# Introduction



Figure (1): The primitive hut

## Objectives

## Innovation

As can be seen in Figure (1)

## Structure

# Index of Figures

1. *Primitive hut* from the frontispiece of Marc-Antoine Laugier’s 1755 second edition of *Esssay on Architecture*, illustration by Charles-Dominique-Joseph-Eisen.

# Requirements

## Runtime Requirements

## Messaging Node Requirements

## Network QoS Policy Enforcement Requirements

## Service Framework Requirements

# State of the Art

## Security in Runtime

In this document, we present an overview about the security in runtime component of the project. In first place, we will define the security goals we want to achieve. Then, possible attacks against those goals are explained. Finally, we present the state of the art on how current systems solve the referred problems, splitting the security runtimes in two distinct areas:

* Web Browsers
* Secure elements

In the Web Browsers section, we will present security mechanisms for JavaScript code protection in fully-featured environments (the web browsers themselves), while in the Secure Elements section, the state of the art on code security runtimes for systems featuring less functionality and computation capabilities but requiring tighter security requirements during its operation.

**Security goals:** The following points consist in specific security goals that the hyperty instances must achieve.

G1 - to protect the confidentiality and integrity of the state of hyperty instances. G2 - to protect the integrity of the code of hyperty instances. G3 - to enforce the security policies attached to hyperty code.

**Possible attack vectors:** Multiple attack types can be performed against the hyperty instances, at multiple levels (software, OS, hardware). We now present and detail the most well-known attack vectors occuring at each level.

*Malicious hyperty code:* E.g., by exploiting a bug in the hyperty runtime (HR), a malicious hyperty manages to escape from its sandboxed environment and escalates its privileges to gain access to a victim hyperty instance. The attacker can then read or modify sensitive data (break G1), modify the code (break G2), or modify the security policy (break G3) of the victim.

*Compromised browser:* E.g., the local user logs in to the computer, and attaches a debugger to the browser’s process. This gives the user full access to the browser’s state, including the HR’s. Therefore all G1, G2, and G3 can be violated. A similar attack can be performed by malware that subverts the browser’s security mechanisms and allows for an attacker to control the HR.

*Compromised operating system:* E.g., a kernel rootkit subverts the kernel, and gains access to the entire memory state of the browser process in which the victim hyperty instance is running. Just like in the previous cases, all G1, G2, G3 can be violated. Jail-breaking the kernel can bring similar consequences.

*Compromised hardware:* E.g., an attacker manages to gain physical access to the host computer and has the capability to extract secrets directly from the RAM memory or by probing the system bus. This attack can violate the security goals G1, G2, and G3.

### Web Browsers

#### Monolithic vs Modular Architectures:

Traditionally, commercial and open-source web browsers employed a monolithic architecture. This means that both user’s and web application’s data are combined into a single security domain, which brings serious performance/usability and security issues. On the performance/usability side, if a web application crashes during its execution, the whole web browser can be affected, harming the user experience. On the security side, if an attacker exploits an unpatched vulnerability in the browser while a user is using it, he may gain access to the whole user space, being able to execute code on behalf of that user and access its private sensitive information, like credentials.

Nowadays, web browsers evolved into modular architectures, in order to achieve privilege separation and overcome monolithic architectures’ flaws. This way, browser developers came up with multiple different architectures to achieve this separation between what is user’s property (credentials, preferences) and what is “web’s” property (applications’ code). Multiple techniques were employed in these architectures to achieve this separation:

* **Sandboxing:** In computer security, a sandbox is a security mechanism which allows untrusted programs to run within a trusted environment, without affecting its trustiness. This is usually done by restricting the resources (disk, memory, network) the untrusted software can access. An example is creating scratch memory and disk spaces where it can read/write and limiting the network capabilities it can use, in order to prevent the host environment from getting damaged. This is what Chromium browser [1] applies to separate the user and the web side in a modular architecture. It features two modules:
* A **browser kernel module** which acts on behalf of the user and is responsible for implementing the tab-based windowing system of the browser, storing user’s data as its preferences, bookmarks, credentials and cookies and also working as a middleware between the native operating system window manager and every instance of the second browser module, the rendering engine.
* The **rendering engine** provides the web application behaviour. It interprets and executes web content, serving calls to the DOM API. It is the unique browser part in contact with the untrsuted web content. Apart from that, it is also responsible for enforcing the same-origin policy between the user and a website he’s visiting.



image

Figure 1. Chromium sandbox scheme

#### Browser Extensions Security:

Browser extensions provide useful additional functionality to web browsers, such as facilitating the access to a website’s content or even as almost standalone applications running on the browser environment. However, these extensions often introduce serious security issues into both user’s browser and websites, because most of the times they’re written by well-meaning developers who are not security experts. Extensions can read and alter users’ bookmarks and preferences, websites’ content and perform requests over the network, many times on behalf of the browser user. Browser extensions are mostly written in JavaScript and HTML, and since JavaScript provides methods for converting a string to code (e.g. eval), an extension may be dangerous if misused. Typically, benign extensions face two types of attackers:

* **Network attackers:** Targeting end-users who connect to insecure networks (public Wi-Fi hotspots), these attacks consist in reading and altering HTTP traffic, in order to detect if an extension adds an HTTP Script - JavaScript file loaded over HTTP - to itself, and altering the code in such case.
* **Web attackers:** A malicious website can launch a XSS attack on an extension if the extension treats the website as trusted, possibly stealing the browser’s userdata, like credentials. This way, it can scale up to attack multiple websites within the same entry point.

According to [2], Google Chrome and its extension platform apply three mechanisms to prevent these vulnerabilities: \* **Privilege Separation:** Every Chrome extension has two types of components which run in separate processes: zero or more content scripts and zero or one core extension. Content scripts read and modify websites as needed. The core extension implements functionality not directly involving websites, like browser UI jobs or long-running background tasks. These two types of components communicate by sending structured clones over a trused channel. Each website that an extension communicates with, receives its own isolated instance of a content script, making content scripts highly bound to attacks. However, only the core extension is able to communicate with the Chrome extension’s API, reducing the risk that a content script is able to access the user data space. The architecture scheme of a Google Chrome extension is on Fig. 2. \* **Isolated Words:** This mechanism ensures that content scripts and websites have separate JavaScript heaps and DOM objects. Consequently, content scripts never exchange pointers with websites, protecting them against web attackers. \* **Permissions:** Extension developers have to specify the desired permissions in kind of a manifest file that is packaged with the extension. For example, the bookmarks permission is needed for the extension to be able to read and alter the user’s bookmarks. Only core extension can use permissions to invoke browser API methods, while content scripts are limited to interacting with the core extension and the website it is running on. This way, an extension is limited to the permissions its developer requested, so an attacker is not able to request new permissions for a compromised extension in runtime.



image

Figure 2. The architecture of a Google Chrome extension.

#### XSS Detection Techniques:

Cross-Site Scripting attacks are getting more common on the web, since they allow an attacker to get control of a user’s browser and execute malicious code (usually JavaScript/HTML) within the trusted context of a web application. This can result in the attacker being able to access any sensitive information associated to the application (cookies, session IDs, etc.). The study of XSS attacks can be split into two distinct categories, according to [3]:

* **Persistent/Stored attacks:** Occurs when a malicious user registers itself into a web application and posts a malicious JavaScript to the application, which, by its turn, save it into the application’s data repository, persistently. After that, if another user fetches the content uploaded by the malicious one onto his browser, and since this code is coming out of the trusted context of the web application, the user’s browser will allow the script to access any possibly sensitive resource it is willing to, overcoming this way the security imposed by the same-origin policy. Apart from stealing the user’s information, XSS attacks can also be used to redirect users to a malicious website which can then perform other distinct attacks within its context. A persistent XSS attack scheme is presented on Fig. 3.



image

Figure 3. Scheme of a persistent XSS attack.

* **Non-persistent/Reflected attacks:** Unlike the first type, reflected attacks do not persistently store malicious code in the web application data space. Instead of that, the content is automatically reflected back to the user through a third-party mechanism. For example, by using a spoofed email, an attacker can make a user click on a link containing malicious code, which will finally be interpreted by the user’s browser, but within the trusted context fo the web application. This type of XSS attacks is often combined with other techniques as phishing, and is the most common type of XSS attacks in web applications. Figure 4 shows a scheme of the architecture of a non-persistent XSS attack.



image

Figure 4. Scheme of a non-persistent XSS attack.

#### XSS (and other types) prevention techniques:

* **Analysis and Filtering of the Exchanged Information:**

This technique consists in defining a list of characters or tags which users are allowed to exchange with the web application, in the form of text inputs, uploaded files, etc. Then, a filtering process simply rejects everything that is not part of the list. Other approach, reported in [4], is having a proxy-server at the web application’s site in order to filter both incoming and outcoming requests. This filtering takes into account a set of rules defined by the application developers. Although, a simple use of regular expressions is able to evade both the referred methods and proxy-servers can rapidly become a performance bottleneck on the application deployment. Authors of [5] also suggested placing a proxy-server on the server-side of the application, but in order to differentiate trusted and untrusted traffic, driving each type to separate channels. This partitioning process uses Information Flow Control techniques to taint information and track it thenceforward. On other point of view, approaches seen in [6,7] propose the content filtering to happen at the client-side. On the one hand, authors of [6] try to achieve the prevention of XSS attacks by blacklisting links embedded within the web application’s pages, making them unavailable for the client. However, the authors say this approach can only detect basic XSS attacks based on the violation of same-origin policy. On the other hand, authors of [7] present another client-proxy solution that is intended to detect malicious requests reflected from the attacker to the victim (non-persistent XSS attacks). If such a request is detected, the malicious characters are re-encoded by the proxy, trying to avoid the success of the attack.

* **Security Enforcement on the Web Browser Runtime:**

There are also other strategies which try to avoid the need for intermediate elements like proxy-servers by proposing startegies to enforce the runtime context of the web browser. In [8], authors propose an auditing system for the JavaScript interpreter of the Mozilla Firefox browser, which detects misuses on JS operations and take counter-measures to avoid violations on browser’s security. Other approach is also followed by [9], which presents the use of dynamic taint tracking on JavaScript code, in order to detect whether browser’s sensitive resources are going to be transferred to an untrusted third-party. In such case, the user is warned and can decide whether he allows or denies the transfer. Finally, the approach followed by [10] propose a policy-based management where a list of actions is embedded into the documents exchanged between the browser and the server. These actions help the browser to decide whether or not a script should be executed. Although, a lack of semantics in the policy-language and the restrictiveness of the approach due to the sandboxing-like mechanism are some of the drawbacks.

### Secure Elements

#### Java Card: Internet Computing on a Smart Card

In secure computing, a smart card is a typical card with a built-in computer chip. Until a few years ago, it was only used to produce credit and debit cards, whose information can only be accessed when in possession of the card itself and a PIN code. Due to the short information on how to communicate and program them, until a few years ago this useful technology wasn’t being used on computer security in general.

* **Hardware:**

This single-chip computer is an off-the-shelf **8-bit microcontroller** with added tamper-safe features. While most 8-bit microcontrollers can support at least **64 KBytes** of 8-bit memory, popular smart cards contain 4 to 20 Kbytes of memory, due to size constraints. The memory space of a smart card is divided into RAM, EEPROM and ROM. RAM is used to store temporary values when a program is running, while EEPROM is used to store sensitive data as an encryption key or the account holder info on credit cards. Finally, ROM is used to store the basic programs that run on the smart card. The single-chip computer is embedded in a plastic chip carrier, and both of them hold several tamper-resistant and tamper-detection features.



image

Figure 5. Java Smart Card scheme [11]

* **Software**

The paucity of 8-bit assembly language courses, books and software tools led engineers to break the smart card application bottleneck by building a Java virtual machine with its runtime support into a 12-Kbyte smart card. **Java** was the obvious answer for three reasons: \* Java brings smart card programming into the mainstream of software development \* Java “safe programming” security model based on a runtime interpreter is a nontrivial side benefit, due to its processor independence. A Java card can be deployed on multiple smart card models. \* Java interpreters were tested to the limit, holes had been found, and fixed

With this in mind, engineers concluded that Java could preserve the required security in the smart card operation, while allowed a more friendly and well-known programming approach. However, available memory was an issue when deploying such heavy language runtime like Java. Features like garbage collection and exceptions handling were not included in Java Card because of that.

* **Internet Computing with Java Smart Card**

Java Cards combine smart card’s identity-verification features with the Java “sandbox”, guaranteeing that only allowed applications run on the card and that applications are protected from each other.

#### Cloud of Secure Elements

Cloud of Secure Elements (CoSE) [12] is an emerging concept whose goal is to provide trusted computing resources to mobile and cloud applications. To reach that, it relies on an infrastructure composed by multiple secure microcontrollers, named Secure Elements.

CoSE, in a WEB-like paradigm, are meant to support Uniform Resource Identifiers (URIs) for users to locate the different secure elements and use their embedded resources. These resources usually target two service types: Near Field Communication (NFC) facilities for mobile applications and trusted cryptogrtaphic features for cloud applications.



image

Figure 6. CoSE architecture

* **Architecture**

A Cloud of Secure Elements has the following components, as Fig. 6 shows:

* NFC kiosks, typically deliver payment facilities
* Users with NFC-enabled devices or terminals needing trusted cryptographic resources
* Grid of Secure Elements (GoSE)
* Secure elements, with resources identifiable by URI
* Remote administration entities, performing management operations over applications and secure elements
* **Grid of Secure Elements (GoSE)**

A grid of secure elements is an Internet server hosting multiple secure elements. Each element may be plugged in through USB readers, hardware sockets or electronic boards. Communication may be achieved with RACS protocol (works over IP/TCP/TLS stack) and performs both the association between elements and unique identifiers and data exchange with secure elements.

# **Malicious Code on Java Cards: Attacks and Countermeasures**

Despite all the advantages on using Java language in smart cards, such as the absence of low-level memory vulnerabilities, Java Cards still have an open door for attacks through malicious code. This attack entry is possible because an on-card bytecode verifier (BCV) is optional on Java Cards, and those who don't feature it, are more open to malicious code that might damage other applets running on the system or even the platform itself.

### **Defenses against malicious code**

Here we present the different mechanisms for protection against malicious code actions present in Java Cards.

#### **Bytecode verification**

Bytecode verification of Java code guarantees type safety, and thus, memory safety. On normal Java platform, bytecode verification occurs at load time. Although, since Java Cards do not support dynamic class loading, this verification must occur at the time an applet is installed to the card. However, most Java Cards do not feature an on-card BCV and rely on a digital signature of a third party who is trusted to have performed bytecode verification off-card.

#### **Applet firewall**

The applet firewall is an additional defense mechanism present in Java Cards. The firewall performs runtime checks to prevent applets from accessing and/or altering data of other applets (concretely, in a different security context). For every object within an applet, the firewall records its context, and for any field or method accessed this context is checked. Only the Java Card Runtime Environment (JCRE) has unlimited permission, since it executes in root-mode, on a UNIX terminology.

### **Getting malicious code on cards**

#### **CAP file manipulation**

This is the easiest way of introducing ill-typed code on a Java Card. This can be achieved by editing a CAP (Converted APplet) file to introduce a type flaw in the bytecode and install it to the card. Although, this will only work for cards without an on-card BCV and with unsigned CAP files. In example, by changing a baload (byte load) opcode onto a saload (short load) one, will make the platform treat a byte array as a short array, and can potentially lead to accessing other applet's memory space.

#### **Abusing Shareable Interface Objects**

The shareable mechanism of Java Card can be used to create type confusion between applets without any direct editing on CAP files. Shareable interfaces allow direct communication between security contexts. Using this to create type confusion is pretty simple: Let two applets communicate through a shareable interface, but compile and generate CAP files for both applets using different definitions of the shareable interface, which is possible because the applets are compiled and loaded separately. This way we can achieve an attack like the CAP file manipulation but without ever touching the CAP file directly.

#### **Abusing the transaction mechanism**

The Java Card transaction mechanism is probably the tricliest aspect of the Java Card platform. It allows multiple byte-code instructions to be turned into an atomic operation, offering a roll-back mechanism in case the operation is aborted, either through card tear or calling an API method. Buggy implementations of the transaction mechanism in some cards tend to make it not behave as expected. When object references are spread around the code, by assignments to instance fields and local variables, it becomes difficult for the mechanism to keep track of all the references that should be nulled out. The root cause of the problem is that stack-allocated variables, such as short[] localArray are not subject to roll-back in the event of a programatically transaction abort (through API method call).

### **Dynamic countermeasures**

Now we present some dynamic runtime checks implemented by some VMs in order to prevent ill-typed code to damage the Java Card platform. These were verified by [13], by performing tests on multiple Java Card models of multiple manufacturers against the referred vulnerabilities.

* Runtime type checking
* Object (array) bounds checking
* Physical (byte size) bounds checking
* Firewall checks
* Integrity checks in memory

## **Automated Analysis of Security-Critical JavaScript APIs**

Current web applications usually rely on JavaScript in order to offer additional features like maps, widgets or social media content. Although, since these additions may manipulate a page Document Object Model (DOM), steal cookies or navigate on the page, untrusted third-party JavaScript may pose security threats to the hosting page.

A widely-used approach is to combine a language-based sandbox to restrict the capabilities of untrusted JavaScript with an API offered by the trusted code part to the untrusted one. This API encapsulates all security-critical resources and guarantees they are only accessed in a safe way.

Given this, the authors of [14] proposed ENCAP, a tool that verifies API confinement, analyzing the isolation level it can offer to the critical objects it is intended to protect. ENCAP relies on a context-insensitive and flow-insensitive static analysis method. It analyses the API implementation and generates a conservative Datalog model of all API methods. Also, they propose SESlight, an ECMA JavaScript-subset language which only allows a strict (syntactically and semantically verified) subset of the whole language to be used.

## Bibliography

[1] - [Barth, A.; Jackson, C.; Reis, C. and Team, Google Chrome. 2008. The Security Architecture of the Chromium Browser.](http://seclab.stanford.edu/websec/chromium/chromium-security-architecture.pdf)

[2] - [Nicholas Carlini, Adrienne Porter Felt, and David Wagner. 2012. An evaluation of the Google Chrome extension security architecture. In Proceedings of the 21st USENIX conference on Security symposium (Security'12). USENIX Association, Berkeley, CA, USA.](http://nicholas.carlini.com/papers/2012_usenix_chromeextensions.pdf)

[3] - [Garcia-Alfaro, J. and Navarro-Arribas, G. 2007. A Survey on Detection Techniques to Prevent Cross-Site Scripting Attacks on Current Web Applications., in Javier Lopez & Bernhard M. Hämmerli, ed., 'CRITIS' , Springer, , pp. 287-298 .](http://eprints.uoc.edu/research/bitstream/10363/605/1/JGA01.pdf)

[4] - [Scott, D. and Sharp, R. Abstracting application-level web security. 11th Internation Conference on the World Wide Web, pp. 396–407, 2002.](http://rich.recoil.org/publications/websec.pdf)

[5] - [Pietraszeck, T. and Vanden-Berghe, C. Defending against injection attacks through context-sensitive string evaluation. Recent Advances in Intrusion Detection (RAID 2005), pp.124– 145, 2005.](http://tadek.pietraszek.org/publications/pietraszek05_defending.pdf)

[6] - [Kirda, E., Kruegel, C., Vigna, G., and Jovanovic, N. Noxes: A client-side solution for mitigating cross-site scripting attacks. 21st ACM Symposium on Applied Computing, 2006.](https://iseclab.org/papers/noxes.pdf)

[7] - [Ismail, O., Etoh, M., Kadobayashi, Y., and Yamaguchi, S. A Proposal and Implementation of Automatic Detection/Collection System for Cross-Site Scripting Vulnerability. 18th Int. Conf. on Advanced Information Networking and Applications (AINA 2004), 2004.](http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=1283902&abstractAccess=no&userType=instima)

[8] - [Hallaraker, O. and Vigna, G. Detecting Malicious JavaScript Code in Mozilla. 10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS’05), pp.85–94, 2005.](http://www.cs.ucsb.edu/~vigna/publications/2005_hallaraker_vigna_ICECCS05.pdf)

[9] - [Jovanovic, N., Kruegel, C., and Kirda, E. Precise alias analysis for static detection of web application vulnerabilities. 2006 Workshop on Programming Languages and Analysis for Security, pp. 27–36, USA, 2006.](https://iseclab.org/papers/pixy2.pdf)

[10] - [Jim, T., Swamy, N., Hicks M. Defeating Script Injection Attacks with Browser-Enforced Embedded Policies. International World Wide Web Conferencem, WWW 2007, May 2007.](http://www2007.org/papers/paper595.pdf)

[11] - [Uwe Hansmann, Martin S. Nicklous, Frank Seliger, and Thomas Schaeck. 1999. Smart Card Application Development Using Java (1st ed.). Springer-Verlag New York, Inc., Secaucus, NJ, USA.](http://dl.acm.org/citation.cfm?id=555354)

[12] - [Pascal Urien. Cloud of Secure Elements Perspectives for Mobile and Cloud Applications Security. IEEE Conference on Communications and Network Security 2013 - Poster Session](http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6682733)

[13] - [Wojciech Mostowski and Erik Poll. 2008. Malicious Code on Java Card Smartcards: Attacks and Countermeasures. In Proceedings of the 8th IFIP WG 8.8/11.2 international conference on Smart Card Research and Advanced Applications (CARDIS '08), Gilles Grimaud and François-Xavier Standaert (Eds.). Springer-Verlag, Berlin, Heidelberg, 1-16. DOI=10.1007/978-3-540-85893-5\_1 http://dx.doi.org/10.1007/978-3-540-85893-5\_1](http://www.cs.ru.nl/E.Poll/papers/cardis08.pdf)

[14] - [Ankur Taly, Úlfar Erlingsson, John C. Mitchell, Mark S. Miller, and Jasvir Nagra. 2011. Automated Analysis of Security-Critical JavaScript APIs. In Proceedings of the 2011 IEEE Symposium on Security and Privacy (SP '11). IEEE Computer Society, Washington, DC, USA, 363-378. DOI=10.1109/SP.2011.39 http://dx.doi.org/10.1109/SP.2011.39](http://www-cs-students.stanford.edu/~ataly/Papers/sp11.pdf)

*should we also study these references: http://seclab.stanford.edu/websec/jsPapers/csf09-camera-ready.pdf*

## Projects

The WONDER project has enlightened some foundations paths to be followed in a post-IMS era dominated by Web technologies that reTHINK is currently exploiting. Notably, the novel Signalling On-the-fly (SigOfly) concept was conceived and successfully demonstrated to enable seamless interoperability between different WebRTC service domains. reTHINK Protocol On-the-fly concept extends WONDER, the Signalling On-the-fly concept to any other service domain where needed protocol stacks can be executed in a Web Runtime.

The WONDER Library used to validate SigOfly concept can be used in reTHINK as a good starting point to design and implement reTHINK runtime APIs and reTHINK Javascript framework.

## Runtime

A very comprehensive analysis and evaluation of existing web runtime solutions was performed.

In order to evaluate the possibility to modify native implementations of WebRTC engines, Ericsson OpenWebRTC and Google WebRTC.org solutions were considered. OpenWebRTC is a promising modular WebRTC implementation based on popular GStreamer multimedia framework open source solution. Unfortunately, OpenWebRTC is not much supported by Ericsson lacking required documentation to let it be adapted to fulfil reTHINK new requirements. Google WebRTC.org solution is the reference implementation of WebRTC specification providing all APIs defined in the standards. However, the effort required to change it to fulfill reTHINK requirements is estimated to be very high. On the other hand, having an extended version of an existing WebRTC implementation would require the user to install a new reTHINK Browser. For all the above reasons, it was decided to re-use existing native implementations of WebRTC engines without modifications.

Javascript engine solutions were evaluated to analyse the possibility to adapt them in order to fulfill reTHINK runtime requirements, notably in terms of security (sandboxing). The V8 JavaScript Engine is an open source JavaScript engine developed by Google for the Google Chrome web browser. It has since seen use in many other solutions and it is considered the most powerful Javascript engine in terms of features and performance. It has mechanisms to facilitate its extension with new features but lacks required mechanisms for sandbox creation. One evaluated alternative, is to use nodejs that runs on top of V8 as well as having nodejs inside Docker taking advantage of its management and security features. Both solutions fulfill reTHINK security requirements and will be considered for reTHINK runtime implementations that are not based on browsers.

Firefox OS is a good candidate to implement reTHINK runtime in mobile devices supporting this Operating Systems. It natively suports JavaScript and HTML APIs 5 (including WebRTC) as programming language, and a robust privilege model to communicate directly with cellphone hardware, and application marketplace.

Three WebRTC based Media Server solutions were evaluated. Jitsi Videobridge supports Selective Forwarding Unit (SFU) for multiuser video communication and it is based on XMPP architecture. Kurento, supports MCU/SFU Star topologies and a modular architecture to implement media processing services. Janus Gateway is a flexible and modular WebRTC gateway that can be used to deploy a full-fledged WebRTC gateway on a cloud provider or just a small nettop/box to handle a specific use case, looking at applications as pluggable modules that a client can connect to through this gateway. These solutions, are good candidates to support server side Hyperties providing media related services.

## State of the Art Summary

The reTHINK project describes a framework that provides solutions to manage real time communication capabilities. To implement this framework the project team tried to use the most suitable existing standards which provides compability which existing technoligies. Using consolidated and widely used standards also make the development more efficient since Open Source libraries can be used in the developments. Addtionally to well-known standards, the project team has also tried to find emerging standards which can be adapted for ReTHINK requirements. In those cases, a tradeoff analysis has been made to determine if the choice of a not consolidated standard is optimal in terms of cost of use due to the lack of existing libraries and projects which use them.

The IETF has been creating and promoting the Internet standards since 1986. The IETF is organized in a large number of Working Groups (WG) which works on specific areas. For ReTHINK project, the team has focused on standards delivered by several WG (namely Rtcweb, TRAM, HTTP/2 and Network). The Rtcweb WG has defined a set of RFCs (many of them are still drafts) which are used in WebRTC, it defines how WebRTC works on the wire. Many of the used protocols already existed but many of them were created ad-hoc to meet WebRTC requirements. Other RFCs are informational and hes been released to gather the WG knowledge in a formal way. The TRAM (TURN Revised and Modernized) working group is carrying out a modernization of the protocols used to transport real-time media over Internet which is the final function of ReTHINK framework.

HTTP/2 is the new version of HTTP/1.1 which has been used in the web for the last 16 years. It provides a new low level design to optimize current Web applications keeping the semmantic of HTTP/1.1 which is still valid. HTTP/1.1 has been historically transported over TCP, however to take advantage of all the new features of HTTP/2 a new transport protocol build over UDP has been designed: QUIC. HTTP/2 draft is based on SPDY but it includes new features and will soon become a definitive RFC. The draft belongs to the HTTP WG. QUIC was developed by Google but it has been recently become an IETF Draft taking over the last changes in the protocol until close the defintiive RFC. HTTP/2 over QUIC has been considered as an alternative for messaging in the ReTHINK framework as it is optimized to be used over wireless connection and minimizes the delay in every communication.

The IETF is in charge of standarizes all the protocols on the wire in Internet. In turn, the W3C (WWW Consortium) is the main international standards organization for the World Wide Web. It standarizes how the browser behave (e.g. WebRTC 1.0 API exposed by the browsers) and and the lenguages (e.g. HTML and Javascript) which can be executed by a standar browser. It is main role is to promote and homogenize the evolution of the Web. During the state of the Art research work we focused on the standards susceptible of being used by any element within the ReTHINK framework.

The WebRTC 1.0 API has been standarized by the W3C is the way in which a Javascript application interacts with the browser to establish real-time sessions with other WebRTC endpoints. A comprehensive knowledge of this API was necessary to make design decissions and to define the architecture and the data model of the framework.

A Community group has been created within the W3C to promote an alternative WebRTC API called ORTC (Object Real-Time Communications) which gives more control to the WebRTC developer making easier to implement some scenarios. There are still not implementations of ORTC in production-ready browser, however the features introduced by this standard which is likely to become the base of the WebRTC 2.0 API have been considered during the design phase.

Another relevant W3C API is the Push API which allows a push service to send "push messages" to a webapp regardless of whether the webapp is currently active on the user agent. This is specially usefull for webapps running on mobile devices where the webapp may need to receive a notification while the browser is not in foreground.

The use of another feature supported by browser called Service Workers has been already evaluated to be used to implement different parts of the Runtime environment. Despite the fact that this specification is still a Working Draft of the W3C it is already supported by the most important browsers. However, this is feature is not supported by server side Javascript-based runtime environment, it only can be used when the Runtime is executed by a browser.

There is another interesting W3C Draft called "Application Lifecycle and Events" which extends the Service Workers with APIs for managing the lifecycle of an application and associated events. This Draft allows web developers to create applications that manage the application lifecycle and react to system events e.g. email or VoIP application. However, this Draft has been not been adopted by many vendors so far.

In this section the standars released by the Open Mobile Alliance (OMA) were also reviewed. The OMA is a Mobile Operator driven industry forum for the definition of interoperable mobile service enablers. OMA defines APIs to offer functionalities and resources of Operator networks to developers. Amongst the API and protocols standarized by the OMA the team decided to reviewed those which are relevant for the project such as the Authorization Framework for Network APIs, the RESTful Network API for WebRTC Signaling, Quality of Service API and Notification Channel. The LWM2M/COAT protocol which was designed to be supported by constrained devices has also been considered as a suitable alternative to interact with the Registry and Discovery services.

Finally, a recent standar called Smart Device Template (SDT) and released by the HGI (Home Gateway Iniative) has been reviewed. It provides a framework to create a consistent representation of Smart Home devices. This makes easier the integration of new devices in Home Gateway or in the cloud being specially interesting to implement M2M within the ReTHINK framework.

# Specification

### Specification of Runtime component

#### [Runtime Architecture](runtime-architecture.md)

#### [Runtime Security Analysis](securityanalysis.md)

#### [Data Synch Communication Model](data-synch-model.md)

#### [Runtime Architecture Dynamic View](dynamic-view/readme.md)

#### [APIs](runtime-apis.md)

#### [Runtime Implementation](implementation/readme.md)

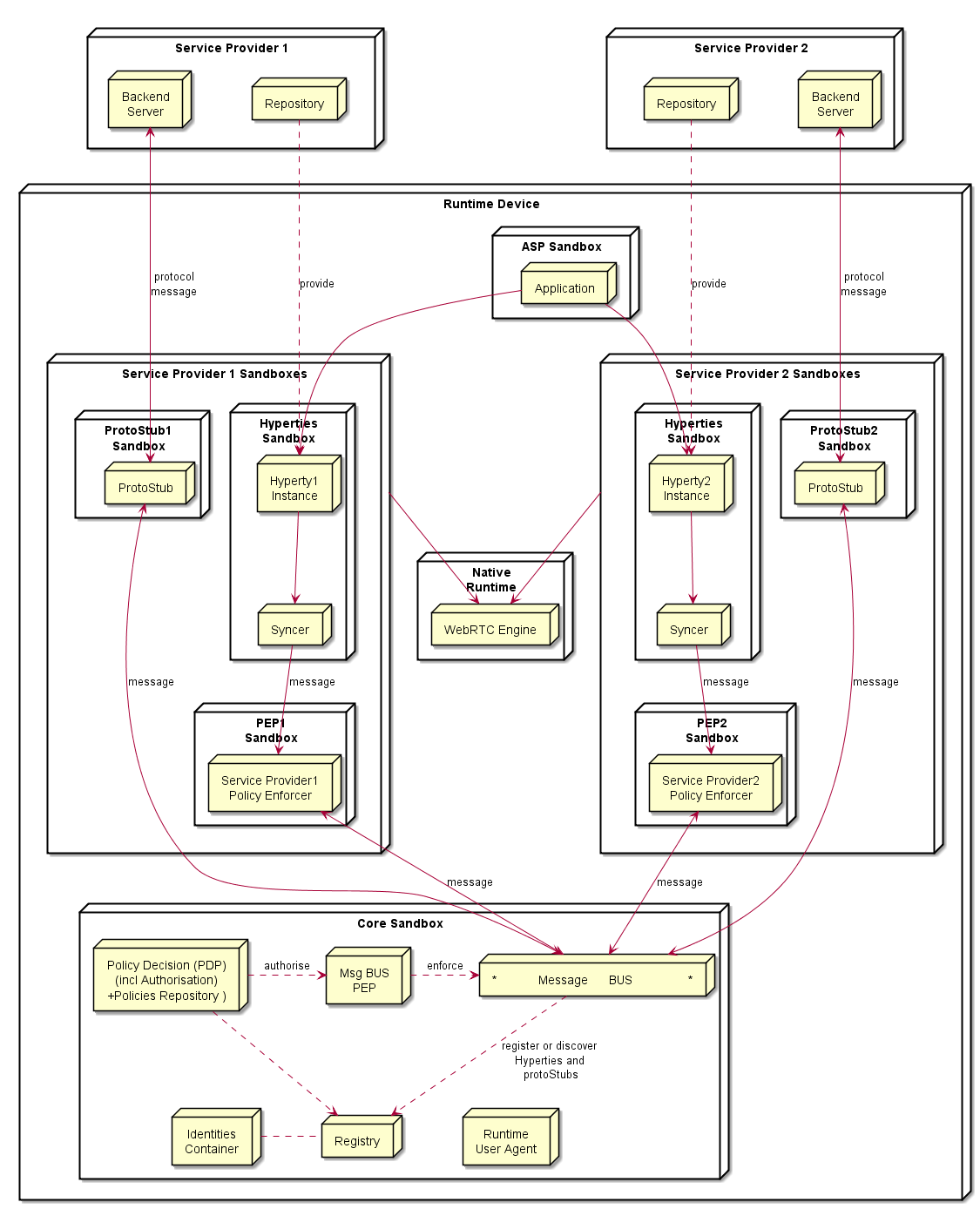
## Runtime Architecture

According to [ongoing discussions](https://github.com/reTHINK-project/core-framework/issues/41): \* one sandbox for the ASP providing the Application \* one sandbox per Hyperty Service Provider Domain that includes the Router/Policy Engine and associated protoStub \* the ProtoStub is used to communicate with Service Provider backend services. \* Hyperty instances communicates with Msg bus through Router/Policy Engine which may also act as a kind of firewall \* according to recommendations provided in the [runtime security analysis](securityanalysis.md), protoStubs and Router/Policy Engines execute isolated in independent sandboxes. \* to prevent cross origin attacks / spy, access to Message BUS may be subject to authorisation \* Different Points of Policy Enforcement: \* Policies enforced at Hyperty Sender Domain Router for outgoing messages \* Policies enforced to control the access to message Bus \* Policies enforced at Hyperty Receiver Domain Router for incoming messages

The different types of policies to be applied on these different points, namely in the Message BUS, requires further research to avoid performance overhead and potential conflicts. Initial thoughts: Message BUS PEP would enforce general access control policies that are agnostic of sender and target domains, or specific to the domain managing the device runtime (Core Runtime Provider). The access control to CRUD operations on standardised data objects like the Communication Data Object (eg only the owner of an active communication may apply delete or update operations on it) is a good candidate.

Below, it is depicted a functional architecture of the Runtime:

*Provide first a higher level architecture without showing detailed components. Distinguish internal APIs from external Interfaces which should be identified according to names defined in D2.2.*



Runtime Architecture

### Service Provider Sandboxes

According to Browser Sandbox model, each Service Provider Sandboxes executes components downloaded from the same Service Povider domain including Hyperties, protocol stubs used to connect and communicate with Service Provider Domain and PEP enabled Connector.

Functionalities to support Hyperty Communication through data object synchronisation are provided by the Syncer component based on Object.observer API. Details are discussed [here](https://github.com/reTHINK-project/architecture/blob/master/docs/datamodel/data-synch/readme.md).

#### Connector/Policy Engine

Handles communication between Hyperties and the local Message Bus, enforcing when needed valid Policies on this communicayion (e.g. authorisation policies) according to Service Provider domain policies. It also enforces access control policies to synchronised object (Object Monitor functionalities as proposed [here](https://github.com/reTHINK-project/architecture/issues/52)).

#### Protocol Stub

Protocol Stack to be used to communicate with Service Provider Backend Servers (including Messaging Server or other functionalities like IdM) according to Protocol on the Fly and codec on the fly concept.

### Core Runtime

#### Policy Decision Point and Message BUS authorisation

It provides Policy decision functionalities for the Service Provider Router sandbox according to Policies downloaded and stored locally when associated Hyperties are deployed. The possibility to consult Policies stored remotely should also be investigated. It also provides authorisation / access control to the Message BUS.

#### Message BUS

Supports local message communication between Hyperty Instances in a loosely coupled manner. Access to message BUS is subject to authorisation to prevent cross origin attacks / spy from malicious Hyperties.

See [postaljs](https://github.com/postaljs/postal.js)

#### Registry

Local Runtime Hyperty registry where Hyperty local addresses are registered and discoverable by other local Hyperties. The Runtime Registry should ensure synchronisation with Remote Domain Registry (to be provided by WP4)

#### Identities Containers

Contains Tokens that associates Hyperties with Users, it also provides Identity assertions. Something similar to [WebRTC IdP Proxy](http://w3c.github.io/webrtc-pc/#identity) but not limited to WebRTC.

#### Runtime User Agent

Manages Core Sandbox components including its deployment and update from Core Runtime Provider. It also handles Device bootstrap and the deployment of Hyperties and Protocol Stubs in the Runtime.

## Native Runtime

Functionalities that are natively provided by the runtime.

### WebRTC Media Engine

Provides the support for Stream communication betweeb Hyperties according to WebRTC Standards.

## Runtime APIs

### Message BUS

postMessage( message , callback)

### Hyperty

init( postMessage )  
report(message)

### Policy Enforcer

To set postMessage() function to be used by the Policy Enforcer to send messages usually the "MessageBUS".

setSender( postMessage )

To set postMessage() function to be used by the Policy Enforcer to receive messages usually the Hyperty or a Syncher. In case the resouce parameter is provided this postMessage() is only valid for messages containing the same resource url.

setReceiver( postMessage, resource )

To receive messages from the message BUS

postMessage(message)

### Syncher

*should we distinguish between Reporter and Observer syncher?*

To set postMessage() function to be used by the Syncher to send messages usually a "Policy Enforcer" but it could also be the MessageBUS.

setSender( postMessage )

Hyperty instance uses this function to provide the object to be changed by the (observer) syncher according to messages received. The Hyperty instance has previsouly used the *Object.observe* javascript api to set as an observer of this object

Promise <SyncObject> createAsObserver( receivedMessage )

To create a new object and ask another Hyperty instance to observe it. A Create Message will be generated and sent by the Syncher. Promise is used to handle Response messages to this object.

Promise <SyncObject> createAsReporter( resourceURL, schemaURL, toURL, dataObject?)

By default the events triggered by changes performed on this object by the Resporter Hyperty will trigger the synchronisation process. Otherwise the Hyperty instance should invoke a separate function, *addAttribute()*, *updateAttribute()*, *deleteAttribute()* defined below and afterwards invoke the *synch()* function to trigger the synchronisation process.

To add an attribute to object without triggering the synchronisation process:

addAttribute(resourceURL, attributeName, attributeValue)

To update an attribute without triggering the synchronisation process:

updateAttribute(resourceURL, attributeName, attributeValue)

To delete an attribute without triggering the synchronisation process:

deleteAttribute(objectId, attributeName)

To delete an Object:

delete(objectId)

To trigger the synchronisation process:

Promise synch(objectId)

To receive messages from other Hyperties that will be reported to the Hyperty:

postMessage(message)

### protoStub

disconnect( )  
connect( )  
postMessage(message)  
addListener( onMessage )

### HypertySandbox

postMessage(message)

### Runtime UA

loadHyperty( URL )  
download Hiperty from URL  
  
loadStub( URL )  
download Stub from URL  
  
discoverHiperty(applId, OSname, capability\_list)   
accomodate interoperability in H2H and proto on the fly for newly discovered devices in M2M  
  
checkForHipertyUpdate(URL)  
used by Applications or Hiperties to check for updates

### Registry

To register a new Hyperty in the runtime which returns the HypertyInstanceURL allocated to the new Hyperty.

HypertyINstanceURL registerHyperty( hypertySandbox.postMessage, hypertyUrl)  
  
registerStub( stub.postMessage, address )

### Identities Container

Functions to deal with assertions compliant with [WebRTC RTCIdentityProvider](http://w3c.github.io/webrtc-pc/#identity-provider-interaction)

IdAssertion generateAssertion( contents, origin, usernameHint )  
validateAssertion( assertion, origin )

### LHCB

getCurrentConnectivityStatistics()  
sendConnectivityStatisticsToBroker()

### Dynamic view of Runtime

*it should be aligned with* [*WP2 Dynamic View*](https://github.com/reTHINK-project/architecture/tree/master/docs/dynamic-view)

#### [Basics](basics/readme.md)

#### [Identity Management](identity-management/readme.md)

#### [H2H Communication](h2h-communication/readme.md)

#### [M2M Communication](m2m-communication/readme.md)

### Readme

#### [deploy runtime](deploy-runtime.md)

#### [deploy protocol Stub](deploy-protostub.md)

#### [deploy Hyperty](deploy-hyperty.md)

#### [Message BUS Routing](bus-msg-routing.md)

#### [Intra domain local communication](intra-local-comm.md)

#### [Inter domain local communication](inter-local-comm.md)

#### [Intra domain remote communication](intra-remote-comm.md)

#### [Inter domain remote communication](inter-remote-comm.md)

### Deploy runtime



Deploy Core Runtime Components in the Native Runtime

In case the device does not support the Hyperty Core Runtime components eg an existing browser like Chrome or a Network Node.js Server, they have to be deployed in the Device.

**Notes from 6th July H2H Comm Work Session:**

Runtime Core components should be as much as possible independent on the Runtime type. They should be deployed once and executed at the background. The next time the runtime is started there should be no need to download the core runtime again unless there is a new version. Runtime core components should be singletons (?) shared by different Apps and Hyperty instances. In order to support these characteristics for the Browser, Runtime Core components should be implemented with Web/Service Workers (FFS).

The Core Runtime is provided by a specific Service Provider that handles a central repository or catalog of the needed Core Runtime components.

This process may be triggered by the deployment of an Hyperty or Protocol Stub using some existing libraries like require.js. Such possibility has to be validated with experimentations.

### Deploy Protocol Stub



Deploy Protocol Stub

The protocol stub deployment may be triggered by the deployment of an Hyperty or by some attempt from a local Hyperty to communicate with a remote Hyperty running in the domain served by the protocol Stub. In this case the Runtime Registry would take the initiative to start the protocol stub deploy (FFS). Such trigger may take advantage of some existing libraries like require.js (to be validated with experimentations).

**Open Issue:** In the diagram above, the protocol stub is instantiated by the native Javascript engine as a normal javascript function/object, and in its constructor the registration process is performed. Another option, is to have in the Core Runtime, a protocol stub loader functionality (a Service/Web Worker?) that would handle the instantiation of the protocol stub and its registration in the runtime.

Protocol stubs are reachable through the Message BUS and not through domain routers (should we change the name). In this way it is ensured that all messages received and sent goes through the message bus where policies can be enforced and additional data can be added or changed including message addresses and identity tokens.

When registered, protocol stubs are associated with the domainURL they connect to.

Protocol stubs are connected by using credentials handled by the Core Runtime Identities Container. To be designed by the Identity Manager group.