Security Filters for IoT Domain Isolation

Dominique Bolignano[1] and Florence Plateau[1]

[1] Prove & Run

dominique.bolignano@provenrun.com

Abstract. Network segregation is key to the security of the Internet of Things but also to the security of more traditional critical infrastructures or SCADA systems that need to be more and more connected and allow for remote operations. We believe traditional firewalls or data diodes are not sufficient considering the new issues at stake and that a new generation of filters is needed to replace or complement existing protections in these fields.

Keywords: Internet of Things; firewalls, filters, data diodes, security, formal methods, embedded devices, connected car.

1 Introduction

Modern IoT (i.e. Internet of Things) security architectures generally make use of partitions to define security domains and try to impose strict information-flow policies on the messages that transit from one domain to another. Typically, this is achieved by forcing all messages to transit through dedicated filters. The correct implementation of such filters is essential for the whole security of the system as the only path available to hackers to perform remote attacks, when the architecture is well designed, is to send triggering messages through these filters. Gateways in new automotive architectures are representative example of devices that implement filters. They are typically used to control the information flows between various security domains, such as the powertrain domain, the infotainment domain, the comfort domain, etc.

The proposed approach is meant to be applied to filters but only in situations where it is possible to explicitly identify and characterize commands and responses that are allowed to go through a given filter. As we will see, this is a sensible requirement to answer to the new security concerns arising in various contexts like: when connecting critical systems (e.g. Cyber Physical Systems), when connecting SCADA[1] systems (e.g. Operational Technology Systems connected to the IT infrastructure), in embedded automotive, aeronautic, or railway equipment, and more generally the IoT. For the IoT, this is mainly due to the fact that the large volume of connected devices creates huge opportunities and extremely good business models for hackers.

[1] SCADA: Supervisory Control And Data Acquisition, a type of industrial control system

In this paper we will first explain why there is a new challenge. We will then explain how this new challenge can be addressed in general, and then show how the security of the more demanding filters can be achieved.

2 The New Challenge with Remote Attacks

In this section we will show that the new challenge is mainly due to the existence of new business models for hackers. In the past, reaching an acceptable level of security mainly boiled down to implementing a few basic ingredients: cryptographic algorithms and protocols (such as digital signatures and encrypted communications), secure elements, etc. However, the advent of the IoT and the need to connect remotely to SCADA and critical systems are changing the security paradigm. There is now a real business model for hackers and organized crime syndicates in performing remote attacks. By investing a few millions of euros, they are now indeed almost sure to be able to identify potential large-scale remote attacks in current connected architectures with potentially a very high return on investment. In the IoT industry hackers can for example send a few devices to "reverse-engineering consultants" located in countries where this can be done legally or without too much risk. With the proper reconstructed documentation, they can then ask "creative" hacking consultants to prepare an attack. With such a budget at hand it is almost always possible to identify dramatic large-scale attacks, at least by exploiting bugs and errors that always exist in the OS and protocol stacks that are included in the Trusted Computing Base (TCB) of a device. Such errors can usually be found in the software architecture, or in the design, implementation or configuration of a device. The business model is usually quite obvious to find as in most situations such attacks make it at least possible to block the normal operation of the targeted infrastructure, causing damages that are way beyond the investment. In many cases such attacks could even create more dramatic situations that might lead to loss of life. An attack similar to the well-publicized Jeep attack [6] would correspond roughly to an investment of less than half a million of dollars (an estimate based on the detailed description of the identification phase of the attack by the authors), and if performed on a massive scale by criminal organizations could have led to the death of a very large number of people. These new business models (which in the case of the IoT is exploiting the combination of high volume and potentially physical impact) are bringing unpreceded security needs on the resistance to logical attacks and this is clearly a disruption in the security needs.

Security for high volume transactions (such as in payment systems) were (and are) mitigated by the use of proper risk management. Such risk management techniques are a lot less efficient (and in some cases not applicable) when it comes to IoT systems, as actions cannot be delayed or canceled as financial transactions can be. It is for example not practically possible to detect and block in real time an attack that would make all cars of a certain model turn right at a given time.

In the next subsection, we try to give more accounts on the fact that it is always possible to use the weaknesses of the OSs or protocol stacks that are part of the TCB.

2.1 The Challenge of Securing OSs, Kernels and Protocol Stacks

Various public databases (such as [2]) provide statistics on public bugs or vulnerabilities on all kinds of software. These databases clearly show that current OSs and kernels suffer from a great number of errors and weaknesses, no matter who writes them, and no matter how long they have been in the field. For example, new errors are still reported in the thousands every year on "well-known" systems such as Linux.

This situation is basically due to the inherent complexity of such OSs and kernels, which rely more and more on complex and sophisticated hardware. OSs and kernels are by nature concurrent and very complex because of the need to support various kinds of peripherals (interruption handling becomes more and more difficult), the performance objectives (e.g. complexity of cache management), the resource consumption issues (e.g. need for a sophisticated power management), etc. This complexity increases with time, increases with new IoT architectures and increases when it comes to microprocessors (as opposed to microcontrollers).

Even Trusted Execution Environments (TEEs), i.e. small security OSs that were introduced to very significantly reduce the size of the TCB, are regularly attacked ([11], [12], [16]).

The real challenge (and only known solution) is to produce and demonstrate that the OSs, kernels and software stacks that are part of the TCB are as close as possible to "zero-bug" i.e. are free from errors (in their design and implementation) that could be potentially exploited for logical attacks.

Traditional software engineering techniques such as exhaustive testing or code inspections are clearly not sufficient anymore to bring the level of assurance that is needed to secure complex OSs. This is due to the fact that there are too many different situations to consider for a kernel designer or tester and no real methods to review the quality of such kernel code in a systematic way, beside the use of proof techniques.

Instead we believe the only valid response to such complexity is a special class of formal methods, which are known as deductive techniques or proof techniques. Even other formal methods such as static analysis or model checking are not fully addressing the problem at hands. More details are presented in [1].

2.2 Limitations of Traditional Firewalls

The firewall is the right concept for controlling and building the segregation of an architecture but it has two significant drawbacks, (1) the configuration of a firewall is usually done on low level protocol concepts such as ports, IP addresses, etc., and making sure that such configuration implements the correct high-level security policy is difficult and very error prone at best (2) most importantly the TCB of a firewall includes at least its OS as well as its protocol stacks. Both are very error prone. In practice the complexity of the attack surface forbids this architecture from meeting the highest level of security, which is a must for the use-cases at hand. The first drawback can be avoided using ap-

plicative firewalls. This kind of firewalls allows to use higher level concepts to implement the security policy, which reduces the gap between the security policy and its implementation and hence the risk of error.

The second drawback is not only much more difficult to cope with, it is also very general: it applies to standard packet filter firewalls, to applicative firewalls, whether they use so-called "protocol break" or not. In all these firewalls there is at least an OS as part of the TCB and this OS is very error prone (i.e. the TCB is complex and not formally proven as it should be). The only exception, besides the new approach we are presenting in this paper is when a dedicated filtering hardware is used instead of an OS, but as of now such dedicated hardware are either too simple to address the need or too complex and error prone to be brought to the right level of security and of certification.

The attack surface of a traditional firewall is indeed unnecessarily large. In order to better understand this, let us consider an extremely simple (and unrealistic) security policy which is meant to impose that only the text command "set" can be sent remotely and that this command has a single mandatory parameter whose values can be only "on" or "off". Let us consider here that these commands are sent using TCP/IP on an Ethernet network and let us consider in a first step, for the sake of simplicity, that we are not using a VPN or more generally that messages are not signed or encrypted. We implicitly assume here in this illustrative example a firewall that is based on a standard secure OS (i.e. not based on a micro-kernel), but similar examples could be shown for other architectures.

Even if this security policy is only to accept two possible commands: "set on" and "set off", the degrees of freedom for the attacker are huge, and hence the surface of attack. First at the lexical level, the attacker could insert spaces in the text command (or other allowed delimiters such as tabs) in an attempt to exploit, for example, implementation bugs they have found in the lexical analyzer. They could in the same way exploit bugs in the syntactic analyzer (typically after reverse engineering it). The chances that they find problems that lead to real attacks there are limited because lexical and syntactic analysis is a well-understood software engineering problem with lots of available scientific know-how and tools. However, such weaknesses may still exist anyway (inadequate grammar type, buffer overflow due to improper memory configuration, etc.). What is important in this case is that such degrees of freedom will typically exist within each layer of the protocol stack (e.g. application layer, host-to-host transport layer, internet layer, network interface layer), which enlarges the attack surface, increasing the possibility of finding an exploitable bug. Wireless communication links are more exposed to these issues compared to wired ones because radio technologies (i.e. GSM, WiFi, Bluetooth, ZigBee, etc.) are usually complex and very error prone. In addition, in an OS such as Linux, protocols stacks are part of the kernel, which makes the attacks even simpler. In any case attackers will have an extremely large surface of attack (i.e. many degrees of freedom) to try to exploit bugs in the various protocol layers or in the OS itself.

2.3 Some Representative Attacks

Many attacks on IT systems are reported every day. Here we present some attacks of diverse kinds as a matter of illustration. The first one is the so-called 2015 attack on the Ukrainian power grid [14]. It is quite representative of weaknesses coming from the

complexity of the general architecture of large-scale IT systems and their configuration. In the case of this attack, it appears that only a weak security policy was enforced, i.e. users with only a low-level credential could still send any commands and receive any response from critical systems. In their comprehensive report Booz-Allen-Hamilton recommends among other measures (1) to install a stateful firewall or data diode, (2) to use a stronger authentication mechanism (such as two-factor authentication) for some of the accesses. Using a stateful applicative firewall would allow to enforce a proper security policy but the security level of existing firewalls[2] is not sufficient to cope with potential attacks (considering the level of return of investment that could be obtained by organized criminal organizations). A data diode is simpler and therefore can be brought to the right level of security (for example some data diodes have obtained an EAL7 Common Criteria certification) but can only make sure that the flow of information goes in a single direction: it cannot selectively block some commands and allow other. In addition, such systems usually require bidirectional communications, so data diodes are not adequate for this purpose. The filter we propose in this paper brings the benefits of both, i.e. the resistance of a data diode with the selectivity and programmability of an applicative firewall.

A second attack is the so-called Heartbleed attack which is one of the many attacks and vulnerabilities that were found on SSL/TLS overtime [13]. This latter attack is very representative of attacks that exploit the complexity of the software itself. Such bugs are very similar to the bugs that can be found in error-prone software components such as OS kernels or communication stacks.

Errors are not only found in software. They can also happen at the hardware level and lead to logical and remote attacks such as the recently announced Meltdown [7] and Spectre [8] attacks. Other cache attacks had been demonstrated in the past ([9], [10]) and new ones will probably be found in the future. We believe that hardware design should also be formally proven eventually, at least for their TCB part (MMU, ARM TrustZone mechanism, etc.). This will not prevent non-logical attacks such as the Rowhammer attack presented in [4], but it would prevent at least a large majority of logical attacks. However, errors in hardware that can be exploited for large scale remote attacks are very rare (one or two are found every year as of now) and they can usually be addressed by proper software countermeasures. Prove & Run has developed ProvenCore [18], a formally proven OS kernel that rely only on a few simple hardware mechanisms and to implement a very secure firmware update mechanism so that not only the risks from such hardware attacks are minimized but also that when they happen such problems can be easily fixed by a very robust over the air firmware update mechanism.

2.4 Addressing the New Challenge

The proposed approach to design an extremely secure filter builds on the approach we presented in [1]. We recall here briefly this approach before presenting new ideas that can be used to develop this filter. Some of these ideas are patent pending.

First it is important to use state-of-the-art security methodologies such as the one proposed by the Common Criteria framework. In particular we assume that for each architecture and use case a proper risk analysis and threat model are made available, and

[2] See the list of existing certified firewalls https://www.commoncriteriaportal.org/pps/

that a proper security target has been defined and is used to guide the security architect, the developers, the testers and the security evaluator. It is worth noticing that such documents can be reused from one evaluation to another so as to further reduce costs.

We also recommend as described in [1] to explicitly describe a clear "security rationale" that fully explains the hypotheses, conditions and reasons why the security architecture meets the desired security level. The security rationale should not only describe the countermeasures used to address each threat but also provide a detailed rationale as detailed and convincing as an informal mathematical proof.

The last step of the approach is to define an architecture that is based on a TCB that contains only formally proven kernels and protocol stacks. So, in the end the security rationale for the most complex parts of the TCB must rely on formally proven software (and using a tool is necessary to check that the proof is itself free of errors) whereas the other, simpler parts of the security rationale are presented as an informal proof which can be easily audited by experts. Now instead of formally verifying large OSs and kernels such as Linux or Android where new features and drivers are added on an ongoing basis so as to address new requirements, we propose to use a separate formally proven secure OS kernel, i.e. in our case ProvenCore, to address peripherals that need be secured and to run secure applications, in a way that allows us to:

- Retain the normal OS (for example Linux, Android or any other proprietary OS or RTOS) and thus benefit from all its features,
- Push the normal OS outside of the TCB, so that any error in the normal OS cannot be used to compromise the TCB,
- Use a proven OS to perform security functions.

Our formally proven kernel, ProvenCore, was designed in a way that makes it generic enough to be used as COTS (Commercial Off-the-Shelf) in virtually any IoT architecture.

We describe here how this can be done on ARM architectures that account for the vast majority of the IoT market, but the same approach can be transposed to other CPU architectures.

On ARM architectures and in particular on the Cortex-A and Cortex-M families of ARM microprocessors and microcontrollers, a security mechanism called TrustZone provides a low-cost alternative to adding a dedicated security core or co-processor, by splitting the existing processor into two virtual processors backed by hardware-based access control mechanisms. This lets the processor switch between two states, i.e. two worlds, typically the "Normal World" on one side and the "Secure World" on the other side. Therefore, TrustZone can be used as an extremely small and security-oriented asymmetric hypervisor that allows:

- The so-called Normal World to run on its own, potentially oblivious of the existence of the Secure World and,
- The Secure World to have extra privileges such as the ability to have some part of the memory, as well as some hardware peripherals, exclusively visible and accessible to itself.

In the proposed architecture the proven secure OS kernel, i.e. ProvenCore in our case, runs in the Secure World, and the rich but error-prone OS (Linux, Android, etc.) runs in the Normal World.

3 Proposed Approach and Solution

Here the key assumption (or in other words the requirement that is to be met for the proposed solution to be applicable) is that the list of commands and arguments that we want to allow in each direction can be made explicit and fully characterized. In other words, the security architect or administrator must be able to express a precise filtering security policy on the commands and arguments that must go across the filter from one security domain to the other. This may be difficult to do so within a standard information system: when security is not considered a high priority, the administrator is often not in a position to fully characterize all the commands and arguments in use nor even to identify all information flows. However, defining such a filtering security policy is a must as soon as a high level of security is needed e.g. for connected SCADA and critical systems. If a filtering security policy goes beyond a few trivial commands taking no arguments, then the implementation of this policy as a filter must be formally proven. In the next part we will explore how formally proven filters can address the challenge of critical IoT systems.

Connected Critical Systems and SCADAs
In the case of critical or SCADA systems it is usually necessary to accept incoming commands sent through a VPN by authorized remote agents either to perform routine maintenance and configuration or to exert manual control, at least in the case of an emergency situation where some remote administrators or decision makers need to take action quickly. In this case it is quite easy to identify and characterize the list of allowed incoming commands and outgoing responses[3]. The filtering security policy may be stateless or state-based. For example, an authorized user might be required to authenticate itself before issuing a command that modifies the configuration of the system. In this case the corresponding filtering security policy will obviously be state-based (i.e. identification and authentication are required before accepting a given command).

In the case of the Ukrainian critical infrastructure we would have proposed to clearly identify the list of remote commands that where acceptable for each authorized (and authenticated) user. This list could have been used as the base of a filtering security policy.

Embedded Devices and the IoT
In the case of embedded automotive, aeronautic, or railway connected equipment, or more generally any equipment part of the IoT, such filters will for example be placed in the gateways that exist for most of these systems, but may also be placed elsewhere (e.g. within the Telematic Control Unit of a car).

[3] The control of outgoing responses is less sensitive but still makes attacks more difficult and is also useful in case confidentially is at stake.

In the automotive industry, this approach could be used to filter incoming V2X[4] alerts coming from the car gateway. Today these alerts are delivered to the driver only through the dashboard, but in the very near future these alerts might be forwarded directly to the brake-control system, forcing the car to slow down. Filtering security policies may for example apply to data exchanged between the OEM and the car, and/or commands between various domains inside the car (such as chassis, engine or infotainment domains)

Because of the new business models available to enterprising hackers, high level security policies need to be expressed and enforced by the gateways. It is not easy (i.e. at the very best error prone and in some cases impossible with the right level of precision) to express such policies on the low-level objects (such as IP packets) that firewalls normally use. The administrator in charge of configuring such firewalls or the security architect defining the gateway has to use low level concepts such as ports whereas they would like to implement a high-level security policy where they could precisely specify and restrict the type of high level commands or responses that gets in or out.

As we have seen in section 2.2, the resistance of such implementations is not high enough to cope with the remote attacks at stake. Thus, even if the firewalls are properly configured, hackers will still have many ways to attack such entry points. They will typically bypass information-flow policies by exploiting bugs and errors commonly found in protocol stacks and OSs used to implement such firewalls. In fact, the security level reached by the most secure firewalls is usually very limited. In addition, the most secure ones have an expensive bill of material, which does not fit well with embedded systems requirements.

3.1 Proposed Architecture

Instead of filtering low-level packets we propose to filter high-level commands and arguments directly. We also propose to use a protocol break and to implement the filter as a formally proven (or at least highly secure) application (stateful or stateless, depending on the requirements of the task) that only operates on high-level commands and arguments, running on a formally proven and secure OS. This OS will have to guarantee a number of security properties (such as separation, integrity, …) and which in addition will have to enforce configurable information-flow policies between its components. This information-flow policy will make sure that communication flows coming from the outside (e.g. incoming commands) go through the filtering application which is the one applying the filtering security policy.

[4] V2X: Vehicle-to-everything communication

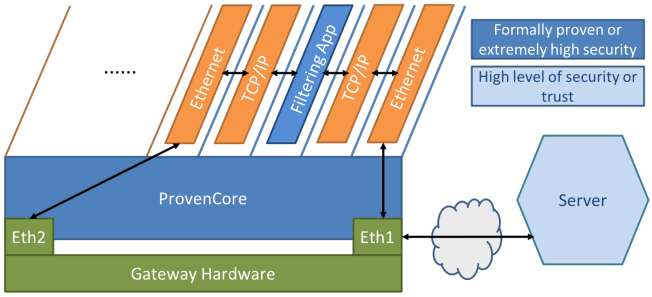


Fig. 1. Proposed Architecture

In **Fig. 1** we present an example of such an architecture in which we use ProvenCore to guarantee the security properties required to host the filtering application such as isolation, confidentiality and integrity [1]. ProvenCore also enforces a (programmable) information-flow policy between the various security applications and between the hardware peripherals and the corresponding drivers and other security applications. This policy ensures that there is no possibility for an incoming command or outgoing response to somehow bypass the filtering application. In the figure, it is materialized by the black arrows that represent the only authorized communication channels.

Since ProvenCore is a micro-kernel that guarantees the integrity and separation of the processes/applications it executes, even a severe problem within the hardware drivers or in the protocols stack themselves will not lead to any security problem besides a lack of availability[5].

In the example above the filtering application implements two filtering security policies: one on incoming commands, one on outgoing responses. More than one filtering applications can be used with more complex topologies in which ingoing (resp. outgoing) messages are routed to different filters according to their nature, but the overall principles remain unmodified.

Such an architecture allows us to design a filter that can be formally proven or more generally brought to the highest level of certification. We have summarized our architecture in **Fig. 2**.

[5] The lack of availability that would result from a successful attack on the protocol stacks can be mitigated by adding complementary security applications running in parallel to detect such attacks (such as a specialized IDS, i.e. Intrusion Detection System) and providing a security application in charge of reloading a new update over the air (or even inspect and repair the other software components). This is not featured here as it is out of scope of the current paper.

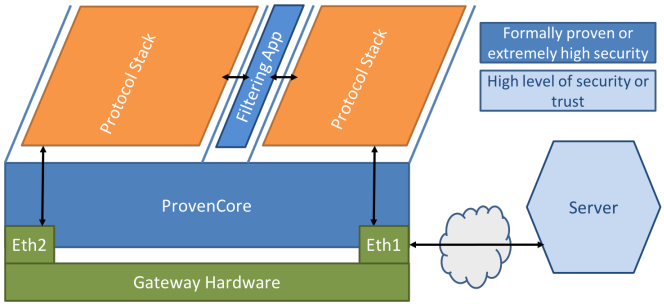


Fig. 2. Proposed Architecture, Simplified View

The TCB is composed of (1) a formally proven kernel, here ProvenCore which is the very first formally proven kernel on the market with the proper security features to support this filtering architecture, and (2) a formally proven filtering application (see section 4), which is by itself a very simple application, even if it includes the filtering per se but also the command and data lexical and syntactic analysis. This architecture thus allows us to obtain a filter (i.e. a particular applicative firewall) whose TCB is entirely formally proven to satisfy the given filtering policy expressed in a simple and high level formal language.

The fact that we use a protocol break on such a secure micro-kernel allows us to put all the protocol stack outside of the TCB. The separation properties of the OS, coupled with the access control mechanism between applications forces the information flow to go through the filtering applications(s). In this architecture the twin protocol stacks used to support the protocol break execute as distinct processes on the same instance of ProvenCore, but in two separate security domains on each side of the filtering application(s) as displayed in **Fig. 2**.

With traditional firewalls we had to cope with a very error prone TCB with a large attack surface, not surprisingly inadequate to meet the highest level of security. With this new kind of filter, we are relying on a bullet proof formally proven TCB, which in addition can be proved to exactly implement the intended filtering function. Non-surprisingly such a formally proven TCB can be brought to the very highest levels of security.

But there is more to it. Even with a bullet proof filter there is still the problem that we might be forced to authorize potentially damaging commands (i.e. it is very likely that we have to accept as part of the filtering security policy some commands that are dangerous but necessary). So the remaining problem is not about tampering with the filter (or the security policy) but with the fact that some valid commands may be used to attack the receiving side. Going back to our artificially simple “set on”/“set off” example of a filtering security policy illustrates in an obvious way the fact that the attackers have almost no degree of freedom left to perform an attack on the receiving side. The only commands that can be sent are “set on” and “set off” as planned and the filtering application will leave absolutely no degree of freedom in the way any of them can be expressed. The situation would be exactly the same for more complex and realistic filtering security policies: the only degree of freedom left is indeed the one allowed by the filtering policy itself. But the commands that are defined as being acceptable by the filtering security policy could be dangerous by themselves. For example, most embedded devices will need a "firmware_update" command to manage the firmware update process for the

whole platform. For this reason, it is usually also important to make sure that incoming commands have not been tampered with and have been issued by authorized and trusted persons. In other words, it is necessary to add proper authentication, and also guarantee the integrity and potentially the confidentiality of the commands. Guaranteeing these security properties is typically the role of a proper VPN. Here we propose to integrate a VPN application that can be brought to the same level of security as the filtering application(s). This will give the simplified architecture presented in **Fig. 3**.

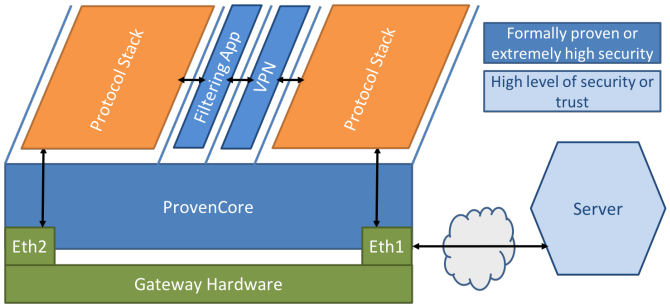


Fig. 3. Proposed Architecture, with Authentication

Using a proper highly secure VPN thus further reduces the attack surface and shows the benefit that can be obtained by the use of these new generation of filters. Our artificially simple filtering security policy makes it easy to see that an attacker would have only one degree of freedom left: the possibility of (either) slowing down (or theoretically accelerating although this would be much harder) the reception of ingoing commands. Attackers would have no other degree of freedom and thus the attack surface for performing any attack would be almost nil. Here the fact that TCB is formally proven and can be brought to the highest levels of security is key. It allows the filtering application itself to be brought to the highest level of security and we believe that such a possibility is a real breakthrough in the firewalling/filtering world.

3.2 A Practical Implementation

In practice, the architecture presented above can be easily implemented on an ARM processor using the architecture presented in **Fig. 4**.

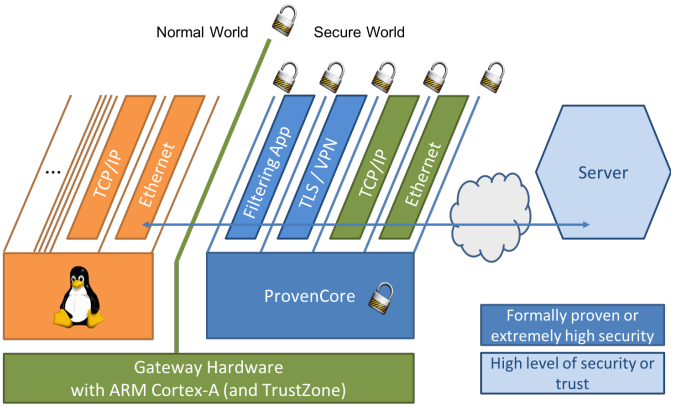


Fig. 4. Practical Implementation

Now the same benefits can be achieved for any kind of (stateful or stateless) filtering security policy. Another significant advantage is that this can be achieved without any impact on the bill of materials and therefore at very little cost. Therefore, such filters are not only much more secure than existing ones, but this architecture is applicable to cost-sensitive devices sold in large volumes. The only costly investment was the design, implementation and formal proof of the security of ProvenCore, an investment which has been done once and for all and can benefit to the huge volumes of compatible devices from various market segments. Depending on the situations these filters can be used to replace existing filters or to complement them (to be put in sequence with another firewall or an IPS[6]).

4 Focus on the Filtering Application

The filtering application (named FilteringApp in figures) is specific to each application domain and security policy, but it can also be implemented and formally proven at little cost, using Prove & Run's formal language and dedicated environment, respectively named `Smart` and ProvenTools (and described in [18], section 3).

Classically, we decompose the filtering application in two main components:

1. a parser, that checks that commands are syntactically correct w.r.t the list of allowed commands, awaited arguments and options. This parser also translates the input string into a structured version, the Abstract Syntax Tree (AST),
2. a validator, that receives this AST and analyses it to check that semantic constraints of the security policy are met (e.g. some command options that cannot be used together).

[6] IPS : Intrusion Prevention System

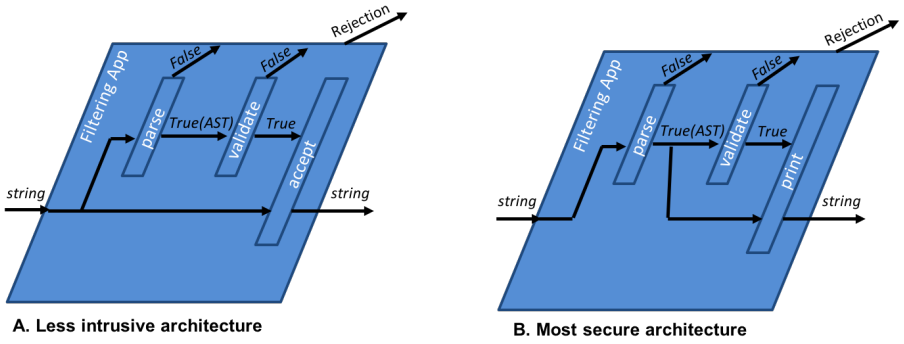


Fig. 5. Focus on the FilteringApp

These two components are described in more details respectively in sections 4.1 and 4.2. They can be combined following two distinct architectures, depending on the needs.

The first one, depicted in **Fig. 5.A**, is the less intrusive. When both filtering components accept the data, the filtering application outputs to the server the exact value of the input. This guarantees that, when the security policy is satisfied, the presence of the filtering application doesn't change at all the behavior of the overall system: it may only block some data, but never alters it.

The second architecture, depicted in **Fig. 5.B**, provides maximal security. It prints back the AST on its output. Thus, the data received by the server is as close as possible to the abstract version of the data on which the semantic checks have been performed. It can differ from the input in aspects which have no semantic impact, as the formatting (e.g. the number of spaces between a command and its argument). As these aspects are a degree of freedom, transmitting a normalized version is a plus.

4.1 Proven Parser Generation

The parser, the printer and the AST shape definition are automatically generated from the commands' syntax definition. Associated specifications are also automatically generated and proven, giving an extremely high confidence in these components. In addition of being extremely reliable, these components are inexpensive to develop and to maintain: the only piece of work is to settle and maintain the high-level specification of the syntax, which is used by ProventTools to generate the executable code, its specifications and proofs.

Let us consider a very simple example of security policy to provide a complete picture of the parser design and generation process. The filter application should accept only the following set of commands (named `shell_micro`):

- `ls` *`[-ltS]`* *`[filename]`*, where *`[-ltS]`* is an optional options block introduced by a dash followed by one or more options (among `l`, `t` and `S`); and *`[filename]`* is an optional argument indicating about which direct folder the information is requested
- `exit` with no option and no argument

The security policy additionally contains semantic constraints on `ls` options block, that are described in section 4.2.

The grammar of accepted commands is defined using a subset the Parsing Expressions Grammars (PEG) formalism [17], that we'll name PEG⁻. A PEG⁻ grammar is a set of rules, each rule consisting of a name, followed by a ←, then a definition body (where a special syntax # marks the presence of mandatory spacing):

```
command ← ls_cmd / exit_cmd

ls_cmd ← "ls" # ls_options? filename?
ls_options ← "-" ls_option+ #
ls_option ← "l" / "t" / "S"
filename ← ['a'-'z' 'A'-'Z' '_' '.']+ #

exit_cmd ← "exit" #
```

Fig. 6. Definition of `shell_micro` in PEG⁻

This formalism looks similar to context-free grammars (CFGs), but has a different interpretation: in PEG, the choice operator (`/`) selects the first match and the option (`?`) and repetition (`+`) operators are greedy (whereas all these operators are ambiguous in CFG). This interpretation guarantees grammar unambiguity by construction, makes suitable the expression of both lexical and syntactical rules in a unified way, and allows to generate a top-down parser which closely mimics the structure of the grammar.

Executable code generation
From a grammar *G*, we generate the following executable `Smart` code:

1. one type *AST_r* per rule *r* of the grammar, defining the shape of the produced AST. These types definitions are inferred as follows:
   - non-constant characters and strings are stored in standard library *character* and *string* types. For instance, the type generated to store the `filename` of **Fig. 6** is an alias to the string type: *type filename = string*.
   - a sequence is stored in a structure, with one field per non-constant sequence element, where repetitions and options are stored in standard library *list* and *option* types. For instance, the type generated to store an ls_cmd is *type ls_cmd = { ls_options_opt: option<ls_options>, filename_opt: option<filename> },* where "*ls_options_opt"* and *"filename_opt"* are the fields names, followed by a ":" introducing the associated field types.
   - a choice is stored in a variant[7], with one constructor per branch of the choice, allowing to store the non-constant content associated to the branch. For instance,

[7] that is, a disjoint union of types, each one being introduced by a constructor

the type inferred to store a `command` is: *type command = Ls_cmd(ls_cmd) | Exit_cmd*, where *Ls_cmd* and *Exit_cmd* are the variant constructors names[8], *ls_cmd* is the type of the content of values belonging to the *Ls_cmd* case, and values belonging to the *Exit_cmd* case have no content.

2. one parse function *parse_r* per grammar rule *r*:

$$parse\_r: string \rightarrow [True(string, AST\_r) \mid False]$$

It takes a *string* as input, and either successes (*True* case) and returns the unconsumed suffix of the input along with the AST node of type *AST_r* (built with the consumed prefix of the input); or fails (*False* case). `Smart` is very well suited to express, manipulate and specify such functions having named exit statuses (describing the internal execution case) associated to distinct sets of return values: here the unconsumed suffix and the built AST node are returned in the successful case, and no return value is available in the failure case.

These parse functions are unsurprisingly defined as follows:

- parsing a character, or a string (as a keyword or a pattern) is done calling library parsers defined once and for all;
- parsing a reference to another rule is done calling this rule's parse function;
- parsing a sequence consists in parsing each element in order and returning in a structure the aggregation of called parse functions results (which must all be successful);
- parsing a choice consists in trying in order each choice branch and returning a variant wrapping with the appropriate constructor the first successful parse result.

3. the entry point parser ($parse\_G: string \rightarrow [True(AST\_r_0) \mid False]$) which simply calls the parser of the first rule r_0 and checks that the remaining suffix is empty.
4. one printer function per grammar rule *r* ($print\_r: AST\_r \rightarrow string$) which prints into a string the content of the AST node, conforming to the syntax defined by the associated rule (including the constant content, as keywords and delimiters, which is not stored in the AST).
5. the entry point printer ($print\_G: AST\_ r_0 \rightarrow string$) which simply calls the printer associated to the grammar's first rule r_0.

Specification and proofs generation

More interestingly, three theorems are also generated along with their proof: the parser correctness and completeness, and a characterization of the AST content.

The formalization generated for the parser correctness and completeness relies on a shallow embedding of PEG$^-$ into a `Smart` library, written once for all, providing:

- a type *grammar* allowing to define a PEG$^-$ grammar in `Smart` (relying on a type *rule* which allows to define a PEG$^-$ rule's definition body);

[8] constructor names are in particular used to define some operations depending on which constructor case a variant value belongs to.

— a relation *recognizes(grammar, string)* which is true if an input *string* is conform to a given *grammar* (and thus defines the meaning of each PEG grammar's construct: sequences, ordered choices /, greedy repetition +, etc). This relation's definition relies on a more basic relation, *denote(rule, string, Success(string)|Reject),* which tells if a grammar rule succeeds in recognizing an input string (*Success* case, which takes in parameter the unconsumed suffix string) or if it rejects this input (*Reject* case).

Using the above described library, from a grammar *G*, we generate a `Smart` definition, named *grammar_G* and of type *grammar,* which embeds *G* in `Smart` (and consists of a set of objects *rule_r* of type *rule* embedding each rule). The correctness and completeness of the parser are expressed with respect to *grammar_G*.

Theorem (Parser Correctness):

$$\forall\, input, parse\_G(input) \Rightarrow recognizes(grammar\_G, input)$$

This theorem states that the parser is correct with respect to the grammar: all inputs accepted by the parser are conform to the grammar.

Proof. The generated proof uses lemmas generated (along with their proof) for each rule *r*:

$$\forall\, input, \forall\, suffix,\ parse\_r(input) = True(suffix, \_) \Rightarrow denote(rule\_r, input, Success(suffix))$$

Leafs of the proof tree use library lemmas stating the correctness of library parsers.

Theorem (Parser Completeness):

$$\forall\, input, \neg parse\_G(input) \Rightarrow \neg recognizes(grammar\_G, input)$$

This theorem states that the parser is complete with respect to the grammar: none of the inputs rejected by the parser is conform to the grammar.

Proof. Similarly to the correctness proof, this proof relies on lemmas stating the property on unit parsers (and on library lemmas lifting the property on library parsers):

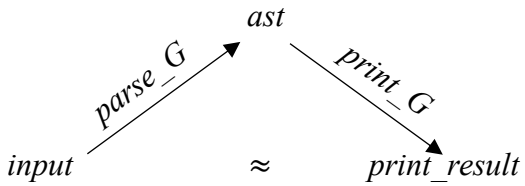
$$\forall\, input, \neg parse\_r(input) \Rightarrow denote(rule\_r, input, Reject)$$

Parser Correctness and Completeness theorems provide the guarantee that the parser component of the filtering application does the awaited syntactic filtering. As the parser is also responsible of building the AST processed by the semantic validator component, it is worth specifying the content of the produced AST. This is done thanks to the *print_G* function, which is thus useful for the sake of the specification, even if the chosen architecture doesn't use it (like in **Fig. 5.A**).

Theorem (AST Content):

$$\forall\, input, \forall\ ast,\ parse\_G(input) = True(ast) \Rightarrow print\_G(ast) \approx input$$

This theorem shows that the *ast* produced by the parser doesn't lose meaningful information contained in the *input* string, by stating that printing the *ast* produces a string which is equivalent modulo spacing (≈) to the parsed one. This can be better understood thanks to the following diagram:



Proof. Same pattern as preceding proofs, using the following expression of the property on all kind of AST nodes:

$$\forall\, prefix, suffix, ast\_r,$$
$$parse\_r(append(prefix, suffix)) = True(suffix, ast\_r)$$
$$\Rightarrow print\_r(ast\_r) \approx prefix$$

4.2 Semantic Validator

The semantic validator is implemented as a regular `Smart` program, allowing to handle arbitrarily complex security policies. In our `shell_micro` security policy, we could imagine the following semantic constraints on `ls` command options: absence of duplicates, and mutual exclusion between options `t` and `S`.

If the security policy is complex, we encourage the use of ProvenTools to also specify the awaited behavior and prove the correspondence with the validator implementation. For instance, in our toy example, the formal specification of what "absence of duplicates" means is simpler (thus less error prone) than the implementation of the associated validator, hence proving that the validator implementation is conform to the specification of the constraint is a plus.

Finally note that, as for syntactic constraints, a specific language could be designed to declare some kind of semantic constraints (as here constraints on lists: absence of duplicates, mutual exclusion, elements enabling other ones, etc.), allowing to automatically generate a proven validator.

5 Conclusion

In this paper we have shown why it is very difficult (or even impossible) to bring traditional firewalls and filters to the required level of security. We have proposed an approach that allows us to build new filters based on protocol breaks where the software TCB is made very simple and is just composed of a formally proven kernel, namely ProvenCore here (which is currently seeking a Common Criteria EAL7 certification), and a few security applications that can also be easily formally proven. The other parts of the software stack which normally compose a firewall, such as the drivers, the protocol stack, and the normal OS are here kept outside of the TCB. This is why such filters

can be brought to levels of security that only simple physical data diodes could previously meet.

Acknowledgments. The authors would like to thank Érika Baëna and Horace Blanc for their valuable contribution to the work presented in section 4.1.

References

1. D. Bolignano, “Proven Security for the Internet of Things,” in proceedings of the Embedded World Conference 2016, February 2016.
2. National Vulnerability Database. NIST. [Online]. Available: https://web.nvd.nist.gov/view/vuln/search [Accessed 15 Jan. 2016].
3. C. Miller and C. Valasek. "A survey of remote automotive attack surfaces". [Online] Available: http://illmatics.com/remote%20attack%20surfaces.pdf [Accessed 15 Jan. 2016].
4. M. Seaborn and T. Dulien, "Project Zero: Exploiting the DRAM rowhammer bug to gain kernel privileges,” 2015. [Online]. Available: http://googleprojectzero.blogspot.fr/2015/03/exploiting-dram-rowhammer-bug-to-gain.html. [Accessed 15 Jan. 2016].
5. "ADAC deckt Sicherheitslücke auf - BMW-Fahrzeuge mit 'ConnectedDrive' können über Mobilfunk illegal von außen geöffnet werden,” 2015. [Online]. Available: https://www.adac.de/infotestrat/adac-im-einsatz/motorwelt/bmw-luecke.aspx?ComponentId=227555&SourcePageId=6729. [Accessed 15 Jan. 2016].
6. C. Miller and C. Valasek, "Remote Exploitation of an Unaltered Passenger Vehicle". IOActive, Seattle, WA, Tech. Rep., 2015. [Online]. Available: http://www.ioactive.com/pdfs/IOActive_Remote_Car_Hacking.pdf. [Accessed 15 Jan. 2016].
7. M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom and M. Hamburg, "Meltdown," [Online]. Available: https://arxiv.org/abs/1801.01207 [Accessed 11 Jan. 2018].
8. P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz and Y. Yarom, "Spectre Attacks: Exploiting Speculative Execution," [Online]. Available: https://arxiv.org/abs/1801.01203 [Accessed 11 Jan. 2018].
9. D.A. Osvik, A. Shamir and E. Tromer, "Cache Attacks and Countermeasures: The Case of AES," in Pointcheval D. (eds) Topics in Cryptology – CT-RSA 2006. CT-RSA 2006. Lecture Notes in Computer Science, vol 3860. Springer, Berlin, Heidelberg.
10. M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, S. Mangard: "ARMageddon: Cache Attacks on Mobile Devices," in 25th USENIX Security Symposium (USENIX Security 16). Austin, TX : USENIX Association, August 2016.
11. C. Cohen, “AMD-PSP: fTPM Remote Code Execution via crafted EK certificate,” [Online]. Available: http://seclists.org/fulldisclosure/2018/Jan/12 [Accessed 11 Jan. 2018].
12. G. Beniamini, "Trust Issues: Exploiting TrustZone TEEs," [Online]. Available: https://googleprojectzero.blogspot.com/2017/07/trust-issues-exploiting-trustzone-tees.html
13. TLS/SSL Explained – Examples of a TLS Vulnerability and Attack, Final Part. [Online]. Available: https://www.acunetix.com/blog/articles/tls-vulnerabilities-attacks-final-part/ [Accessed 11 Jan. 2018].
14. When The Lights Went Out - A Comprehensive Review Of The 2015 Attacks On Ukrainian Critical Infrastructure. [Online]. Available: https://www.boozallen.com/content/dam/boozallen/documents/2016/09/ukraine-report-when-the-lights-went-out.pdf [Accessed 11 Jan. 2018].

15. Internet Security Threat Report, Volume 22, April 2017, [Online]. Available: https://www.symantec.com/content/dam/symantec/docs/reports/istr-22-2017-en.pdf [Accessed 11 Jan. 2018].
16. G. Beniamini, "Extracting Qualcomm's KeyMaster Keys - Breaking Android Full Disk Encryption," [Online]. Available: http://bits-please.blogspot.fr/2016/06/extracting-qualcomms-keymaster-keys.html [Accessed 15 Jan. 2018].
17. B. Ford, "Parsing Expression Grammars: A Recognition Based Syntactic Foundation". *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. (2004) doi: 10.1145/964001.964011.
18. S. Lescuyer, "ProvenCore: Towards a Verified Isolation Micro-Kernel" MILS-Workshop (2015) Available: http://www.provenrun.com/wp-content/uploads/2015/01/Prove-Run-ProvenCore-Towards-a-Verified-Isolation-Micro-Kernel.pdf [Accessed 03 Aug. 2018].