) and MQTT (Message Queue Telemetry Transport), as they are long standing and easy to handle. Multipurpose hardware is also the prefered alternative whenever possible, as it is less constrained and allows more flexibility when energy consumption is not the most relevant feature. These pieces of hardware allow to use solutions based on generic POSIX (e.g. Operating Systems fulfilling certain standards compliance for compatibility such as Linux), rather than baremetal, and this alternative is becoming dominant too [2].

However, all these features make it easier for hackers to take control of IoT devices. This can happen if security and updating are not addressed carefully, as multipurpose hardware and software can be used to run arbitrary applications, including malicious ones. Therefore, unkown vulnerabilities also become quickly widespread, and hacking into them in an automated fashion becomes feasible. Although new vulnerabilities in communication and security protocols might be rapidly disclosed, urgent updatings are difficult to deploy, even if they become mandatory. Furthermore, options as replacing entire devices or depending on the final user should be better avoided.

In this context, an approach based on decoupling the security of IoT devices from applications can expedite progress towards the achievement of the following goals:

- Reduced development costs and maintenance during product life cycle.
- Easier "*security-by-design*" implementations.
- Scalability and interoperability in cross-domain IoT environments.
- An easier and inexpensive way to certificate device compliance to security regulation.
- A simpler security market: hackers love complexity.
- A reusable security pattern not limiting the traditional design of vertical solutions: it is possible incorporate IoTsafe into already deployed projects with no changes in IoT devices' software design.
- Reduced human interaction when upgrading software. Neither the interaction with the original software development team (usually no longer available), nor with the final user are required, as upgrades can happen on an automated fashion.

All in all, the contribution of the present paper can be summarized as:

- The proposal of IoTsafe, an approach relying on SSH, able to decouple security from core IoT applications in IoT devices, allowing a set of security improvements in IoT environments based on HTTP.
- In order to illustrate its advantages, a comparison between the traditional approach (HTTP with TLS) and our scheme (HTTP with SSH) is performed over Low-power Wireless Personal Area Networks (6loWPAN) through 802.15.4 interfaces.
- An important element, namely the IoT *Host Server*, is analyzed by means of stress tests related to RAM consumption and escalation strategies.

The remainder of the article is organized as follows: in the next section we summarize the status of the ongoing regulation efforts in Europe and the United States, as we consider it a major contextual factor required to be considered when pondering the problems addressed by IoTsafe. Section III summarizes the Related Work; the Proposed Framework is explained in section IV, and section V details the Test Scenarios. The results concerning IoT *devices* are presented in

section VI and the ones related to IoT *Host Servers* are detailed in section VII. The paper ends with the Conclusions.

II. ONGOING REGULATION EFFORTS FOR IOT SECURITY

Many regulation efforts are in place, encouraging IoT business to seriously address security into their projects, products and services, to overcome previous limitations of current software and devices. This behavior will result into a productivity increase in the mid and long term, both for IoT developers and clients, henceforth transforming security into a major market differentiator. Two examples of this trend are found in the European Union and the United States of America; particularly the former is strongly pushing this paradigm change through regulations, as mere competition in market could not be enough.

In order tackle these challenges, the European Commission has brought forward a proposal of regulation known as the "*Cybersecurity Act*" [16] to establish a European Cybersecurity Certification Framework for Information and Communication Technology (ICT) products and services.[1] The European Cybersecurity organization members on the EU Certification Framework have already presented some ideas [17] that may outline future regulations. In order to provide guidance and help to fulfill future certifications and regulations, the European Union Agency for Network and Information Security (ENISA) has also published a set of baseline security recommendations for IoT [15].

In the United States, new possible regulations are being proposed in the IoT Cybersecurity Improvement Act [18] and would require IoT vendors to certify vital features as the ability of IoT devices to get patched whenever security updates are available.

These certifications will improve surveillance on IoT devices and the use they are intended for. If cybersecurity is not governed, it will add further uncertainty which may prevent users from trusting devices, and investors from including these technologies into their business [17].

These regulations might derivate in liability responsibilities under a cybersecurity breach. It is unclear how extra-contractual and contractual liability could flow from a cybersecurity attack to the software vendor. It is clear such liability between main parties: the attacker (final responsible) and the end user as the victim. Alas, once the software vendor is required to be certified, it will have to perform further actions than the ones addressed today if its device or software was actually put into market with unknown vulnerabilities.

However, in our view, these efforts for enforcing cybersecurity certification are not enough: one problem with static certification of products is that its validity ends as soon as new unpatched vulnerabilities are disclosed. Therefore, certification schemes need to be more flexible [17], and to take into account the management of software patches during a product life cycle.

[1] https://www.enisa.europa.eu/news/enisa-news/towards-the-emerging-eu-framework-on-certification

Thus, it is imperative to search for disruptive approaches that allow to address these issues on a standard way. This would enable the emergence of effective and reliable solutions with low complexity, able to prevent security flaws from being overseen, and rendered into exploits by hackers. It should be taken into account that simpler approaches tend to gather less design vulnerabilities and help to improve security [19].

According to Gartner Inc. [20], security has frequently been implemented mostly in an ad-hoc manner for each vertical project at the business-unit level. Besides, no coordination via a common architecture or a consistent security strategy has usually been defined, thus difficulting reuse and interoperability. However, developing different verticals for every project would skyrocket the security resources required in nearby future.

IoT security components in the industry are only now just starting to be addressed across established IT security standards bodies, consortium organizations and vendor alliances. This leads to a scenario in which, by 2021, regulatory compliance will become the prime influence for IoT security uptake, increasing security budget by 240% since 2016 [20].

All in all, this investment could be further profitable or even cut down if products and software are designed not only to fulfil certification at a certain moment. The inclusion of a procedure for automatic and mandatory updating, based on security decoupling, could be put in place. This would increase the reliability at different levels: software, product, clients and the rest of the Internet. Thus, mandatory updates on security should not affect existing certifications, service contracts, and the functionality on the rest of the software, sensibly reducing software maintenance cost in the mid and long term, and allowing devices to have longer life cycles.

III. RELATED WORK

A. DECOUPLING SECURITY FROM APPLICATIONS

Security decoupling techniques have usually been addressed with complementary specific hardware [21]. This approach barely constrains any design or requirement of main software or hardware. Likewise, it also sidesteps possible main hardware and software limitations, particularly in security features. One example of the use of specific hardware for securing sensible procedures is credit card payment, which can be performed through *Payment Card Industry - Pin Transaction Securit*y (PCI–PTS) approved payment devices [22]. This technology has become the general rule in points of sale where credit card payment is required. Besides, an approved PTS integrated in points of sale facilities is most welcome by businesses required for a GDPR regulation compliance.

The use of a more integrated solution has also been achieved through cryptographic Physical Unclonable Functions (PUF) exploiting each circuit's specific uncopiable and unique hardware features of unpredictable (but repeatable) response [23]. Cryptographic PUF functions require less hardware and energy than traditional algorithms, and their

reliability is based on the unpredictability of some features every single chip has. Decoupling security from applications, using this hardware-based technology, has been proposed as a solution for IoT environments [24] and particularly for firmware update procedures [25].

However, some studies have shown that the major feature of these functions (the inability of being copied) can be overridden under specific circumstances [26]. In many cases, the equipment and tools required for this feat is not worth the hacking success [27], but this fact demonstrates that certain implementations only based on this technology might not be secure enough in all possible scenarios. In this regard, recent research has provided important results in straightforward cloning when XOR PUF functions are used in RFID technology using inexpensive generic hardware and software (and just in milliseconds) [28]. Nevertheless, PUF should be considered a highly promising technology, currently under heavy development, that will overcome such limitations, particularly in conjunction with nanotechnology [29] and machine learning [30].

A network level approach aimed at decoupling security for applications has also been addressed, to a certain extent, through Software Defined Network's (SDN) configurations. A reactive and coordinated response to certain attacks can be automated, providing a successful response when mitigating them in IoT environments, and protecting other devices before imminent attacks may happen [31].

Decoupling security and applications using just software is also feasible, but it requires the use of powerful enough IoT devices so as to handle a non bare-metal solution: almost any device able to run an OS suffices, even the most constrained ones. This solution should be able to run stand-alone complimentary security software/firmware. Alas, this would irremediably require less constrained devices with more hardware features, mainly RAM, ROM, and also demanding more energy.

IoT middleware solutions [32] offer hardware abstraction in order to increase interoperability and improve security [33]. These approaches help to decouple security from software design [34]. However, as the abstraction layer is embedded into the IoT device, it eventually holds together (in common/related binaries) the implementations and configurations of the security protocols (generally based on TLS/DTLS). Therefore, a security upgrade also requires a middleware update so as to incorporate these changes into the embedded IoT device layer. A feature that could be desirable for a better security decoupling solution should be the ability of upgrading them without requiring an update in IoT middleware.

Such security decoupling approach is possible in these kind of middlewares, if device abstraction software and Core IoT application are embedded in different binaries (Fig. 1). In this case, a *successful* security decoupling can be achieved whenever an upgrade in the middleware platform concerning security only applies to device abstraction binaries, and does not change the internal API that the Core IoT application requires to communicate through the device abstraction layer. Anyhow, a security decoupling for the IoT middleware would still be advisory in order to reduce patching times and to simplify the middleware architecture. Otherwise, the security decoupling problem would have just been moved towards middleware software implementation.

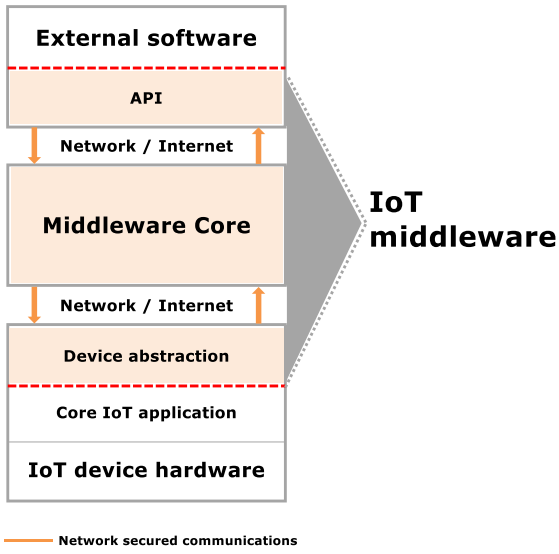

FIGURE 1. IoT middleware structure.

The hardware of IoT *devices* sets a limitation for the software they can run. In many cases, microcontrollers have been widely used as hardware platforms when developing IoT *devices*, due to their low power consumption and reduced price. Additionally, new protocols especially tailored for IoT traffic, able to run in constrained nodes and networks, have been developed, as e.g. the Constrained Application Protocol (CoAP) [35], a REpresentational State Transfer (REST) protocol to be used in low-power and loss-prone environments. Due, the restricted capabilities of these microcontrollers may lead to insecure implementations [36], and to the impossibility of future and remote updating [37]. This may be the case when the changes needed in core cryptographic functions involve different mathematical operators: for example, recent vulnerabilities discovered on the standard Diffie-Hellman key exchange algorithm require major changes in the transition to elliptic curves [38].

However, not all IoT pieces of hardware are so limited in terms of processing power. In fact, many of them do have processors capable of advanced computations, as e.g. System on a Chip (SoC) devices [39]. Therefore, in a significant number of cases, more mature security solutions can be employed where decoupling security is feasible, as the use of these emerging hardware architectures with higher capabilities is appropriate, whilst energy consumption and pricing can be maintained in reasonable levels.

The scheme proposed in this document represents a contribution to resolve the security problems of IoT *devices* with a certain level of processing capacity (e.g. SoC devices).

In these cases, those potential vulnerabilities could be resolved by just upgrading this modular security software remotely.

Some examples of companies developing SoC/SoM devices where this approach is feasible are Emcraft, Pengutronix and Beck-ipc, among many others. Some relevant technical IoT examples where this scheme could be implemented are: a) ARM9 Linux-based embedded devices for industrial purposes [40]; b) FPGAs hosting small IoT Linux-based platforms [41]; c) approaches based on embedded Linux OS in wireless microcontroller chips [42].

B. LIMITATIONS OF TLS/DTLS

Software security solutions are frequently provided by means of TLS/DTLS (Transport Layer Security / Datagram Transport Layer Security) [43], but these techniques alone may not be so practical for the kind of constrained environments addressed in IoT projects. For example, TLS may easily suffer from man-in-the-middle attacks if certificate authorities are not properly validated by the IoT *devices* [44], [45], thus requiring some sort of complimentary authentication procedures [46].

Besides, TLS has to be linked to IoT core applications' binaries, leading to difficult security updates as it is shown in Fig. 2. In this context, SSH may represent a valid alternative to TLS in scenarios using TCP and higher layer protocols like HTTP. It should be taken into account that mature and largely tested implementations of SSH (e.g. OpenSSH, DropBear) are available. Additionally, man-in-the-middle attacks can be easily mitigated[2] in SSH through public/private key authentication (different for each device) without requiring further development.

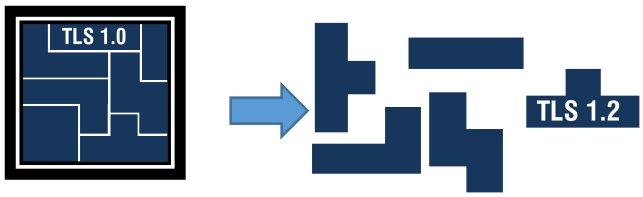

FIGURE 2. Upgrading TLS on an IoT Core Application of an IoT device.

The processing requirements of SSH can be comparable to those of TLS when running key exchange algorithms, ciphers and message authentication codes, as they share many of these algorithms. But the main advantage of using SSH is the simplification in the upgrading procedures, particularly when a security flaw is detected or a feature becomes obsolete or unsecure, as SSH implementations can work standalone in POSIX-like environments. As shown in Fig. 3, in these cases SSH is not a part of the firmware or the core IoT application itself, but another software element that can be updated without modifying the core IoT application. This can be considered as a major advantage with respect to standard

[2]https://www.ssh.com/attack/man-in-the-middle

FIGURE 3. Upgrading IoTsafe on an IoT device.

TLS deployment, which is usually deeply integrated within the core IoT application or its firmware.

The fact of TLS being part of the core IoT application itself slows down the upgrading process of devices, exposing vulnerabilities during moderate periods of time, as highly constrained devices usually require manually flashing the new firmware, or even renewing the hardware. As an example, TLS prior to version 1.1 was already proven to be insecure [38] and therefore all Android devices prior to version 5 are unable to use TLS 1.2 unless a full upgrade of the operating system is performed [47]. Anyhow, current Android applications are able to use third-party security libraries to overcome this problem. However, insecurity persists in time once a vulnerability is discovered. As an example, the use of RC4 ciphers in TLS Android communications apps took nearly a year to drop its usage to 50% after that cipher was announced vulnerable [48].

Therefore, protocols and network infrastructure should provide the required features to fulfill previous requirements and mostly in an expedited and automated fashion on whichever platform, IoT *device* technology or software version. In this context, the novelty of the proposed framework is to use SSH as a valid alternative to TLS for protocols such as HTTP and HTTP/2 to transparently establish a secure bidirectional communication between IoT *devices* and platforms, acting as a complementary independent security interface from IoT software architecture. This includes:

- The proposal of a proxy-like scheme called IoTsafe, which decouples security from core IoT applications, making security independent from the firmware libraries and IoT software. It is based on stand-alone SSH protocol embodiments instead of integrating TLS into the IoT software.
- This approach allows TCP-based application protocols such as HTTP and HTTP/2 to work securely with new low-powered interfaces, even with high MTU and data transfer rate constraints.
- The proposed solution satisfies the security requirements of *portability* and *upgradeability*. The results of different tests are presented in next sections so as to support this statement.

IV. PROPOSED FRAMEWORK: IOTSAFE

In this section we explain in detail the proposed framework for IoT security. We first include a subsection (*A*) in which the concept of *socket proxification* is presented. The next one (*B*) includes a detailed explaining of our proposal; then,

the included protocols and tools are specified *(C)*, and the characteristics of the system are explained *(D)*.

A. SOCKET PROXIFICATION

The proposed security scheme is intended to be modular and stand-alone, relying on the benefits provided by both Linux-based operating systems and SSH protocol. Its main goal is to grant the highest security level, in the simplest way and transparently.

To achieve these features, its communication scheme is based on proxifying sockets in operating systems: proxification consists on forwarding the traffic involving one socket to another socket that can be local or placed in a remote machine. In SSH literature this is usually referred as *port forwarding* (see Section 7 of RFC 4254) [49].

In this paper, we will refer to the new socket resulting from this action as a "*proxification of the original socket*" as, for all intents and purposes, it acts in lieu of the original socket. Proxifications can benefit from privilege separation and network packet features as internal *marking* and *filtering*. POSIX-like operating systems and many Linux families include these features by default, and they have been deeply implemented and used all over the years.

B. DETAILED DESCRIPTION OF THE PROPOSED METHOD

The families of IoT *devices* addressed in this article (those using TCP and HTTP) are usually based on trustworthy low-powered hardware that works on a robust but simple operating system/firmware, able to run a single application based on HTML5 content.

This application may ideally run on a lightweight web server with server-side scripting, as e.g. Golang, Node.js, Python or PHP. It may also serve as a front-end interface, with touch capabilities if needed.

FIGURE 4. IoT communication scheme using TLS.

Fig. 4 shows the traditional approach, based on TLS libraries. In this case, the security features are set during the software development stage, and are included into the binary code during the linking process. The resulting binary is highly efficient, but limited in the possibility of upgrading core security features or configurations as, most of the times, it would be necessary to recompile the whole code after modifying the specific parts implementing the new security features.

In contrast, Fig. 5 illustrates IoTsafe, our proposed scheme using SSH, where the security layer is provided by a proxy entity. It works transparently for the IoT application, making

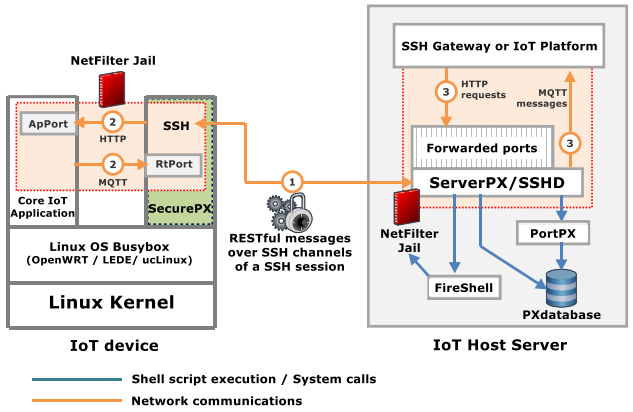

FIGURE 5. IoTsafe communication scheme: IoT devices using protocols such as HTTP or MQTT through SSH secured channels.

the IoT software independent from the employed security technology and its configuration.

IoTsafe provides two pieces of software: *SecurePX*, installed in the IoT *device*, and *ServerPX*, which consists of a set of software scripts, binaries and configurations installed in the IoT *Host Server*. In this scenario, the IoT application should run inside a "*network sandbox*" provided by the Linux kernel NetFilter framework,[3] by means of local IPv4/IPv6 policies, such as forbidding all incoming/outgoing traffic except for responses to specific requests.

IoT *devices* can establish SSH communications with an IoT *Host Server* through *SecurePX*, therefore *proxifying* IoT communications. As shell access is never used, this scheme only relies on SSH forwarding features. This IoT *Host Server* can act as an IoT *Gateway* or as a host for an IoT *Platform* to provide end user services.

All the information requested by the IoT *Host Server* is exchanged through those SSH connections previously established and thus, the communications between the IoT *Platform* and the *Core* IoT *Application* become transparent and secured through SSH bidirectional channels. These communications are handled by ServerPX at the IoT H*ost Server*, which grants transparent access to local forwarded sockets exploiting Linux user system structure and privilege separation features for security purposes.

PortPX and *Fireshell* are two pieces of software belonging to *ServerPX*, which handle specific aspects of communication: *PortPX* assigns and registers the sockets forwarded to IoT *devices*, and also monitors incoming communications, whilst *FireShell* is in charge of handling those incoming/outgoing communications, allowed or denied upon configurations and privileges stored in *PXdatabase*.

Following this scheme, the IoT Platforms do not have to handle any of these security concerns, and may be designed and programmed assuming all the IoT *devices* were accessible locally and securely. Besides, neither the SSH server nor the client require any modification, because standard SSH

[3] http://www.netfilter.org

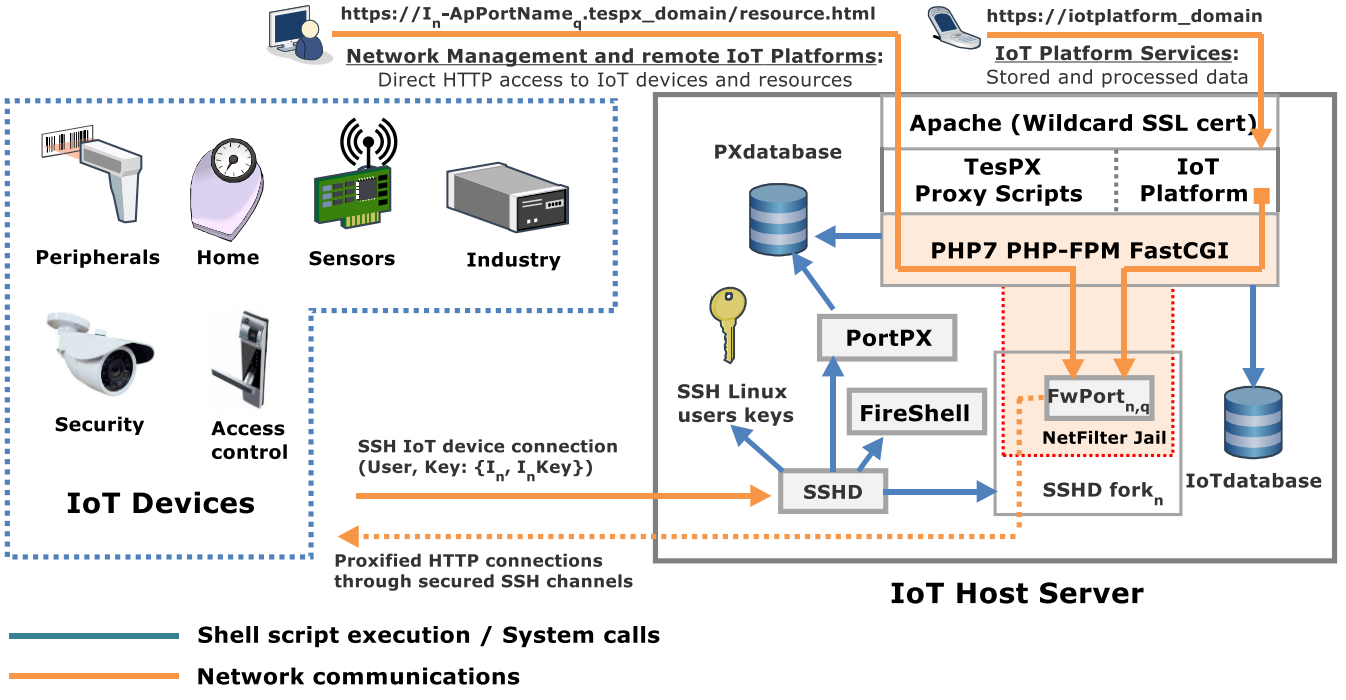

FIGURE 6. Conceptual diagram of an IoT infrastructure based on HTTP and secured with SSH trough IoTsafe.

embodiments are usually flexible enough so as to provide and also to limit the scope of all the features required for this scheme. Another advantage is that several communication channels (proxified sockets) can be multiplexed and compressed into a single SSH session. Furthermore, if bursty communications are avoided [50], as it usually happens in IoT environments, the default OpenSSH code will maintain latency in low levels.

The IoT *Host Server* should also be able to allow all incoming remote communications from web browsers or specific applications mostly used by the staff in charge of monitoring and surveillance of the IoT infrastructure, as detailed in Fig. 6.

In these cases, a direct bidirectional connection can be established with any IoT *device* using TLS: this is allowed when communications established from external applications are addressed through the IoT *Host Server*. All these incoming connections are secured using a dynamic hierarchical subdomain structure, together with a wildcard SSL certificate.

The IoT *device* numbers (In) and the IoT Application Port Name *(ApPortNameq)* are mapped through dynamic subdomains from each request to the IoT *device* addressee. This allows a remote client to easily access any IoT resource, if the necessary privileges are available. These virtual subdomains allow the redirection of any HTTPS request through *ServerPX* scripts, as all requests can be protected by a single wildcard certificate. Every HTTP request is parsed and redirected into the *TesPX* proxy scripts by means of generic Apache rewriting policies, transparent to the HTTP exchange. *TesPX* acts as a transparent proxy entity towards local sockets (IoT forwarded resources called *FwPorts*) allowing transparent communication to IoT *devices*.

A local IoT Platform, hosted in the IoT *Host Server* itself, can also connect directly to all those *FwPorts*, but it must first authenticate itself against *TesPX* to gain access to those local sockets. All *FwPorts* are protected through FireShell and NetFilter, and also by the Linux user structure and privileges. In order to connect to any of these local sockets, access must be first granted to the local Linux user of the application by means of system calls or HTTP requests performed towards *TesPX*.

C. INCLUDED TOOLS AND PROTOCOLS

It can be observed that the whole scheme is based on well-known technologies and deeply tested and maintained software as the Linux Kernel, NetFilter framework, Apache web server, PHP engine, MySQL/Maria DB and SSH protocol, all of them common in most server deployments nowadays.

Installing and configuring *ServerPX* into an IoT *Host Server* does not require patching any default software, and it is designed to be able to securely coexist with other existing Linux services. This scheme can also be implemented into other technologies such as NGINX, Node.js or GoLang, improving its performance when several local interfaces have to be used to handle a large number of IoT *devices* and connections, overriding port limitation range.

It should be noted that this modular scheme is based on proxification rather than on tunneling. For a tunnel to be established, there must be a full encapsulation of incoming traffic. In contrast, the proposed scheme uses proxification and security contexts, providing end to end secure communication between Core IoT Application and IoT *Platform* by means of three communication segments:

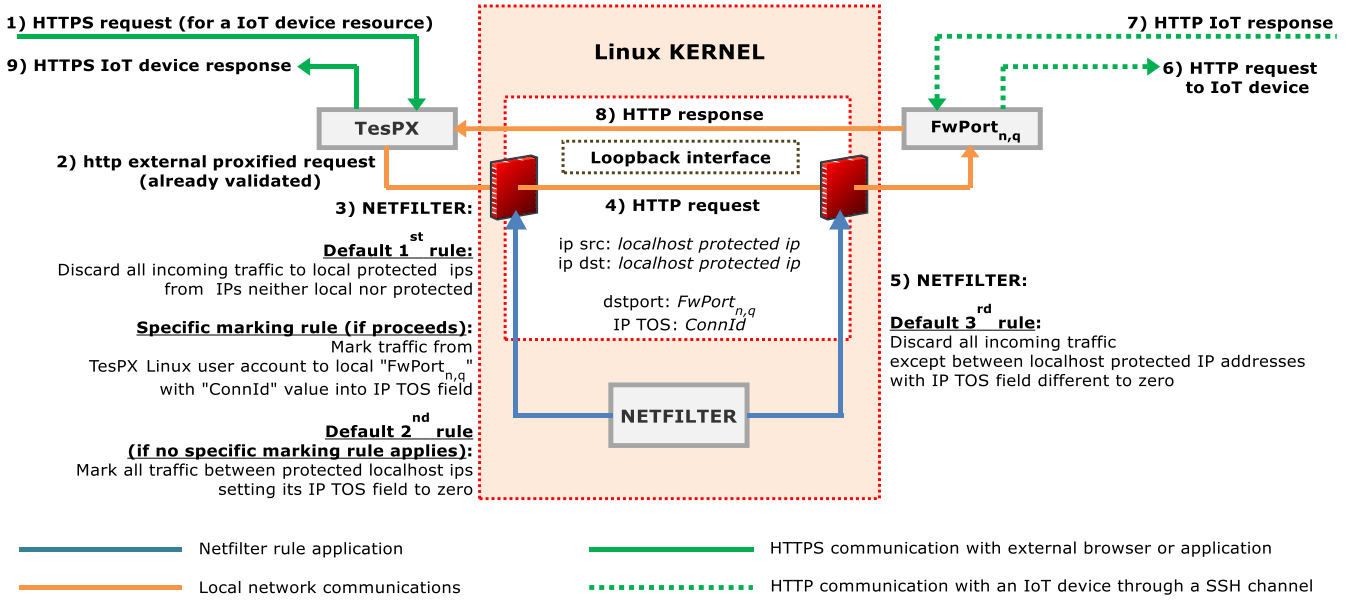

FIGURE 7. Example of NetFilter rules usage in an external transparent access to an IoT device resource.

1) The proxification itself in the SSH session, in charge of securing communications between IoT *devices* and the IoT *Host Server*. It is secured through the proxifying application and its secure protocol. In Fig. 5, this segment corresponds to the SSH channel in the SSH session between *SecurePX* and *ServerPX* and represented with "1".
2) The second one communicates the core IoT application and *SecurePX*. In Fig. 5 it is represented with "2". Please be noted there are two of them because as two example communications presented (HTTP and MQTT).
3) The last one covers the segment between the other end of the proxification (in *ServerPX*) and the IoT *Platform*. In Fig. 5 it is represented with a "3"

These last two segments (2 and 3) are secured by means of security contexts defined by filtering and marking IP packets rules, allowing the scheme to control access and outgoing traffic from proxifications. In the case of an IoT *device*, the security context may consist of the entire device itself if there is no privilege separation. Regarding the IoT *Host Server*, each security context comprises an operating system user account and a destination local socket. In order to elaborate on the communication scheme proposed, specific key concepts are next presented in detail.

1) PROTECTED SOCKETS

In this security scheme, in an IoT *Host Server,* a protected socket is considered as such when its IP address belongs to a set of local ones reserved to this security scheme. All these local IP addresses are handled and protected by default with several filtering/marking rules. In Linux systems, these rules are applied by NetFilter kernel framework and configured by system calls from a privileged user. Default rules defined in the proposed scheme state three general policies as depicted in Fig. 7:

- *Default rule 1*. (Type: FILTERING): All incoming packets from external IPs and all outgoing packets to external IPs should be discarded if either the source or destination of the communication comprises a protected local IP.
- *Default rule 2*. (Type: MARKING): All local traffic should have its TOS (Type Of Service) field reset to zero when both origin and destiny comprises a local IP belonging to the protected local IP addresses.
- *Default rule 3*. (Type: FILTERING): All local traffic should be discarded if both origin and destiny comprise a local IP address belonging to the protected IP address pool but TOS field is set to zero.

Filtering and marking packets happens at different moments of the communication, particularly when packets enter or leave the Linux kernel space. This allows marking packets (from applications), being unable to override the third default rule, but only with another specific marking rule that overrides the second default rule. NetFilter processes rules for each packet linearly in a special way, as it may only apply a single rule for each packet in a given chain of rules. Thus, should a marking rule apply to a packet, no more rules will be applied to that specific packet as it would be already marked. This same behavior applies to filtering policies likewise. Based on this principle, specific marking rules may be defined to override the second general rule and thus, to allow communication at will. As NetFilter provides a stateful firewall service, this marking rule will also apply to responses unless something else is specified, allowing a bidirectional communication.

In the unlikely case a local program might try to mark packets by itself, it would not succeed in this attempt of auto-granting access to a protected socket, as marking rules cannot be overridden that way. Its marking will not be considered and discarded by the second default rule.

These policies may apply for both the IoT *Host Server* and the IoT *device*. However, if the Linux implementation of the IoT *device* does not provide privilege separation (as it may happen in certain low-cost devices with highly constrained Linux embedded systems), it should be dealt as a single user/application device and the second and third rule would not apply.

2) PERMISSIONS

In this security scheme, user and group permissions are used. The first ones allow processes from a Linux system to access a protected socket. The second ones also grant the same access but to an entire group of users. Each permission comprises the security socket whose access is to be granted and either the *UserID* or *GroupID* whose process are to be granted access.

3) SECURITY CONTEXTS

This concept consists of a set of marking and filtering rules for network traffic that help the operating system to define and identify traffic flows in an automated and simple fashion, allowing to extend security from the proxification to the source or destination of the communication.

This security context does not require extra overhead in communications and it is only applicable to local traffic. A set of rules required for a security context to be defined comprises the general ones needed to secure local sockets and an extra specific priority marking rule defined to override the second default rule.

The objective of this security scheme is to enable traffic filtering according to permissions. Permissions thereby comprise user accounts (which map IoT *devices*, groups of devices and also IoT *Platforms* or other entities) and destination sockets. However, filtering policies would only allow to take into consideration traffic features. Thus, when access is granted to a user for a socket, a *connection_ID* is defined and thereby marked into the TOS field, to be able to pass through the filtering of NetFilter. Hence by, a security context is defined, wherein any process from the permission's user account is able to freely communicate towards the protected socket whose access has been granted.

These *connection_IDs* are assigned to traffic from processes whose users are allowed to access the specific protected socket thanks to a user or a group permission. When access is granted, a *connection_ID* is generated and also its marking rule is put into action to mark traffic *from* this Linux user account *to* that specific local protected socket. However, as the IPv4 field used for this security scheme is TOS, only 255 valid *connection_IDs* can be generated for each protected socket.

Local IoT protected sockets would usually be set for proxifications of core IoT application sockets (*ApPort*) from the IoT *devices*. These applications are not meant to allow huge numbers of simultaneous connections. Usually, only the Linux user account of the IoT *Platform* is the one which connects to that kind of socket. However, if the Core IoT Application requires to connect to a remote socket in the IoT *Host Server (Port_A)* through a reverse proxification *(RtPort)*, this *Port_A* should have to accept the requests from all IoT d*evices* and thus, the number of accesses granted would easily surpass 255. On these scenarios, a group permission should be used to arrange all similar IoT *devices* with common features into a single group and grant their Linux user accounts at IoT *Host Server* an access to *Port_A* through a permission granted to their Linux *Group_ID*. Thereby, a security context is defined by the *User_ID* or *Group ID* of the operating system and the *connection_ID* assigned.

When the security contexts comprise a *User_ID* and a *connection_ID,* this is mostly intended to grant access to a specific proxification of an IoT *device*, and thus it is referred as a *strict security context*. Conversely, if the security context is defined by a *Group_ID* instead, it is referred as a *wide security context*, as the related marking rule applies to all Linux users' traffic to the local protected socket to which the context refers.

4) PROXIFIED COMMUNICATION

A process can request to communicate with another process through a local proxified communication involving protected sockets. This can only be done once a resolution of the proxified socket has been addressed. The process may know the ID of the device and the alias of the socket it needs to connect to, but the value of the proxified socket is yet unknown. Hence, it should be resolved through a resolution query against *Pxdatabase*.

5) RESOLUTION QUERY

A resolution query has to be performed before any proxified communication can start. Through this query, a process aiming to communicate to a proxified socket can obtain the value of the local proxification that leads to the remote process socket. The resolution query is addressed to a local software module called *ServerPX*. This module checks the user Linux account from which this resolution request was issued and verifies whether it has permission. Once this permission is validated, the user account is granted an access permission as a *connection_ID* that it is translated into a marking rule. This rule applies only when communicating with that protected socket, and allows the process to override general marking rule 2 in that specific case.

6) KEY MANAGEMENT

A key management protocol does not have to be defined in this scheme, as local processes do not require keys to communicate with IoT *devices*. Access to proxified ports is gained through permissions and its handling is performed through NetFilter rules. However, it is required as a part of the upgrading procedure of IoTsafe.

7) CONFIGURATION PROFILES

There are two kinds of configuration profiles in the proposed security scheme. An *IoTProfile*, which consists of the credentials and configurations that an SSH client at an *IoT device* should use to connect successfully to an IoT *Host Server*. This *IoTProfile* consists of some configuration options such as the required proxifying sockets, both direct and reverse. Those options are to be validated at IoT *Host Server* and cannot be overridden. Connection limits specified for those parameters include:

- Maximum number of concurrent connections per IoT *device*.
- Maximum number of proxifications (reverse or direct).

In case an IoT *device* is compromised or hacked, it should not be able to perform any kind of hacking attempt to the IoT *Host Server* or other IoT *devices*. This level of security is provided through to the versatility of current SSH implementations, single key using for each IoT *device*, separation privileges of Linux operating system families and filtering rules of NetFilter. This network framework can log specific hacking attempts derived from rule matching and enable the network manager and third party applications to take action disabling henceforth compromised keys. *HostProfile* is another configuration profile which defines specific aspects of SSH server configurations and supporting applications. These include the reserved local IP addresses pool, and the privileged local Linux account configurations in which IoTsafe binaries and scripts are run.

D. CHARACTERISTICS OF THE SYSTEM

After the detailed description, in this subsection we summarize and discuss the main characteristics of the system.

1) SECURE BY DESIGN

The proposed scheme considers several scenarios where possible vulnerabilities might appear. As stated before, its security mainly relies upon the SSH protocol and privilege separation of Linux operating systems. Thus, vulnerabilities may arise mostly for escalation privileges, denial of services, and SSH key compromising.

Despite all these potential dangers, SSH has been safely and successfully deployed in all sort of devices and servers over the years. It is however plain that servers particularly require secure and reliable Linux implementations to work *securely*. To achieve a better grade of security in Linux servers and mitigation procedures to endure potential security breaches, Linux kernel provides an API called Linux Security Modules (LSM), intended to enhance security (double) protecting sensible operations carried out at kernel level.

The LSM API can be used by a defined and configured computer security model as AppArmor[4] or SELinux.[5] Any of these approaches contribute to increase the security of the system helping to address possible unknown threats related to poor security planning, configurations or overseen flaws in the system that could be rendered into exploits [51]. These complimentary features are considered standard in production environments, particularly to prevent root privilege escalation.

It is however clear that, no matter whichever security technology is used (particularly in server deployments), once an IoT *Host Server* able to address automated updates towards IoT devices is compromised to root level, it is only a matter of time that the whole IoT network it handles gets compromised likewise.

Keeping IoTsafe scheme this simple might not clearly present a barrier for hackers achieving this, but it actually makes the process of hacking the server a more challenging task: IoTsafe minimalist design relies solely on standard and well-known security technology, being straightforward to implement a *security-by-design* scheme, and in case of security breach, much easier to detect and mitigate it using standard procedures.

All the actors involved in IoT communications through an IoTsafe implementation are shown in Fig. 8, and how they interact through IoTsafe. The security provided by IoTsafe is present in different parts of the whole communication process and it is gained through standard Linux features identified in the figure with letters *a, b, c* and *d* featuring the following:

a) IoT device protection: IoT device communications are meant to be addressed through an IoTsafe encrypted session with the IoT *Host server*. Any other communication is discarded.
b) Communications between IoT *devices* and IoT *Host servers* are encrypted through SSH sessions.
c) All local communications addressed in the IoT *Host server* are managed (and thereby protected) by Netfilter, whose rules involving IoT traffic are defined by IoTsafe software and security contexts.
d) A strict surveillance over Linux accounts and privileges is handled by a correctly configured and deployed

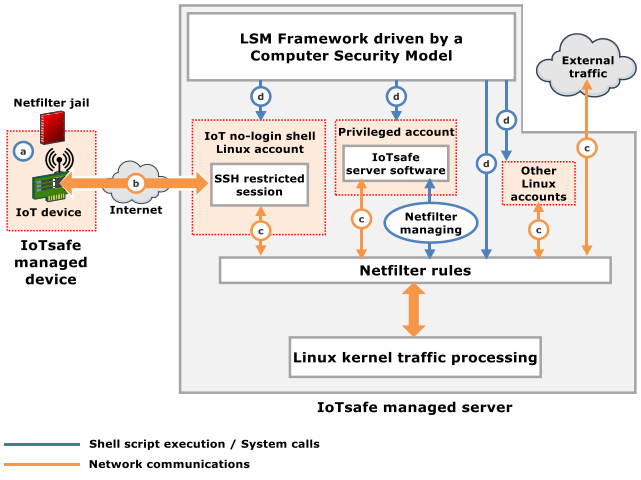

FIGURE 8. Example of NetFilter rules usage in an external transparent access to an IoT device resource.

[4]https://gitlab.com/apparmor/apparmor/wikis/home/
[5]http://selinuxproject.org/page/Main_Page

Computer Security Model such as SELinux or AppArmor. These features are supported by the Linux Security Modules (LSM) framework at kernel level since 2.5 version. A *Computer Security Model* is also able to control Netfilter and can work complimentary with IoTsafe to detect unexpected behaviors from IoT accounts. This can be particularly achieved monitoring Netfilter logs.

It is important to note that SELinux and AppArmor are considered standard tools already enabled by default in certain Linux server deployments (CentOS/Redhat and Ubuntu) and can be considered also well-known features by Linux systems administrators or security engineers. Thereby, a suitable security configuration required by IoTsafe to work can easily be deployed.

In the event an IoT server is comprised to root level through an unknown vulnerability, the options for mitigation are slim as it was previously stated. However, a true diversification effort into the network would provide real and measurable resilience: a positive *K-Zero Day Safety* metric [52]. A physical diversification effort is achieved managing all IoT *devices* comprising the IoT network through small *Gateways*, which gather communications towards IoT *Host Servers*. Nevertheless, a *true* diversification able to contain the threat (even in the IoT *Host Server*) would further require suitable account permissions, SSH configurations and keys management (example 5.1 [52]) to be set up into *Gateways* and IoT *Host Servers*.

2) KEY COMPROMISING

In order to allow a wider flexibility in its deployment, IoTsafe does not suggest any specific algorithm to handle SSH keys. Nevertheless, key management is required less frequently than usual, as neither users nor processes have to access to SSH keys: their connections are handled through permissions and security contexts and this prevents keys from being misused.

Anyhow, in case an IoT *device* is compromised, the SSH server would only allow a use of the key according the configuration profile corresponding to that specific device, and only whence it works *similarly* before being comprised it would be unnoticed. Thereby, any comprised device and key would be formally detected and tainted whenever it is detected an attempt to:

- Proxify more connections than allowed.
- Proxify connections using different protected sockets.
- Proxify connections to access other IoT Server socket or remote sockets.

In all these cases, the requests would either way be denied by NetFilter rules previously explained. If the scheme embodiment considers it appropriate, NetFilter logs could be processed so as to identify these infractions, making it able to take the corresponding measures: disabling IoT keys from those devices working strangely as they are subject of having being compromised.

In the event of compromised keys being used to generate multiple parallel connections, they would also be ignored and discarded according to the *IoTProfile* which IoT *Host Server* also validates.

3) TRANSPARENT COMMUNICATION

In most cases, IoT *devices*' communications are managed by the IoT *Platform* but in some specific scenarios a final application or a final user might require to *directly communicate* with one of these IoT *devices* e.g. to activate a relay, to trigger an asynchronous sensors' measure, to connect to a video/audio secure camera feed, etc. In these cases, the IoT *devices* are required to be accessed by final users *transparently*.

This communication is accomplished using an inner proxy as a middleware that transparently translates the secure version of the IoT final user application protocol (HTTP Secure, HTTPS) to the unsecure version of that protocol used by the Core IoT Application (HTTP). This inner proxy authenticates the user credentials against *ServerPX* or IoT *Platform* records in database.

In this second case, the inner proxy should instance itself on the IoT *Platform* account to have suitable permissions granted and be able to access to the required protected proxified socket. Such proxy can be programmed in multiple languages. The proxy belonging to *TesPX* scripts (shown in Fig 5) can be easily deployed using a PHP client (e.g. Guzzle[6]).

V. TEST SCENARIOS

This section includes the description of the test environments that have been created in order to measure and validate the presented proposal. SSH performance is studied in an IoT *Host Server* using a *Linux operating system*. These are the two core components in any deployment of IoTsafe and they are thus analyzed in this paper. The other modules, databases and scripts can be implemented in multiple languages and their required level of optimization depends of the scope of the project they are intended to address. Therefore, the performance of these modules will not be studied here.

Two different scenarios are presented. The first one is designed to measure SSH performance in IoT *devices* and to compare it with the use of TLS in HTTPS. This is the scenario proposed to obtain the results presented in Section VI. The second one focuses in *Host Servers* and aims to validate the scalability and the RAM resources required when a high number of connections or channels are to be handled (the results will be presented in Section VII).

A. SCENARIO FOR COMPARING SSH WITH TLS IN IOT DEVICES

The first test scenario, depicted in Fig. 9, comprises two IoT *devices* able to support HTTP, TLS and SSH. Thus, we have used two Raspberry Pi equipped with 802.15.4 Openlabs

[6]http://docs.guzzlephp.org/en/stable/

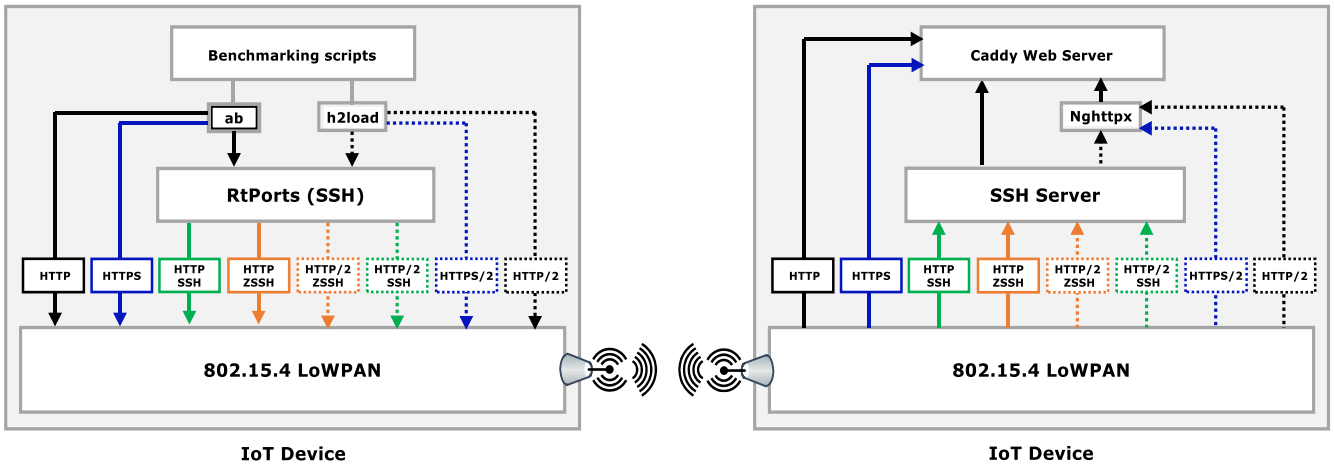

FIGURE 9. IoT *device* test diagram to compare SSH vs. TLS performance in HTTP communications.

interfaces (based on the at86rf23 Amtel transceiver). Communications are established between the two Raspberries over WPAN: the first one acts as an IoT *device*, and the second one as an IoT Gateway/Relay. The transmission is conducted on channel 13 (2415 MHz), and the virtual LoWPAN interface is configured on both devices to fragment packets when needed: IPv6 requires a minimum MTU of 1280 bytes, whilst the 802.15.4's MTU is 127 bytes (working with LoWPAN fragmentation, it only offers 96 bytes of payload). Linux kernel 4.7.4 is used, which includes a stable IEEE 802.15.4 LoWPAN implementation. The transfer rate of this device is set to 250 kbps, i.e. the maximum specified in the 802.15.4 standard. The TLS vs SSH comparison will be performed using HTTP versions 1.1 and 2: we compare HTTPS (HTTP using TLS) vs. HTTP tunneled through a secured SSH channel. Similar algorithms, able to provide an acceptable degree of security, have been selected in both cases.

HTTPS is provided by Caddy, a portable web server [53] which also supports TLS 1.2. During the tests, we use the 'TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384' algorithm, one of those recommended B CLASS in the National Security Agency (NSA) suite for TLS [54]. This suite implements Elliptic-curve Diffie-Hellman Ephemeral key exchange, using the Elliptic-curve Digital Signature Algorithm (ECDSA), with AES-256 as the block cipher, and SHA-384 for the hash message authentication code (HMAC).

SSH is provided by OpenSSH, configured to work with the same algorithms to establish connection: it uses curve25519-sha256 as an ECDSA key exchange algorithm, aes256-gcm as block cypher and hmac-512 as HMAC (this last algorithm is stronger than the one used in TLS because hmac-sha384 is no longer available as an HMAC algorithm in current OpenSSH default implementations). SSH compression is provided by ZLIB library [55] and can be activated by default editing the ssh_config file.

B. SCENARIO FOR CHARACTERIZING THE PERFORMANCE OF THE IOT SERVER

The second scenario is designed to characterize the major limitations of SSH in IoT *Host Servers* and the penalty in performance and resources consumption the system suffers according the number of established proxifications, active proxifications and SSH embodiment.

The testbed for this second scenario is depicted in Fig. 10 and comprises two IoT *Host Servers* able to run multiple communications in parallel. They are virtual servers based on Kernel Virtual Machine (KVM)[7] provided by Linode cloud platform[8]. Each virtual machine is a Linode 16 GB, comprising Centos 7 operating system, 16 GB of RAM memory, 6 virtual cores Intel(R) Xeon(R) CPU E5-2680 v3 2.50 GHz (single thread, single core each) and a local average connection of 1 Gbps between them. The first server *(Server_A)* is used for running a number of SSH clients, and the second server *(Server_B)* generates multiple SSH proxifications and sessions to model incoming communications into *Server_A*.

If *Server_B* requires to relay proxifications to another server, it would actually play the role of a *Server_A* generating all the required instances of SSH clients to forward those proxifications. Thereby, the testbed proposed constantly measures the worst-case of the two scenarios (a server receiving multiple IoT proxifications, but also a server relaying multiple proxifications to another server). This testbed should be considered legit, as a full-fledged IoT *Host Server* should be able to relay those proxifications to third party servers too.

Other variables are taken into consideration such as active/established proxifications and established SSH sessions to clarify its dependence and the relation between

[7]https://www.linux-kvm.org/page/Main_Page
[8]https://www.linode.com/

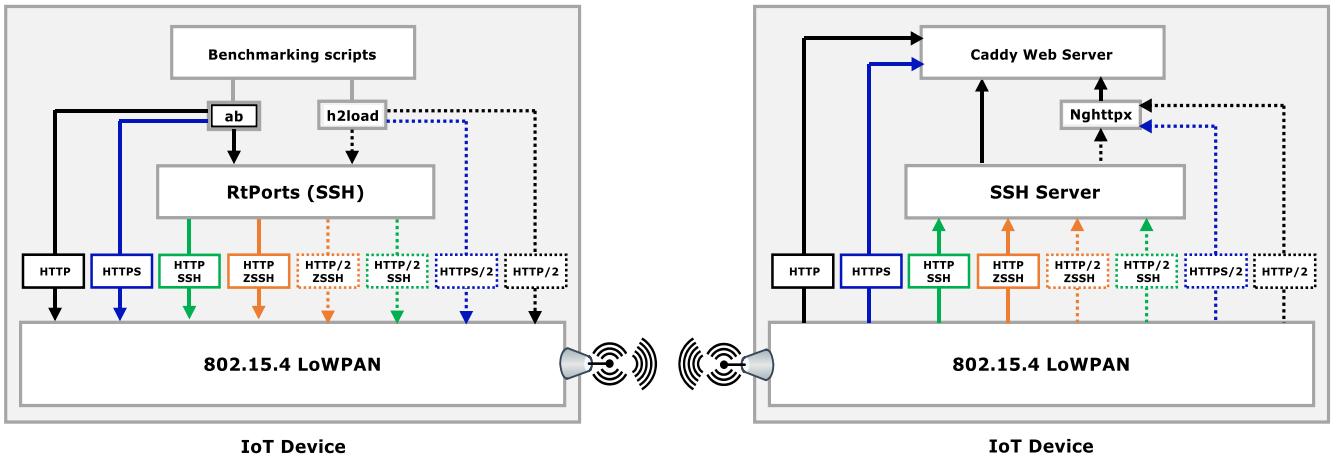

FIGURE 10. IoT *device* test diagram to compare SSH vs. TLS performance in HTTP communications.

them all to suggest an optimized combination to maximize performance.

VI. IOT DEVICE RESULTS

In this section we present the results obtained with the first test scenario, comprising two IoT *devices*. The main parameters to be measured are the achieved throughput, and also execution times, so as to compute and compare performance and efficiency. Herein, four scenarios are compared for both HTTP and HTTP/2:

a) *HTTP:* plain HTTP connections without security;
b) *HTTPS:* HTTP with security provided by TLS1.2;
c) *HTTP SSH:* HTTP secured over SSH; and
d) *HTTP ZSSH:* HTTP secured over SSH with payload compression by ZLIB.

The tests consist of performing HTTP *GET* requests of text files of different sizes, ranging from 4 bytes to 262 Kbytes, simulating shorter and longer connections. This is useful in order to compare different scenarios: short communications are common in querying specific sensor data, whereas longer ones arise in certain telemetry and firmware upgrading scenarios. For each test, 100 files of the same size but random content are sequentially transferred, and the average results are calculated and presented. Note that all the figures in this section use non-linear scales on both axes.

For HTTP, *ab*[9] benchmarking tool is employed, whilst *hdload*[10] benchmark tool is used for HTTP/2. Caddy server is proxied through *nghttpx*[11] to provide support for an HTTP/2 connection through h2c upgrade token and HTTPS/2 connection through h2 upgrade token.[12] Caddy server does provide native support for HTTPS/2 but not for just HTTP/2 and thus *nghttpx* tool is required.

[9]http://httpd.apache.org/docs/2.2/programs/ab.htm
[10]https://nghttp2.org/documentation/h2load-howto.html
[11]https://nghttp2.org/documentation/nghttpx-howto.html
[12]https://tools.ietf.org/html/rfc7540

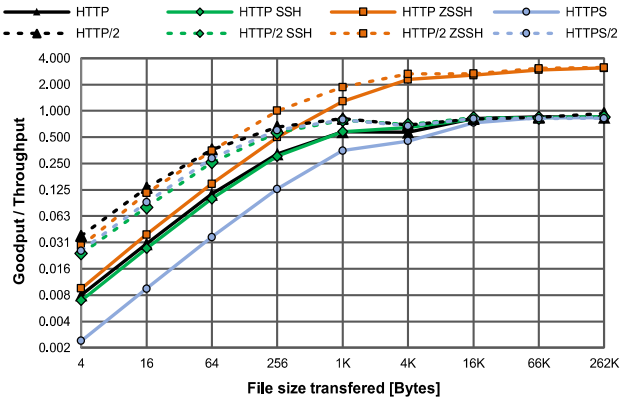

FIGURE 11. Channel usage efficiency of tested protocols.

Fig. 11 shows the efficiency in terms of *goodput / throughput rate* (the *goodput* is calculated by the benchmarking tools using the size of the file and the transfer time). It can be observed that *HTTPS* presents the worst performance for every file size, but particularly for smaller ones. The reason is that the overhead at low rate transmissions is high: when the connection is established, the certificates of each of the 100 requests that compose each series of tests have to be verified.

However, this is not necessary in the *HTTP SSH* case, as it keeps open a communication channel requiring only one verification. As an example, the efficiency when transferring 256 bytes is 0.3 for *HTTP SSH*, and only 0.13 for *HTTPS*. The impact of this issue is reduced when the file size is increased, but the performance of *HTTP SSH* is always better.

Some reasons can be adduced for explaining this difference: HTTP connections through uncompressed SSH *(HTTP SSH)* are handled through channels created inside an SSH session. These channels are flow-controlled, so the information contained in several TCP packets can be multiplexed and sent

at once through the secure channel. However, this alone does not explain its better performance when the payload is small.

The reason behind this is that the Core IoT application only has to connect to the local SSH proxy, which will be in charge of connecting it to the remote destination socket. During TCP connection handshake and release, several TCP packets are exchanged to configure both sides of the connection, without carrying any part of the test files. However, when using SSH, all these packets are sent locally between the IoT application and the SSH local proxy, so they do not have to be transmitted through the 802.15.4 interface. This saves at least 3 RTT times for each HTTP request together with the task of certificate validation. In this respect, the *keep_alive* feature [56] can be used to overcome the TLS limitation of having to check certificates after every single handshake request. Nevertheless, this is not a transparent feature, as it requires a valid heartbeat request to be first coded and performed from IoT applications.

Finally, the saving is even higher when *HTTP SSH Compressed* is used, thanks to the high compressibility of the SSH payload: HTTP datagrams and their own IoT payload are usually plain text with semantic-based descriptions [57].

HTTP/2 incorporates some of the features already provided by SSH, and this results into a performance boost. The benchmark tool *hdload* makes use of the streaming feature of HTTP/2, allowing this protocol to reuse existing TCP connections. This fact, together with its binary compressed header, fully optimizes its transmissions. Nevertheless, the provision of decoupled security through SSH with compression will still result in a better performance when comparing to HTTPS/2.

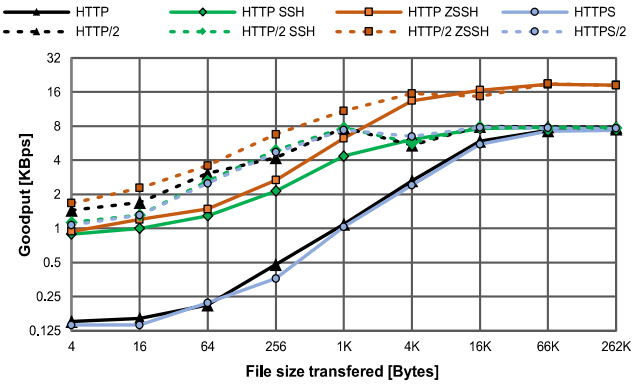

FIGURE 12. Goodput comparison among tested protocols.

In Fig. 12, the achieved *goodput* is compared between protocols. This chart is congruent with the previous one, showing again a performance boost on SSH tunneled communications for HTTP, which results higher in the scenario using compression, particularly when the payload is moderate. When testing HTTP/2, this performance increase only happens when compression features of SSH are enabled, and this improves as the file size increases.

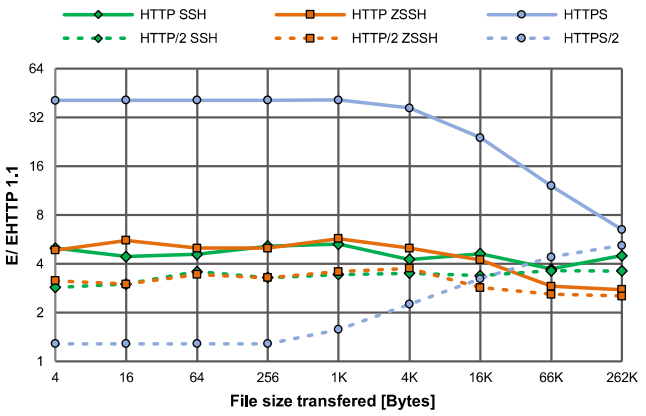

FIGURE 13. CPU energy consumption increment taking HTTP as reference.

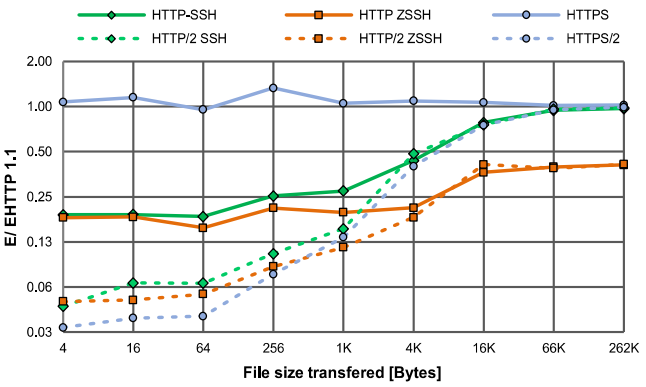

FIGURE 14. Energy saving on wireless interface taking HTTP as reference.

In Fig. 13, the impact of using security on CPU energy consumption is shown. The measurements take as a reference the plain HTTP requests. Increases in CPU energy consumption are considered to be proportional to the CPU time needed for these requests to go [58]. CPU time is computed using the "*time*"[13] Linux tool on benchmarking programs and also on SSH when tunneling is used.

In Fig. 14, the impact of each option on the energy consumption of the 802.15.4 interface is presented. It is also considered to be proportional to the time a communication needs to use the interface. On both figures, an increment in energy saving when using SSH to secure HTTP communications can be observed, wereas a regression when small files are used appears when using HTTP/2. This is due to the extra complexity of establishing tunnels when using SSH. However, as this only happens once, this effect is minimized when enabling compression and once the payload size increases or the number of transmissions between each tunnel establishment is higher.

VII. IOT HOST SERVER RESULTS

The aim of the tests presented in this section is to gauge the performance of the IoT *Host Server,* mainly in terms of scalability. Thus, potential limitations inherent to the proposed

[13]https://linux.die.net/man/1/time

SSH-based security decoupling solution are studied and characterized when a high number of devices are to be managed by a single server. Two implementations of SSH will be used in the tests, namely OpenSSH and Dropbear.

The first parameter to be measured is RAM consumption. In addition, execution time will be evaluated in different tests, so as to characterize its dependence with the SSH embodiment, the payload requests, and the distribution of proxifications in SSH parallel sessions. These results would help to outline other possible SSH limitations, related to the number of established sessions' channels and concurrent proxifications sharing a single SSH session. This scenario includes two servers:

- *Server_A*, using Centos7, hosts both SSH server implementations. The first one is DropBear 2018.76, built in Ubuntu 14.04.5 LTS as stand-alone (portable version and in debug mode). This build is made using a modified version of *eglibc* v3.10.0 where FD_SETSIZE has been increased to override *select()* POSIX legacy limitation of 1,024 file descriptors. DropBear was also modified to increase that limit, and also others as MAX_LISTENERS, MAX_CMD_LEN and MAX_CHANNELS. The second one is standard OpenSSH_7.4p1 with OpenSSL 1.0.2k-fips, included in Centos7 by default.
- *Server_B*, also using Centos7, is a traffic generator gathering both SSH client implementations. DropBear client has been built using the same features and characteristics as DropBear server used in *Server_A*.

A. RAM CONSUMPTION CHARACTERIZATION

The RAM consumption of SSH servers is the first aspect to be analyzed, as the number of established and active proxifications is mainly constrained by the total RAM consumption.

To proceed into this analysis, each SSH client instance in *Server_A*, emulating an IoT *device*, should establish its own session towards *Server_B* and therefore, the number of sessions between servers would represent IoT *devices*' SSH connections. Each session gathers one communication channel through which a direct proxification can be established. This is a usual case whence IoT *devices* need to communicate periodically with the IoT *platform* in the IoT *Host Server*.

This test aims to evaluate the RAM consumption of every single proxification related to every IoT *device* a IoT *Host Server* is managing, when each IoT *device* requires just one proxification. Proxifications are gathered together in SSH sessions as follows: let S be the total number of SSH sessions the IoT *Host Server* manages, and P the total number of proxifications per session. In this case P = 1 and S is equal to the number of total proxifications to be established (total number of IoT *devices*); all according to Fig. 10.

The test also allows a comparison between OpenSSH and Dropbear. In each measurement, either OpenSSH or DropBear are used both as client and server, i.e., each SSH session is established using either DropBear client and server or OpenSSH client and server.

As IoT *Host Servers* also may relay proxifications between them, particularly when a small IoT *Host Server* acts as a gateway, this configuration (namely "*aggregated*") is also studied and compared here. In this other case, a single SSH session is established from *Server_B* to *Server_A* with multiple proxifications on that very single SSH session and thus, S = 1 and P is equal to the number of proxifications to be established, again according to Fig. 10.

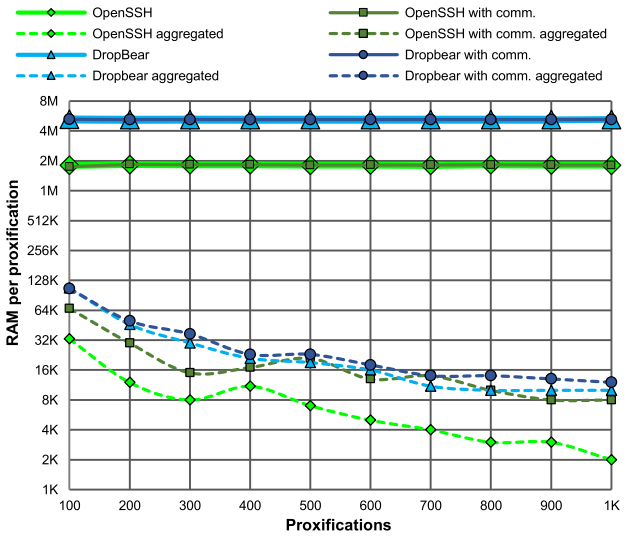

FIGURE 15. RAM required per SSH proxification.

The results of this first test (Fig. 15) show the RAM consumption per proxification when they are aggregated into a single SSH session. Once SSH sessions and proxifications are established, each of them is tested via an HTTP request of 1 Kbyte payload. This request is performed through every single proxification in parallel, using a simple client programmed in Golang named "*Concurrent*" in *Server_B*. *Server_A* is hosting Caddy web server and it is reached locally from the proxification. Caddy is also programmed in Golang.

The RAM measuring process is performed globally using the " *free*[14]" Linux command through the entire system. *Server_A* is used exclusively for this testing and free RAM is continuously monitored during the whole process. Its maximum value is measured for three different phases through each test: *i*) before starting the test (and after " *sync*[15]"), *ii)* once all SSH sessions have just been established and *iii)* during the HTTP scheduled requests are performed through proxifications.

As observed from the figure, the consumption of RAM per proxification both in Dropbear and OpenSSH when a single SSH connection handles a single proxification is remarkably high. This is mainly due to the fact that every incoming SSH connection handled by *Server_A*, instances a completely new SSH server process, which consumes a large amount of RAM. DropBear's consumption doubles OpenSSH's one, as it is

[14]http://man7.org/linux/man-pages/man1/free.1.html
[15]http://man7.org/linux/man-pages/man1/sync.1.html

stand-alone and every required library is statically linked. Besides, it is built with debug functionalities, which increases its RAM footage even further.

Despite this result can be dramatically improved through severe optimizations in both SSH implementations, it would not easily reach similar figures to those of TLS (tens of Kbytes per session). This is mainly caused by the process instancing for every different incoming SSH session. To mitigate this effect, the proposed scheme can benefit from the current trend in IoT deployment: it is usually required an IoT *Gateway* to be installed between IoT *devices* and IoT *Host Servers*.

This IoT *Gateway* would act as a small IoT *Host Server* which gathers small numbers of local IoT *devices* performing an intermediary role. This approach can be easily rendered using SSH and its proxification aggregation features.

In this case, each proxification between the IoT *Gateway* and the IoT *Host Server* requires much less RAM allowing this two-level structure to dramatically reduce the IoT *Host Server* RAM requirements.

The scenario is now used to run another test, aimed at measuring the dependence of RAM footage per proxification in the server, versus the payload of each HTTP request to be forwarded. In this test, a set of 1,000 active proxifications (handling an HTTP request each) are processed in two different ways: a) each one through its very own SSH session and b) all of them aggregated together into just one SSH session. Different payloads are tested to feature a possible dependence with RAM footage.

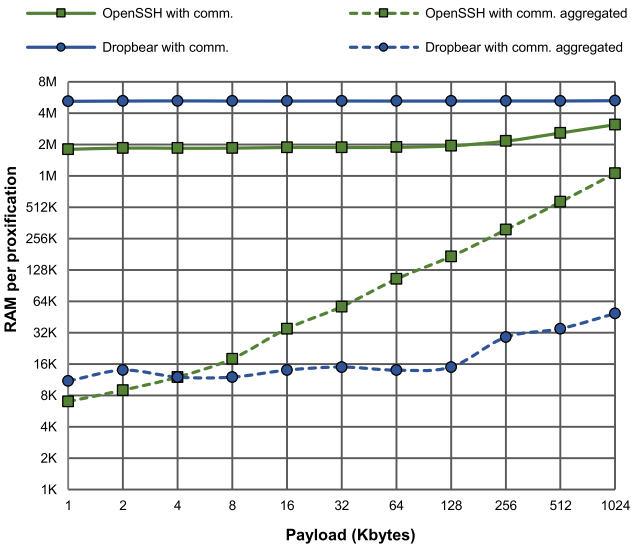

FIGURE 16. Dependency of RAM required per proxification with the HTTP request's payload the proxification handles.

Fig. 16 shows the results of these tests. RAM footage has little dependence with payload when direct connections from IoT *devices* are forwarded to the IoT *Gateway*. However, in an aggregated scenario whence an IoT *Gateway* forwards proxifications towards an IoT *Server*, it is clear that OpenSSH renders much worse than DropBear. This is mainly due to its internal handling of SSH channels' buffers, as its window communication channel, i.e. for each proxification. This explains the linear dependence of the RAM footage required per aggregated proxification with the payload of the HTTP request.

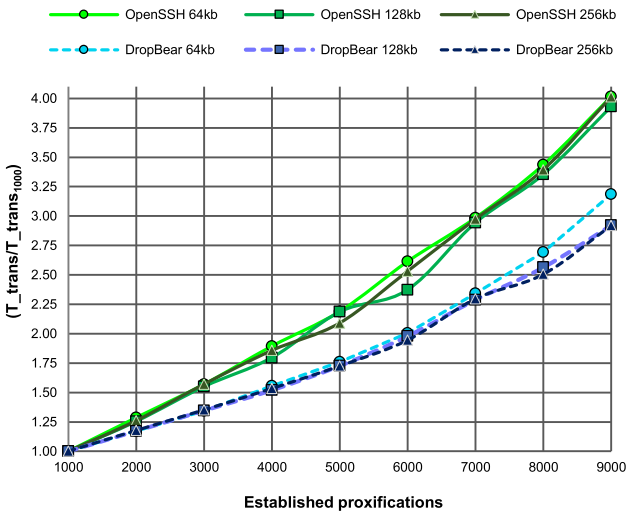

FIGURE 17. Normalized performance of 1000 active proxifications according the total established proxifications, the HTTP request's payload to process per proxification and the SSH embodiment used.

Another test is proposed in the second scenario in order to study the possible performance penalty an SSH instance has to face when coexisting with inactive proxifications, and its possible dependence with payload size. This test only deals with aggregated SSH sessions. Each SSH session has 1000 active proxifications (P = 1000) and each of them processes a single HTTP request of different payloads (64kb, 128kb and 256kb).

The required time for completing each test is measured by the script "*Concurrent*". This HTTP client starts all scheduled requests at once and measures both total payload responses and total time to complete all HTTP parallel requests. "*Concurrent*" is launched once all parallel SSH sessions and proxifications have successfully been established. A total of 25 tests of each series are performed and the results averaged and presented in Fig. 17.

The results show a penalty performance suffered simply dividing the time required to process 1000 parallel requests on each case (*T_trans*) by the time it is required when there are only 1000 proxifications aggregated in total with no inactive proxifications (T_trans_{1000}). Please note that each test series has its own T_trans_{1000}.

These results show little dependence with the payload size, but a strong one with inactive proxifications in the same SSH session. This exponential dependence has to do with the way the implementations of SSH handle every communication channel. Though they are inactive, they seem to penalty the overall performance despite no extra work seems to have been done. The main cause for this performance degradation is the way process' opened sockets are handled: This poor

performance is related to the legacy I/O framework used by both SSH implementations, which is not intended for processes with a large amount of sockets (descriptors) to handle.

Table 1 is computed from a testbed data of 100,000 randomly monitoring operations across N contiguous descriptors [59]. The table presents the performance degradation in monitoring N descriptors experienced by *select()* and *poll()* compared to the performance when N = 10. This performance is computed dividing the time required to monitor N descriptors by the time required to monitor just 10.

TABLE 1. Performance degradation of different I/O frameworks when monitoring 100,000 operations among N descriptors.

Number of descriptors monitored (N)	*select()*	*poll()*	*epoll()*
10	1.00	1.00	1.00
100	4.75	4.11	1.02
1,000	57.38	47.95	1.29
10,000	1,622.95	1,273.97	1.61

Both *select()* and *poll()* are system calls from a multiplexing driven I/O framework, whose performance penalty behaves exponentially with the total number of descriptors able of being monitored in the process. This is mainly because "*On each call to select() or poll(), the kernel must check all of the specified file descriptors* (handled by the process) *to see if* (the required descriptors) *are ready.*" [59]. The Linux kernel, under this I/O frameworks behavior, must always check all descriptors the process has opened, independently of which ones actually require to query.

On the other hand, the performance penalty suffered by *epoll()* is much lower because it embodies a signal driven I/O API whence the demanding process is sent a signal by the Kernel once I/O is possible on the very specific file descriptor that signal refers to. Furthermore, this smaller performance penalty can be even lowered as it heavily depends on hardware features. In [59], the author states regarding *epoll*() that "*the small decline in performance as N increases is possibly a result of reaching CPU caching limits on the test system*".

Due to the limitations of SSH implementations, a session is not able to handle large amounts of sockets. On the other hand, using a session per proxification may not be feasible in all circumstances, as RAM footage is high. In the next tests, we evaluate the slower degradation of having multiple parallel SSH sessions with 128 established proxifications each. However, only 64 sessions out of the total would be active processing each one's proxifications a HTTP request of 64 Kbytes adding up a total of 8192 requests for each test. Again, every test series is performed 25 times and averaged.

As shown in Fig. 18, as the amount of processes grows, the kernel would have to split the CPU resources between them, leading to an inefficient scenario: a performance penalty is noticed even though no further requests are processed. However, this degradation trend is much slower than the previous one suffered by processes themselves.

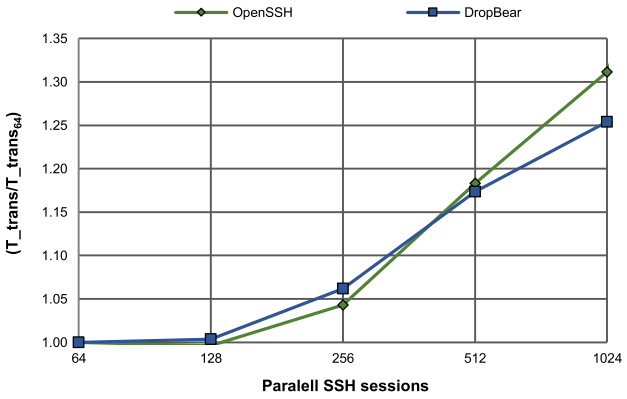

FIGURE 18. Normalized performance of 64 parallel SSH sessions (S = 64) each one with 128 active proxifications (P = 128), handling a HTTP request of 64 Kbytes payload each, with the increasing number of established parallel SSH sessions with 128 established (and unused) proxifications each.

A performance comparison between both approaches (Fig 17. and Fig.18) is summed up in Table 2. In this table, "N" features the amount of *unused proxifications* (established – used). The tests presented in Fig. 17 were performed having 1,000 active proxifications whereas the tests in Fig. 18 had 8,192.

Please be noted that only a degradation of 25-32% is suffered once $(1024 * 128) - 8192 = 122,880$ unused proxifications are set distributed in SSH sessions (Fig. 18), whereas degradations of 290% to 400% were suffered when only $9,000 - 1,000 = 8,000$ unused proxifications were established through a same SSH session process (Fig. 17). Thereby, a possible solution for escalation could be this trade-off between SSH sessions and proxifications per SSH session.

B. ESTIMATION OF THE OPTIMAL RELATIONSHIP BETWEEN P AND S

The next test is presented to calculate the optimal relation between these two parameters: Proxifications per session and number of sessions. In this test 10,000 active proxifications are to be handled in different configurations.

Each configuration will have increased the total number of SSH sessions (S) among which all proxifications (P) have to be distributed into, provided that

$$P * (S - 1) <= 10,000$$

and

$$P * S >= 10,000$$

and leaving the extra proxifications unused. The time computed to finish a HTTP request of 64 Kbytes through each one of the 10,000 active proxification is noted as *T_trans*. Y-axis represents the performance comparison calculated as the time required for each test, against the time required by the optimal configuration found referred to as T_trans_{min}. This optimal time is reckoned as the one having the lowest *T_trans* of the

TABLE 2. Performance penalty (%) of different SSH configurations with N unused proxifications.

Configurations	*N=1,000*	*N=2,000*	*N=4,000*	*N=8,000*	*N=24,576*	*N=57,344*	*N=122,880*
OpenSSH (64kb) P=N, S=1	29%	57%	119%	302%	-	-	-
DropBear (64kb) P=N, S=1	17%	35%	76%	219%	-	-	-
OpenSSH P=128, S=N/128	~ 0%	~ 0%	~ 0%	~ 0%	4%	18%	32%
DropBear P=128, S=N/128	~ 0%	~ 0%	~ 0%	~ 0%	6%	17%	25%

entire series and thereby, a penalty in performance is suffered in any test wherein T_trans/T_trans_{min} is higher than 1. Both OpenSSH and DropBear test series have each one its own T_trans_{min}. Please be noted that this figure uses non-linear scales on both axes.

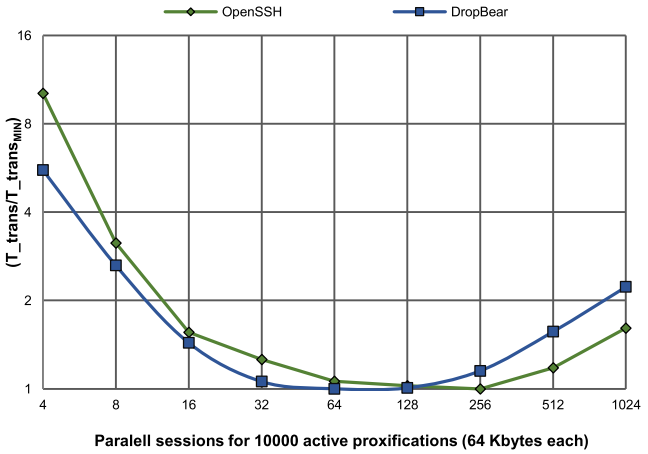

FIGURE 19. Normalized performance of parallel SSH sessions with different P and S configurations to achieve 10,000 active proxifications (SSH sessions with all proxifications active handling a HTTP request of 64 Kbytes payload each).

According to the results displayed in Fig.19, a reasonable reference ratio between proxifications per SSH session and SSH sessions seems to be P/S = 79/128, for both OpenSSH and Dropbear. This result is, however, only applicable for this scenario, where 6 virtual cores (single thread) are available in the IoT *Host Server*. The amount of sessions in the previous relation is likely to change if the number of cores differs. Thus, in virtual machines with different number of virtual cores the grade of dependency on the number of cores should be computed to correct this ratio.

Once this optimization has been devised, a comparison between SSH proxification scheme and direct HTTPS connection is of paramount importance for testing the feasibility of the proposed scheme. Therefore, our final test will compare the time spent by each approach, allowing us to compute the *Request Per Second* (RPS) metric: it represents the amount of HTTP requests of 1Kbyte that can be processed in parallel in just one second by the IoT *Host Server*. The aim of this test is to measure the server performance according to the approach chosen to establish each communication with the IoT *devices*. Thereby, several specific conditions are to be met:

- Each HTTPS direct request from the concurrent client in *Server_B* to Caddy server in *Server_A* has to be performed independently. Thus, HTTPS/2 will not be able to use its multiplexing features. Tests are performed this way as these multiplexing features are only available when a single client performs multiple requests in parallel (e.g.: loading a website with multiple objects), whereas the present test aims to compare independent communications: one per IoT *device*, where HTTPS/2 multiplexing cannot easily apply mainly because their origins are different IoT *devices*.
- HTTPS requests are performed without validating SSL certificates. This task is intended to be done by the client alone and would slow down every HTTPS request as a result. As the aim of the test is to measure the performance of the server, the time for HTTPS certificate validation for each request does not apply this time.
- DNS resolutions are avoided, so as to prevent an unfair performance penalty to HTTPS and HTTPS/2. Each DNS resolution (or even cache query) would increase the time required for each request and would penalize server performance measure as explained through previous point and thus, it is left out of consideration.
- Several kernel and system optimizations are included in this test to overcome security limitations of the operating system.[16] These modifications are mainly intended for avoiding the wrong assumption of an ongoing DDoS attack[17] to the IoT *Host Server*, which would unfairly penalize much more HTTPS and HTTPS/2 requests.
- HTTP requests secured through proxifications provided by OpenSSH and DropBear are performed as in previous tests. This particularly refers to the fact all SSH sessions and proxifications will already be established before the proper test is run and the measuring time starts to compute. This is scheduled this way because the scheme presented handles persistent SSH sessions from IoT *devices* to the IoT *Host Server*. Therefore, when an HTTP request is required from the IoT *device*, there should be no need in establishing any further SSH session and proxification as they are already established

[16]https://www.kernel.org/doc/Documentation/networking/nf_conntrack-sysctl.txt

[17]https://www.kernel.org/doc/Documentation/networking/ip-sysctl.txt

(this is a major advantage of the proposed SSH-based scheme of IoTsafe).

Each test series is performed 25 times and the results are averaged. Results of these test series are shown in Fig. 20 using non-linear scales on both axes.

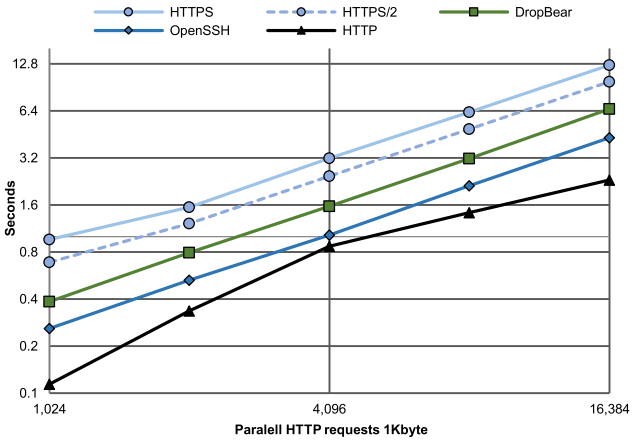

FIGURE 20. Time required to perform different parallel HTTP requests of 1Kbyte through studied protocols. SSH proxifications are established using P/S optimized values.

TABLE 3. Averaged requests per second (rps) and performance improvement (ΔP).

Protocol	Requests Per Second (RPS)	Performance increment (ΔP) vs. HTTPS	Performance increment (ΔP) vs. HTTPS/2
HTTPS	1,297	0 %	-22%
HTTPS/2	1,665	28 %	0%
DropBear	2,542	96 %	53%
OpenSSH	3,857	197 %	132%
HTTP	6,284	-	-

The linear behavior of each test series with respect to the amount of parallel HTTP requests shows a stable rate of RPS processed by the IoT *Host Server*. Should the amount of requests increase any further, some connections would have to be discarded and the effective RPS would decrease due to congestion. Plain HTTP insecure processed requests are displayed as reference.

As a summary, a comparison in terms of RPS and performance has been computed (see Table 3). A significant improvement in the performance of IoT *Host Server* can be observed for both SSH implementations with respect to HTTPS and HTTPS/2: OpenSSH and DropBear. As an example, OpenSSH outperforms HTTPS by a factor of 3 in terms of RPS.

VIII. CONCLUSIONS

This paper has presented IoTsafe, a novel communication scheme based on SSH for IoT environments using HTTP or HTTP/2 that fits the security requirements of portability and upgradeability when integrating into IoT *devices*. It also provides performance improvements in both energy savings and goodput with respect to a TLS scheme, when HTTP is used. Regarding HTTP/2, a performance boost in channel usage and goodput is achieved, in exchange for an increase in energy consumption, but only when small files are transferred and proxifications are to be set frequently. The proposed SSH-based approach allows HTTP and HTTP/2 to work with new low-powered interfaces with high constrains in MTU size and data transfer rates. This is achieved thanks to a proxy-like scheme that makes security independent from the firmware libraries and IoT software. In addition, SSH provides compressing features.

The results show that IoTsafe is a viable alternative in IoT projects. As it relies on well-known protocols and Linux long-term supported technology, it can benefit from its features allowing simpler IoT RESTful applications to be designed as security is decoupled from core IoT software.

IoTsafe allows simpler upgrading procedures, reducing software maintenance costs and extending IoT *devices* lifespan whilst their security algorithms can easily be kept up to date. This feat is easily achieved as SSH can work stand-alone in IoT *devices* allowing its upgrading procedure to be independent from the rest of the IoT *device's* pieces of software.

The analysis of SSH performance in the IoT *Host Server* concludes that SSH current implementations are suited for both small and medium IoT environments and are able to scale to larger ones using a three level scheme: IoT *device, IoT Gateway* and IoT *Host Server*. In the proposed scheme, each IoT *Gateway* could host and communicate simultaneously with more than 1,000 IoT *devices* each, with a RAM consumption of 5GB using the stand-alone debug version of DropBear. Anyhow, IoT *devices*' communications aggregated into IoT *gateways* towards an IoT *Host Server* would allow to gather into a single server a far larger number of proxifications than in a plain gateway. A server with 16 Gb of RAM and 6 virtual cores could accept more than 262K established proxifications (262K IoT *devices* if each one requires just 1 proxification). This implementation could reach a rate of 2,542 RPS with DropBear stand-alone embodiment and this figure could improve up to 3,857 RPS with OpenSSH, in both cases managing HTTP requests of 1Kbyte payload each secured through used proxification.

As future work, the performance could be further boosted if a more efficient SSH embodiment was devised: removing unused SSH functionalities and using a more efficient I/O framework. A promising alternative would stand in Golang. This language provides high performance in those required features (as it uses *epoll()*[18]) and also presents an encouraging SSH package[19] which simplifies the development of highly efficient SSH servers and clients. Anyhow, IoTsafe implementations are able to proceed with current SSH embodiments, and in a nearby future, thanks to its unique features in decoupling security from the rest of the software, a straightforward update will be possible just replacing SSH binaries and related IoTsafe configuration files and scripts

[18]https://github.com/golang/go/blob/91c9b0d568e41449f26858d88eb2fd085eaf306d/src/runtime/netpoll_epoll.go

[19]https://godoc.org/golang.org/x/crypto/ssh

only. This would extend IoT *devices* life cycle and cut costs in maintaining its firmware and features updated in terms of security.

ACKNOWLDEGMENT

A preliminary version of this article, titled "SSH as an Alternative to TLS in IoT Environments using HTTP," was presented at Global Internet of Things Summit (GIoTS), Bilbao, Spain, 2018, pp. 1-6. doi: 10.1109/GIOTS.2018.

REFERENCES

[1] C. Visual, *The Zettabyte Era: Trends and Analysis*. San Jose, CA, USA: Networking Index, 2014.

[2] Eclipse IoT Working Group, IEEE IoT, AGILE IoT. (2017). *IoT Developer Survey*. https://ianskerrett.wordpress.com//iot-developer-trends-edition/

[3] Eclipse IoT Working Group, IEEE IoT, AGILE IoT. (2016). *IoT Developer Survey*. [Online]. Available: http://iot.ieee.org/images/files/pdf/iot-developer-survey-report-final.pdf

[4] Flashpoint-intel. *Mirai Botnet Linked to Dyn DNS DDoS Attacks*. Accessed: Oct. 21, 2016. [Online]. Available: https://www.flashpoint-intel.com/mirai-botnet-linked-dyn-dns-ddos-attacks/

[5] D. Fitzgerald, "Hackers infect army of cameras, DVRs for massive Internet attacks," *Wall Street J.*, Sep. 2016. [Online]. Available: https://www.wsj.com/articles/hackers-infect-army-of-cameras-dvrs-for-massive-internet-attacks-1475179428

[6] C. Stupp. *Commission Plans Cybersecurity Rules for Internet-Connected Machines*. Accessed: Oct. 4, 2016. [Online]. Available: https://www.euractiv.com/section/innovation-industry/news/commission-plans-cybersecurity-rules-for-internet-connected-machines/

[7] S. Ray, A. Basak, and S. Bhunia, "The patchable Internet of things," *IEEE Spectr.*, vol. 54, no. 11, pp. 30–35, Nov. 2017.

[8] V. Bourne. (2018). *Seizing Control of Software Supply Chain Security*. [Online]. Available: https://go.crowdstrike.com/rs/281-OBQ-266/images/ReportSupplyChain.pdf

[9] C. S. I. R. T. S. (CSIRT.SK), *Security Advisory Skcsirt-sa-20170909-Pypi-Malicious-Code*. Accessed: Apr. 4, 2018. [Online]. Available: https://www.sk-cert.sk/wp-content/uploads/2018/04/skcsirt-sa-20170909-pypi-malicious-code.pdf

[10] P. Anand, "Overview of root causes of software vulnerabilities—Technical and user-side perspectives," in *Proc. Int. Conf. Softw. Secur. Assurance (ICSSA)*. St. Polten, Austria, Aug. 2016, pp. 70–74.

[11] W. Qiang, J. Yang, H. Jin, and X. Shi, 'PrivGuard: Protecting sensitive kernel data from privilege escalation attacks," *IEEE Access*, vol. 6, pp. 46584–46594, 2018.

[12] T. Y. C. Ylonen Lonvick, *The Secure Shell (SSH) Protocol Architecture*, document RFC 4251, Internet Engineering Task Force, Jan. 2006. [Online]. Available: https://tools.ietf.org/html/rfc4251

[13] NIST,*Recommendation for Key Management*. Gaithersburg, MD, USA: Elaine Barker, 2016.

[14] The European Parliment and the Council of the European Union. *General Data Protection Regulation*. Accessed: Apr. 27, 2016. [Online]. Available: https://eur-lex.europa.eu/legal-content/EN/TXT/PDF/?uri=CELEX:32016R0679&from=EN

[15] European Union Agency for Network and Information Security (ENISA). *Baseline Security Recommendations for IoT*. Accessed: Nov. 7, 2017. [Online]. Available: https://www.enisa.europa.eu/publications/baseline-security-recommendations-for-iot

[16] *Directorate-General for Communications Networks, Content and Technology, Cybersecurity Act*. Accessed: Sep. 27. 2017. [Online]. Available: https://eur-lex.europa.eu/legal-content/EN/ALL/?uri=COM:2017:477:FIN

[17] E. C. Organisation. (Oct. 2017). *Elements from ECSO members on the EU Certification Framework*. [Online]. Available: http://www.ecs-org.eu/documents/uploads/elements-from-ecso-members-on-the-eu-certification-framework.pdf

[18] M. Warner, C. Gardner, and R. Y. Wyden, S. Daines, "Internet of Things (IoT) Cybersecurity Improvement Act," in *Proc. 115th Congr.*, 2017, pp. 1–20. [Online]. Available: https://www.congress.gov/115/bills/s1691/BILLS-115s1691is.pdf

[19] L. Singaravelu *et al.*, "Reducing TCB complexity for security-sensitive applications: Three case studies," *ACM SIGOPS Operating Syst. Rev.*, vol. 40, no. 4, pp. 161–174, Apr. 2006.

[20] Gartner Inc, *Gartner Says Worldwide IoT Security Spending Will Reach 1.5 Billion in 2018*. Accessed: Mar. 21, 2018. [Online]. Available: https://www.gartner.com/newsroom/id/3869181

[21] J. Y. G. Camacho Macía, "Device, system and method for the secure exchange of sensitive information over a communication network," World Patent WO2015128523 Spanish, Mar. 9, 2015. [Online]. Available: https://patentscope.wipo.int/search/es/detail.jsf?docId=WO2015128523

[22] *PIN Transaction Security (PTS) Device Testing and Approval*. Mastercard, American: PCI Security Standards Council, 2018.

[23] C. Herder, M.-D. Yu, F. Koushanfar, and S. Devadas, "Physical unclonable functions and applications: A tutorial," *Proc. IEEE*, vol. 102, no. 8, pp. 1126–1141, Aug. 2014.

[24] M. Vai *et al.*, "Physical unclonable functions for IoT security," in *Proc. IEEE High Perform. Extreme Comput. Conf. (HPEC)*. Waltham, MA, USA, Aug. 2015, pp. 1–5.

[25] M. A. Prada-Delgado, A. Y. I. Vázquez-Reyes and I. Baturone, "Trustworthy firmware update for Internet-of-thing devices using physical unclonable functions," in *Proc. Global Internet Things Summit (GIoTS)*. Geneva, Switzerland, Jun. 2017, pp. 1–5.

[26] C. Helfmeier, C. Boit, D. Nedospasov, and J. P. Seifert, "Cloning physically unclonable functions," in *Proc. IEEE Int. Symp. Hardw.-Oriented Secur. Trust (HOST)*. Austin, TX, USA, Jun. 2013, pp. 1–6.

[27] U. Rührmair, U. Schlichtmann, and W. Burleson, "Special session: How secure are PUFs really? On the reach and limits of recent PUF attacks," in *Proc. Design, Automat. Test Europe Conf. Exhib. (DATE)*. Dresden, Germany, Mar. 2014, pp. 1–4.

[28] G. T. Becker, "The gap between promise and reality: On the insecurity of XOR arbiter PUFs," in *Proc. Conf. Cryptograph. Hardw. Embedded Syst. (CHES)*. Saint Malo, France, Sep. 2015, pp. 535–555.

[29] Y. Gao, D. C. Ranasinghe, S. F. Al-Sarawi, O. Kavehei, and D. Abbot, "Emerging physical unclonable functions with nanotechnology," *IEEE Access*, vol. 4, pp. 61–80, 2016.

[30] Y. Wen and Y. Lao, "Enhancing PUF reliability by machine learning," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*. Baltimore, MD, USA, May 2017, pp. 1–4.

[31] G. Grigoryan, Y. Liu, L. Njilla, C. Kamhoua, and K. Kwiat, "Enabling cooperative IoT security via software defined networks (SDN)," in *Proc. IEEE Int. Conf. Commun. (ICC)*. Kansas City, MO, USA, May 2018, pp. 1–6.

[32] M. A. Razzaque, M. Milojevic-Jevric, A. Palade, and S. Clarke , "Middleware for Internet of things: A survey," *IEEE Internet Things J.*, vol. 3, no. 1, pp. 70–95, Feb. 2016.

[33] A. H. Ngu, M. Gutierrez, V. Metsis, S. Nepal, and Q. Z. Sheng, "IoT middleware: A survey on issues and enabling technologies," *IEEE Internet Things J.*, vol. 4, no. 1, pp. 1–20, Feb. 2017.

[34] R. T. Tiburski, L. A. Amaral, E. D. Matos, and F. Hessel, "The importance of a standard securit y archit ecture for SOA-based iot middleware," *IEEE Commun. Mag.*, vol. 53, no. 12, pp. 20–26, Dec. 2015.

[35] Z. Shelby. (2012). *Constrained RESTful Environments (CoRE) Link Format, Internet Engineering Task Force*. [Online]. Available: https://tools.ietf.org/html/rfc7252

[36] T. Grance and J. Voas, "The Internet of things: Epic changes to follow," in *Proc. CSA Congr.*, 2015, pp. 25–28.

[37] K. A. McKay, L. Bassham, M. S. Turan, and N. Mouha, "DRAFT NIST IR 8114: Report on Lightweight Cryptography," Nat. Inst. Standards Technol., Gaithersburg, MD, USA, Tech. Rep., Aug. 2016. [Online]. Available: https://doi.org/10.6028/NIST.IR.8114

[38] D. Adrian *et al.*, "Imperfect forward secrecy: How diffie-hellman fails in practice," in *Proc. ACM Conf. Comput. Commun. Secur.*, Oct. 2015, pp. 5–17.

[39] S. J. Johnston, M. Apetroaie-Cristea, and Y. Scott, "Applicability of commodity, low cost, single board computers for Internet of things devices," in *Proc. IEEE 3rd World Forum Internet Things (WF-IoT)*. Reston, VA, USA, Dec. 2016, pp. 141–146.

[40] C. Duan *et al.*, "Design of an ARM9-based embedded industrial," in *Proc. IEEE Conf. Ind. Electron. Appl.* Hangzhou, China, Jun. 2014, pp. 1932–1935.

[41] A. S. Haron, M. S. A. Talip, A. S. M. Khairuddin, and T. F. Tengku, "Internet of things platform on ARM/FPGA using embedded linux," in *Proc. Int. Conf. Adv. Comput. Appl. (ACOMP)*. Ho Chi Minh City, Vietnam, Dec. 2017, pp. 99–104.

[42] K. Yangjian, "Development of ralink's wireless network interface card based on uclinux," in *Proc. Chin. Automat. Congr. (CAC)*. Jinan, China, Oct. 2017, pp. 3278–3281.

[43] S. Deshmukh and S. S. Sonavane, "Security protocols for Internet of Things: A survey," in *Proc. Int. Conf. Nextgen Electron. Technol. Silicon Softw. (ICNETS2)*, Mar. 2017. pp. 71–74.
[44] M. Conti, N. Dragoni, and V. Lesyk, "A survey of man in the middle attacks," *IEEE Commun. Surveys Tuts.*, vol. 18, no. 3, pp. 2027–2051, 3rd Quart., 2016.
[45] B. Visan, J. Lee, B. Yang, A. H. Smith, and E. T. Matson, "Vulnerabilities in hub architecture IoT devices," in *Proc. IEEE Annu. Consum. Commun. Netw. Conf.(CCNC)*. Las Vegas, NV, USA, Jan. 2017, pp. 83–88.
[46] O. Salman, S. Abdallah, I. H. Elhajj, A. Chehab, and A. Kayssi, "Identity-based authentication scheme for the Internet of Things," in *Proc. IEEE Symp. Comput. Commun. (ISCC)*. Messina, Italy, Jun. 2016, pp. 1109–1111.
[47] Android Open Soure Project, *SSLSocket Android Developers*. Accessed: Jan. 23, 2018. [Online]. Available: http://developer.android.com/intl/zh-cn/reference/javax/net/ssl/SSLSocket.html
[48] A. Razaghpanah *et al.* "Studying TLS usage in android Apps," in *Proc. Int. Conf. Emerg. Netw. Exp. Technol.* Seoul/Incheon, South Korea, Nov. 2017, pp. 350–362.
[49] T. Ylonen and C. Lonvick, *The Secure Shell (SSH) Connection Protocol*, document RFC 4254, Internet Engineering Task Force, 2006. [Online]. Available: http://tools.ietf.org/html/rfc4254
[50] J. D. de Hoz Diego *et al.*, "Leveraging on Digital Signage Networks to Bring Connectivity to IoT Devices,"in TELCON UNI, Lima, 2015.
[51] R. Zhang, G. Liu, X. Yuan, S. Ji, and G. Zhang, "A New intrusion detection mechanism in SELinux," in *Proc. Int. Symp. Syst. Softw. Rel. (ISSSR)*. Shanghai, China, Oct. 2016, pp. 532–58.
[52] L. Wang, J. Sushil, S. Anoop, P.Cheng, and S. Noel, "k-zero day safety: A network security metric for measuring the risk of unknown vulnerabilities," *IEEE Trans. Dependable Secure Comput.*, vol. 11, no. 1, pp. 30–44, Feb. 2013.
[53] (2016). *Caddy—The HTTP/2 Web Server with Fully Managed TLS*. [Online]. Available: https://caddyserver.com/
[54] NSA: Information Assurance Directorate. (2017). *Directorate of Capabilities Mobile Access Capability Package Version 2.0*. [Online]. Available: https://www.nsa.gov/Portals/70/documents/resources/everyone/csfc/capability-packages/MA_CP_v2.0.pdf
[55] T. Ylonen and C. Lonvick, *The Secure Shell (SSH) Transport Layer Protocol*, document RFC 42, Internet Engineering Task Force, Jan. 2006. [Online]. Available: https://tools.ietf.org/html/rfc42
[56] R. Seggelmann, M. Tuexen, and M. Williams. (2012). *Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS) Heartbeat Extension, Internet Engineering Task Force*. [Online]. Available: https://tools.ietf.org/html/rfc6520
[57] S. Kanti Datta and C. Bonnet, "Describing things in the internet of things. From CoRE link format to semantic based descriptions," in*Proc. Int. Conf. Consum. Electron.*, Aug. 2016, pp. 18–25.
[58] F. Kaup, P. Gottschling, and D. Hausheer, "PowerPi: Measuring and modeling the power," in *Proc. IEEE 39th Conf. Local Comput. Netw. (LCN)*, Sep. 2014, pp. 1–6.
[59] M. Kerrisk, *The Linux Programming Interface: A Linux and UNIX System Programming Handbook*. San Francisco, CA, USA: No Starch Press, 2010.

JORGE DAVID DE HOZ DIEGO was born in Valladolid, Spain, in 1983. He received the degree in telecommunications engineering from the University of Valladolid, in 2008, and the M.S. degree from the University of Zaragoza, Spain, in 2013, where he is currently pursuing the Ph.D. degree in information technologies and communications in mobile networks.

He was with FURIA Government Consortium, from 2008 to 2009. He focused on researching and the validation of DVB-H/DVB-SH standards and involved technologies. He has actively participated developing digital signage technology at Servicios de TI de Durango, Spain and Mexico. He has also lead several research projects related to the IoT environments partially funded by the National Council of Science and Technology of Mexico (CONACY).

JOSE SALDANA (M'11) was born in San Sebastián, Spain, in 1974. He received the B.S. and M.S. degrees in telecommunications engineering in 1998 and 2008, respectively, and the Ph.D. degree in information technologies from the University of Zaragoza (UZ), in 2011.

He is currently a Senior Postdoctoral Researcher with the Department of Engineering and Communications, UZ. His research interests include quality of service in real-time multimedia services, as VoIP and networked online games, traffic optimization, and resource management in wireless LANs.

Dr. Saldana is a member of the Internet Society. He currently serves as an Editor of the IEEE Access, and as an Area Editor of *KSII Transactions on Internet and Information Services*. For several years, he served on the Organization Committee of the IEEE Consumer Communications and Networking Conference, and in the Technical Program Committee of many other conferences as, e.g., IEEE ICC and IEEE GLOBECOM.

JULIÁN FERNÁNDEZ-NAVAJAS was born in Alfaro, La Rioja, Spain, in 1969. He received the degree from the Universidad Politécnica de Valencia, Spain, in 1993, and the Ph.D. degree from the University of Zaragoza (UZ), in 2000, all in telecommunications engineering.

From 1994 to 2002, he was an Assistant Professor with EINA, when he became an Associate Professor. He is currently with the Department of Electronics Engineering and Communications, Higher Engineering and Architecture School, UZ. He is a member of the Aragón Institute of Engineering Research (I3A). Since 1995, he has been a co-investigator of research grants from EU Research Projects, the Ministry of Science and Technology, the Sanitary Research Funds, and the Government of Aragón, Spain, in distributed multimedia system and wireless networks. Major industrial and mobile companies in wireless communications also support his work. He is also a co-investigator of several research projects supported by companies as Telefónica. His current research interests include communication networks with special emphasis on wireless networks, distributed multimedia systems, quality of service, and quality of experience.

JOSÉ RUIZ-MAS was born in Lorca, Murcia, Spain, in 1965. He received the degree from the Universitat Politécnica de Catalunya, Spain, in 1991, and the Ph.D. degree from the University of Zaragoza (UZ), in 2001, all in telecommunications engineering.

He was a Software Engineer at the company TAO Open Systems, from 1992 to 1994. From 1994 to 2003, he was an Assistant Professor with EINA, when he became an Associate Professor. He was the Director of Telefonica Chair with the University of Zaragoza, from 2004 to 2008, and a Coordinator of master in information technology and mobile communications, from 2007 to 2009. He is currently with the Department of Electronics Engineering and Communications, Higher Engineering and Architecture School, UZ. He is a member of the Aragón Institute of Engineering Research. Since 1995, he has been a co-investigator of research grants from EU Research Projects, the Ministry of Science and Technology, the Sanitary Research Funds, and the Government of Aragon, Spain, in distributed multimedia system and wireless networks. Major industrial and mobile companies in wireless communications also support his work. He is also a co-investigator of several research projects supported by companies as Telefónica and Teltronic. His current research interests include communication networks with special emphasis on wireless networks, distributed multimedia systems, quality of service, and quality of experience.