MicroApp: Architecting Web Application for Non-Uniform Trustworthiness in Cloud Computing Environment

Yen-Chun Hsu, Yu-Sung Wu[*], Tsung-Han Tsai, and Yi-Pin Chiu
Department of Computer Science
National Chiao Tung University, Taiwan
head2568.cs96@gmail.com, hankwu@g2.nctu.edu.tw, smartPG@gmail.com, t3706408@gmail.com

Chih-Hung Lin and Zhi-Wei Chen
CyberTrust Technology Institute
Institute for Information Industry, Taiwan
chlin@iii.org.tw, zhiwei0115@iii.org.tw

***Abstract*—An increasing number of web applications are now hosted in cloud infrastructures such as Amazon Web Services. Cloud infrastructures generally lack a uniform guarantee on security, reliability, performance, and cost. A privately owned cloud infrastructure may be considered more secure but less performant than a third-party public cloud infrastructure. Infrastructures that span across geographical regions may further incur complications on the trustworthiness of infrastructures due to the varying power of jurisdiction. Application developers have to be aware of the non-uniformity of infrastructure trustworthiness when deploying applications in the cloud. We propose the MicroApp architecture that help address the difficulty in dealing with the non-uniformity. MicroApp splits a web application into multiple micro applications. Each micro application encapsulates a port of the code and data with the same level of security and integrity requirement. The micro applications will then be deployed to corresponding infrastructures that satisfy the respective requirements. MicroApp provides an RPC mechanism to allow control flows across micro applications. The architecture can be transparently applied to existing web applications and allows an application to effectively adapt to the cloud environment.**

Keywords—cloud computing, web application, remote procedure call, information flow, trust level

I. INTRODUCTION

The infrastructure-as-a-service or platform-as-a-service type of cloud services have greatly reduced the barrier of web application deployment. The elasticity and scalability as offered by the cloud service makes it even more appealing to host a web application on the cloud throughput its lifecycle. However, the infrastructure of a cloud service may not necessarily be trusted. A poorly managed datacenter may have reliability issues, that which could lead to data corruption and service unavailability. Also, a datacenter without adequate security protection may endanger both the confidentiality and integrity of sensitive information in an application. On the other hand, the cloud service may be quite competitive in terms of cost and performance, and not every portion of the application requires high reliability and/or high security. We may also assume that there are cloud service providers (possibly a private cloud), who do offer high security and reliability but at a much higher cost.

If we look at an application, we know that an application may employ proprietary design that should be kept confidential. An application may hold data that is deemed sensitive and should not be disclosed to an unauthorized entity or be placed at locations beyond the area of jurisdiction. As a result, web application developers who are interested in adopting cloud computing should ensure that these portions of the application be placed on infrastructures that can meet the security requirements. In this work, we propose the MicroApp framework to facility the deployment of web application in cloud computing environment of inhomogeneous security and reliability guarantees. Under MicroApp, a web application developer assigns security labels to application data and code segments. The MicroApp framework then treats an application as a collection of micro applications, each of which contains code segments and data bearing the same security requirement. The micro applications will then be deployed onto corresponding infrastructures with matching security and reliability guarantees. The MicroApp framework also provides a RPC mechanism to allow control flows across micro applications boundaries to operate as if the application was a singleton.

We implemented a MicroApp prototype supporting native Node.js applications (server-side JavaScript) and LAMP-based web applications. We conducted several experiments to evaluate the performance of the prototype system and demonstrate the feasibility of MicroApp architecture for real-world applications.

The rest of the paper is organized as follows: In Section II, we summarize the related works. In Section III, we present the system design of MicroApp. In Section IV, we describe some implementation details of the MicroApp prototype. In Section V, we present the evaluation results of the MicroApp prototype. In Section VI, we present the conclusion and our future work.

II. Related Work

Fabric[1] is a distributed application framework, which supports Java object model and employs a peer-to-peer network to connect application components. A developer can define security policies over the application data. Security policies are enforced by both compile-time check and run-time check with the help of information flow analysis.

In view of the inconsistencies in the trust relationship with third-party application component in mash-up[2] web application development, Hails[3] provides a framework that extends the original MVC (Model, View and Controller) model to the so called MPVC (Model, Policy, View and Controller). During runtime, Hails enforce the security policies via mandatory access control (MAC). Similar to Fabric, Hails uses information flow analysis technology to ensure data within an application are kept secure. Mobile Fabric[4] extends Fabric by considering the security of mobile code or client-side code in mash-up web applications.

Swift[5] partitions and moves a portion of the server-side application code to the client-side to attain balance between security and efficiency. Swift views the client-side as the untrusted domain and the server-side as the trusted domain. Swift uses information flow analysis to determine the portion of code that is not sensitive and allows the insensitive code to be moved to the client-side to improve application efficiency and responsiveness. Swift is based on the Jif[6] language. Jif is an extended Java language, which supports information flow check. Fabric and Mobile Fabric are also based on Jif.

The work mentioned above all use information flow analysis to ensure the security of application. Fabric's dissemination layer is similar in concept to the replication mechanism in MicroApp. Swift can automatically partition application code distribute them to the client-side or the server-side. MicroApp shares the same concept in this regard. Hails can only be applied to web applications designed with MVC model. In comparison, MicroApp has no restriction on the application model. Fabric requires manual modification to the client-side application source code, while MicroApp can automate code partitioning through information flow analysis. Both Hails and Mobile Fabric are designed to solve the problem of potentially untrusted third-party components. In comparison, MicroApp is designed to solve the problem of potentially untrusted infrastructure nodes. Also, in MicroApp, the communication and function call state management is fully transparent to the application developers. Therefore, developers do not need to manually readjust the control flows in the application. The application can be treated as a singleton during the development. Although Swift supports automatic partitioning of application code, it assumes a simple environment model consisting of an untrusted client-side domain and a trusted server-side domain. In comparison, MicroApp adopts a more general environment model, where there are multiple infrastructure nodes, each of which can be trusted or untrusted. With respect to security policies, Hails only

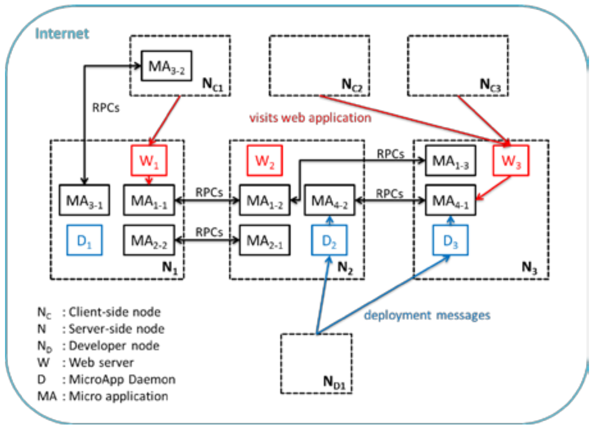


Fig. 1. MicroApp architecture

allows policies to be applied on the "Model" part of the MVC architecture. Jif, as employed in Fabric, Mobile Fabric and Swift, only allows security polices to be applied to the application data. In comparison, MicroApp allows security polices (confinement labels in Sec. III.A.1)) to be applied to both application source code and data.

III. Design of MicroApp

MicroApp enables a web application to be distributed across multiple infrastructure nodes, of which the trustworthiness levels may not be uniform. Formally speaking, under MicroApp, an application A will be restructured as a collection of n micro applications $MA_{1..n}$, each of which contains a portion of the application's source code and data. On the other hand, MicroApp assumes that there are m infrastructure nodes $N_{1..m}$, on which application A ,or the n micro applications, may be deployed on. In practice, the m nodes may be servers located at different datacenters or even client-side devices (e.g. smartphones, tablets, desktops, etc.). We assume that the m nodes are interconnected, either through direct network connections or via an overlay network.

Fig. 1 shows the architecture of MicroApp. The infrastructure nodes include server nodes N_1, N_2, and N_3, and client nodes N_{C1}, N_{C2}, and N_{C3}. There is a developer N_{D1}, from which the micro applications are deployed. Each server node N_i has a MicroApp daemon D_i and a web server W_i. The MicroApp daemon is responsible for the deployment of the micro applications on a server node. The web server serves as a portal for users of an application to invoke the application through the client-side web browser.

In Fig. 1, there are four applications A_1, A_2, A_3 and A_4, possibly owned by different parties. Each application is partitioned into a collection of micro applications. For example, application A_1 is split into 3 micro applications MA_{1-1}, MA_{1-2} and MA_{1-3}. They are deployed onto node N_1, N_2, and N_3 respectively. An infrastructure node may host multiple micro applications. For instance, node N_1 hosts 3 micro applications $MA_{1\text{-}1}$, $MA_{2\text{-}2}$ and $MA_{3\text{-}1}$.

Both the infrastructure nodes and the micro applications carry confinement labels (Sec. III.A.1), which define confidentiality and integrity levels for the nodes and the micro applications. The deployment of micro applications on the infrastructure nodes will have to adhere to rules on the confinement labels (Sec. III.A.2)). Control flows across micro applications are supported by MicroApp remote procedure calls (RPCs), which will be discussed in Sec. III.B.

As shown in Fig. 2, a micro application *MA* is made up of snippets of code and data bearing similar confidentiality and integrity requirements from the original application. A micro application also carries a copy of the *MicroApp Runtime Library*, a *MicroApp Session*, a *MicroApp RPC Request Handler*, a *MicroApp Runtime Server*, and a *MicroApp Parser*. *MicroApp RPC Request Handler* is used to process RPC calls between micro applications (Sec. III.B). *MicroApp Session* is a persistent storage for keeping the runtime states of a micro application that has to persist across RPC sessions. *MicroApp Runtime Server* implements the network communication layer is responsible for the invocation of the micro application. When an application needs to be re-partitioned, the *MicroApp Parser* of each composing micro application will be invoked to analyze, slice, and redistribute the application source code and data snippets to form a new formation of micro applications. A micro application may also be deployed on a client-side node (i.e. a web browser). A client-side micro application does not have *MicroApp Session*, *MicroApp RPC Request Handler* or *MicroApp Runtime Server*.

A. Partitioning Application Code and Data

The partitioning of an application's code and data is handled by MicroApp Parser. MicroApp Parser analyzes the application's source code and partitions the code into snippets according to the security and integrity levels specified in the confinement labels associated with the code. MicroApp Parser also uses information flow analysis to infer the security and integrity levels to deal with incomplete or inconsistent confinement labels. Code snippets with similar confidentiality and integrity levels will be grouped together

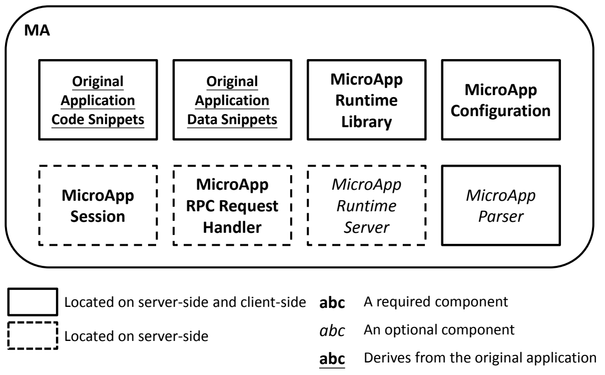

MA
Original Application Code Snippets
Original Application Data Snippets
MicroApp Runtime Library
MicroApp Configuration
MicroApp Session
MicroApp RPC Request Handler
MicroApp Runtime Server
MicroApp Parser
Located on server-side and client-side
Located on server-side
abc A required component
abc An optional component
abc Derives from the original application

Fig. 2. The components of a micro application

and then packaged into micro application(s).

MicroApp assumes that application source code is structured as a collection of *functions* and, for the case of object-oriented programming styled application, as a collection of *classes*. MicroApp Parser considers each global scope function and each global scope class as the basic unit for partitioning. Note that a class may have associated member functions. MicroApp Parser also considers a class member function as a basic unit and will not partition a class member function into different snippets.

```
function setGlobalA($v) {
  global /*i:1;c:1;g:FBSDK,APP*/ $a;
  $a = $v;
}
```
confinement label

Fig. 3. Confinement label for application code and data

1) Confinement and Trust Labels

An application developer may specify the security and integrity level requirements for each code statement in an application's source code. This is achieved by marking the security and integrity level in the associated confinement label as shown in Fig. 3. A confinement label may also be associated with an application's data. A confinement label is a tuple $\{i, c, g\}$, where i corresponds to the integrity level, c corresponds to the confidentiality level, and g corresponds to the group association. The integrity level and confidentiality level are numeric values. The group association is a set of strings (separated by commas). The group association may be omitted, in which case a default group association will be used.

Each infrastructure node has an associated trust label in the MicroApp configuration file. A trust label indicates the integrity level, the confidentiality level, and group association for a node.

$\forall$ snippet D, node N

D can be placed on $N \Leftrightarrow$ $\boldsymbol{C}(N) \geq \boldsymbol{C}(D)$ and
$\boldsymbol{I}(N) \geq \boldsymbol{I}(D)$ and
$\exists$ i s.t. $\boldsymbol{G}_i(N) \in \boldsymbol{G}(D)$

$\boldsymbol{C}(x)$ is confidentiality level of x
$\boldsymbol{I}(x)$ is the integrity level of x
$\boldsymbol{G}(x)$ is the group association of x
$\boldsymbol{G}_i(x)$ is the i-th group in x's group association

Fig. 4. The principle of classifying basic units for partitioning

2) Distribution of Micro Applications

Given the confinement labels and trust labels, MicroApp Parser will partition an application into code snippets and data snippets to be grouped into micro applications. A snippet *D* can only be placed on a node with sufficient security and integrity guarantee, and matching group association according to the rules in Fig. 4. Provided that the constraints on security, integrity, and group associated are satisfied, MicroApp Parser will attempt to maximize the performance of the application by load-balancing the distribution of micro applications on the nodes.

B. *MicroApp Remote Procedure Call (RPC)*

MicroApp RPC allows a micro application to communicate with another micro application (possibly on a remote node over the network). The communication is necessary because there can be control flows across micro applications, as such control flows would exist in the original singleton application. The RPC mechanism is used to carry function calls, remote object creation, remote object member function invocation, or internal MicroApp system coordination. The RPC mechanism is made up of two functional blocks: the RPC stubs and the call stack, which will be explained in details in the following:

1) *RPC Stubs*

Similar to traditional RPC mechanisms[7], MicroApp RPC is initiated through a *client stub*, and handled by a *server stub*. In MicroApp, the server stub is referred to as the *MicroApp RPC Request Handler*. There are two types of client stubs: One is "*G-stub*", and the other is "*O-stub*".

Each micro application has one *G-stub*, which is an object that provides the interfaces for invoking global functions and remote object creation. A *G-stub* is created at the beginning of the whole-life time of the object. Each created remote object has a corresponding *O-stub*. The methods of a remote object can be invoked via its *O-stub*. For performance, the *O-stub* also maintains a local cache of the properties of the remote object. The local cache in the *O-stub* only includes a subset of the remote object properties, which the node can carry under the security confinement constraints (Sec. III.A.2).

2) *RPC Call Stack*

A RPC call can be nested. The callee of a RPC call may invoke a further RPC call to another micro application, and etc. The RPC mechanism maintains the states of the ongoing calls in a structure called the *RPC call stack*. A *RPC call stack* is created when a micro application is started. A *RPC call stack* will remain active until the corresponding micro application terminates. As a result, a *RPC stack* also represents the whole-life time of an application. Each nested RPC call will cause the creation of a frame in the call stack, which is referred to as a sub-life time (of the caller micro-application).

An example of RPC call stack is shown in Fig. 5. When a micro application starts at node N_1, a whole-life time P_1 is also started for the micro application. When the micro application invokes a remote function or a remote object on node N_2, it uses MicroApp RPC to start a sub-life time P_2 on the remote node. Similarly, P_3 and P_4 are sub-life times created for further nested RPC calls. Each life-time will last till the completion of the corresponding *RPC call*.

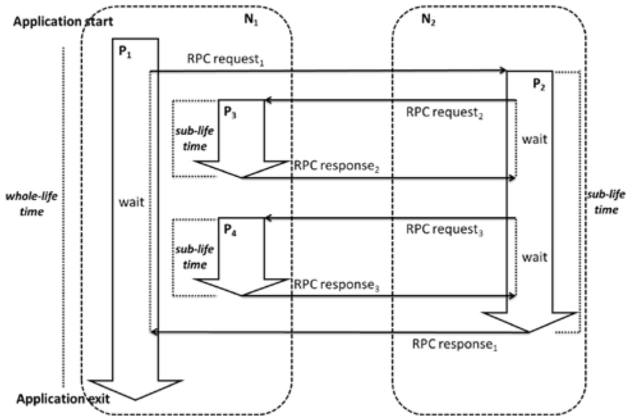


Fig. 5. RPC Call Stack

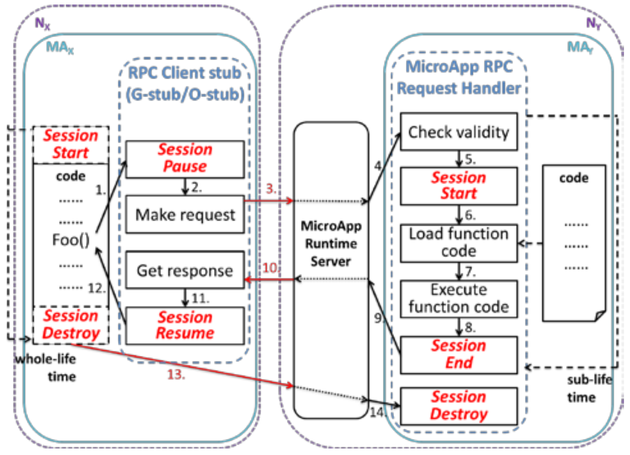


Fig. 6. Workflow of MicroApp RPC

3) *Workflow of MicroApp RPC*

Fig. 6 shows the workflow of MicroApp RPC. When micro application MA_X wants to invoke a remote function or a member function of a remote object at MA_Y, the *client stub* (i.e. the *G-stub* or the *O-stub*) at MA_X will send a MicroApp RPC request to the MicroApp Runtime Server of MA_Y. This corresponds to Step 1~4 in Fig. 6. At Step 5, MicroApp Runtime Server of MA_Y brings up the MicroApp RPC Request Handler of MA_Y to process the incoming RPC call. The MicroApp RPC Request Handler checks whether the target function exists and also checks whether the RPC caller is valid. It will then load the corresponding function (or object method) at MA_Y and executes it. After the function completes its execution, the RPC Request Handler will send a response indicating the status of the function call back to MA_X. MA_X can then process the returned status and continue execution.

4) MicroApp Session Mechanism

The RPC call stack is maintained by keeping call states of each micro application in a data structure called MicroApp Session. Each micro application has a corresponding MicroApp Session, which contains the states of local variables, global variables and constant values of the micro application during its whole-life time. As shown in Fig. 6, a session is created through *Session Start*. Before making a RPC call, the variable and constant values of the caller micro application will be saved to MicroApp Session through *Session Pause*. Later, when the RPC call returns, the variable and constant values of the caller micro application will be loaded back from its MicroApp Session through *Session Resume. Session End* returns the RPC call status back to the caller. *Session Destroy* releases the space of a MicroApp Session.

In the following, we will explain how each type of variables are maintained in the MicroApp Session:

a) Local Variables

A local variable can only be accessed by other scopes via parameter passing (including function return value). For MicroApp RPC, it is sufficient as long as all the involved local variable can be passed across life times. We consider two types of function call parameters. The first type is a scalar value, or an array of scalar values. The second type is an object reference, or an array of object references. The scalar values can be passed directly. The object references cannot be passed directly as an object's class definition may not be accessible on the remote node. Instead, MicroApp Runtime will pass an *O-stub* for each object reference parameter. The remote micro application can use the *O-stub* to access the methods and properties of the object.

b) Global Variables

Unlike local variables, global variables can be accessed from any scope. MicroApp uses a container object to wrap the global variables and make them as the properties of the container object. The container object intercepts access to global variables and automatically synchronizes its content to the other micro applications that require access the global variables. The container object also ensures access to the global variable follows the constraints of the confinement. The container object is kept in the MicroApp Session at a *Session End* or a *Session Pause*. It will be restored when *Session Start* or *Session Resume* is triggered.

c) Constants

Constant values can be treated as global variables except that their values are immutable. We have to keep constant values in a MicroApp Session as well because a life time may use a constant value that was defined in a previous life time. MicroApp Runtime synchronizes the constant values in MicroApp Session at *Session End* or at *Session Pause*.

C. Compatibility with Existing Web Application Security Models

Same origin policy (SOP) [8] is the dominant security model employed by modern web browsers. When a micro application runs on the browser, it will be constrained by the same origin policy. Essentially, same origin policy restricts access to browser-side data based on the origin of the data. An origin is defined as the tuple of domain name, application layer protocol and the port number. The data include HTTP cookies, DOM object, frames, XHR (XMLHttpRequest)[9], and etc.

We consider two scenarios, where the same origin policy might cause incompatibility with MicroApp. One is that all the browser facing codes are placed in a single micro application. In this scenario, the application only has one origin from the perspective of the client-side web browser, so the same origin policy does not actually cause any issue.

The other scenario is that the browser facing code are scattered into multiple micro applications, which are further placed on multiple nodes. As a result, the application will have multiple origins from the perspective of the client-side web browser. For HTTP cookies, when a cookie is set by one of the server-side micro application, the browser-side micro application will forward the cookie to the other browser facing micro applications. This essentially merge cookies issued from all the server-side micro applications (under different origins) into a single logical origin. For cross-origin access, possibly due to a XHR[9], MicroApp will only allow those accesses whose origins are valid.

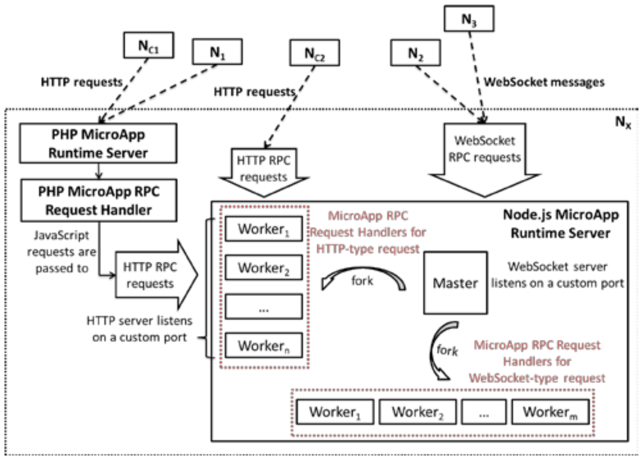
IV. IMPLEMENTATION

We apply the MicroApp architecture to the native Node.js[10] applications (server-side JavaScript) and the popular LAMP (Linux + Apache + MySQL + PHP) web applications which consist of server-side PHP and client-side JavaScript. The prototype supports web application developed in the languages of HTML, PHP and JavaScript. HTML is a markup language, for which we do not need to support RPC communication. For PHP, we have implemented a PHP MicroApp RPC Request Handler, PHP MicroApp Runtime Library, PHP MicroApp Session and PHP MicroApp Runtime Server. The PHP MicroApp Session adopts the PHP built-in session mechanism[11] to manage the storage, and the PHP MicroApp Runtime Server adopts the Apache web server with the *mod_php* module. For server-side JavaScript, we have implemented a JavaScript MicroApp RPC Request Handler, JavaScript Runtime Library for Node.js, JavaScript MicroApp Session in Node.js, and Node.js MicroApp Runtime Server. For client-side JavaScript, we have implemented a JavaScript Runtime Library for browser. The MicroApp Parser is implemented for both Node.js and browser. It can parse application code of HTML, PHP and JavaScript and generate corresponded micro applications and MicroApp Configuration in JSON format[12].

A. *MicroApp Runtime Server*

MicroApp Runtime Server support two communication protocols: HTTP and WebSocket. A WebSocket connection can be long-lived. MicroApp Runtime Server leverages it to avoid the expensive three-way handshakes of frequent short-lived HTTP connections. A MicroApp Runtime Server consists of a Master (parent) process and several Worker processes. Each worker is a MicroApp RPC Request Handler which receives MicroApp RPC request from the Master or the other nodes. The MicroApp Runtime Server for PHP uses Apache as the server to receive MicroApp RPC requests and the *mod_php* module as the engine to create a process running PHP MicroApp RPC Request Handler.

The MicroApp Runtime Server for Node.js uses the Node.js built-in module called "*cluster*"[13] to create a master process and several worker processes. It also uses the Node.js built-in module called "*http*"[14] to support HTTP protocol. We also install several third-party modules such as the "*websocket*"[15] module for supporting WebSocket[16] protocol. The architecture of Node.js MicroApp Runtime Server is shown in Fig. 7. N_X is a server-side node, to which the client-side nodes N_{C1} and N_{C2} and the other server-side nodes N_1, N_2 and N_3 can make JavaScript MicroApp RPC calls.



N3
NC1
N1
NC2
N2
HTTP requests
HTTP requests
WebSocket messages
NX
PHP MicroApp Runtime Server
PHP MicroApp RPC Request Handler
JavaScript requests are passed to
HTTP RPC requests
HTTP server listens on a custom port
HTTP RPC requests
WebSocket RPC requests
Worker1
Worker2
...
Workern
MicroApp RPC Request Handlers for HTTP-type request
fork
Master
Node.js MicroApp Runtime Server
WebSocket server listens on a custom port
fork
MicroApp RPC Request Handlers for WebSocket-type request
Worker1
Worker2
...
Workerm

Fig. 8. MicroApp Runtime Server for Node.js

B. *MicroApp Session Mechanism*

MicroApp Session is a persistent storage for storing RPC states. PHP has built-in session mechanism[11], so we directly utilize the PHP built-in session mechanism to implement the MicroApp Session for PHP applications.

In order to distinguish MicroApp Session from the default PHP session, the MicroApp Session uses its own session ID. The PHP built-in session mechanism allows only one session id at any time, so the MicroApp Runtime has to switch between the PHP session ID and the micro application session ID.

JavaScript (Node.js) has no built-in session mechanism, so we implemented a dedicated session mechanism for JavaScript-typed micro application. It serializes variables into JSON format strings and keep them to the files. Later, when the variables need to be restored, it will de-serialize them back from the files.

In order to determine when *Session Destroy* should take place, MicroApp Runtime Library has to detect the time when a whole-life time ends. For a PHP micro application, invocation of the destructor of a *G-stub* can indicate the end of a whole-life time. For a browser-side JavaScript micro application, MicroApp registers a handler for the event *beforeunload*[17] (supported by the browsers such as Chrome, Firefox, Internet Explorer, Safari, and etc.) to detect at the moment when an user is closing the webpage. This can be used to indicate the end of the whole-life time of a browser-side micro application. For a server-side JavaScript (Node.js) micro application, MicroApp registers a handler for the event *exit* as defined in the Node.js built-in library *process*[18] to detect the termination of a micro application process.

C. *Improving Performance of PHP MicroApp RPC through MicroApp Runtime Server Persistent Connections*

For PHP-based micro applications, MicroApp RPC uses HTTP protocol for communication as PHP web applications do not persist across page view by default. Persistent connections such as WebSocket are not natively supported. As a result, each RPC request requires creating a new connection. The involved three-way handshakes add significant overhead to the latency of RPC call. To improve RPC performance, we leverage WebSocket to establish a persistent connection between two micro applications to tunnel the HTTP-based RPC communications. A PHP micro application only needs to create new connections with the local MicroApp Runtime Server. The MicroApp Runtime Server will then tunnel the RPC calls to the MicroApp Runtime Server on the remote node via a persistent WebSocket connection. The remote MicroApp Runtime Server will then make local connections with the corresponding micro application to complete the RPC call. The workflow is shown in Fig. 8.

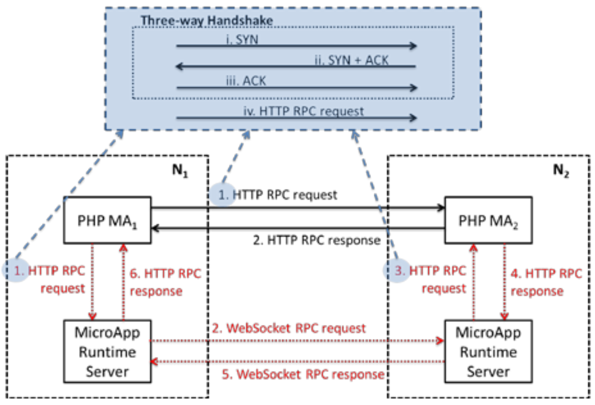


Fig. 7. Improving performance of PHP MicroApp RPC with MicroApp Runtime Server persistent connection

V. Experiment Results

We evaluate the effectiveness of the MicroApp architecture through a case study of applying MicroApp to a real-world application and looking whether the resulting micro applications meet the requirement of the given confinement labels. We also conduct a series of experiments to evaluate the performance overhead of running MicroApp. Our testbed consists of two server nodes N_1 and N_2 and one client node N_3. N_1 is equipped with two Intel(R) Xeon(R) E5620 processors and 65 GB memory and runs FreeBSD 9.0-RELEASE. N_2 is a host with one AMD Phenom(tm) II X6 1055T processor and 16 GB memory and runs Ubuntu 10.04.2. N_3 is equipped with one Intel(R) Core(TM) i5-2400 processor and 16GB memory and runs Windows 7.

A. MicroApp PHP Runtime Library Overhead

In this experiment, we evaluate the performance overhead of integrating MicroApp PHP Runtime Library with a PHP web crawler application PHPCrawl[19]. We compare the execution time, memory usage, and the resulting code size between the original PHPCrawl and the MicroApp version of PHPCrawl.

TABLE I. Code size comparison with PHPCrawl

	Declared Classes	Declared Functions	Incl. Files	Code Size (bytes)	# of Lines
PHPCrawl	27	290	38	273,225	8,321
MicroApp PHPCrawl	27 (45)	290 (520)	38 (49)	279,269 (445,185)	8,601 (12,342)

TABLE I shows the code sizes of the original PHPCrawl and the MicroApp version of PHPCrawl. PHPCrawl has 26 classes and 290 functions (including global functions, class constructors, and class member functions). The MicroApp version has 18 additional classes and 224 additional functions (including global functions, class constructors and class member functions) from the MicroApp PHP Runtime Library. The MicroApp PHP Runtime Library includes 11 files with 3741 lines of code and uses about 165,916 bytes of disk space. PHPCrawl by itself consists of 38 files. The MicroApp version of PHPCrawl also inherit the same 38 files. The MicroApp PHPCrawl version increases the code size by about 2.2%.

TABLE II. Characteristics of workloads

	Links	Received (bytes)
Sample 1	31	5,260
Sample 2	151	26,480
www.plurk.com (Plurk)	98	1,109,000
java.com (Java)	121	1,057,000

To measure the execution time overhead and memory usage, we ran PHPCrawl on N_1. The experiment began by invoking a web browser N_3 to visit PHPCrawl on N_1 and then directed PHPcrawl to start web crawling. The overhead of network communication was not considered because the traffic between N_1 and N_3 was negligible.

To measure the runtime performance overhead of MicroApp, we directed PHPCrawl to crawl two sample webpages and two real-world webpages. The number of the links and the amount of bytes received for each webpage are shown in TABLE II. The field "Links" represents the number of html links traced by PHPCrawl. The fields "Received" represents the total bytes received by PHPCrawl during the crawling. The results for Plurk and Java websites were averaged because both webpages contain dynamic contents.

We used both the original PHPCrawl and the MicroApp version of PHPCrawl to crawl each of the 4 websites and look at their respective execution time and memory usage. We used the PHP built-in *microtime*[20] function to measure execution time and used the *memory_get_peak_usage*[21] function to measure memory usage. Because PHPCrawl does use any PHP session, we also conduct an experiment, where the MicroApp session ID switching (Sec. IV.B) is turned off.

TABLE III. Execution time of PHPCrawl

Work-load	*# of func. calls*	*# of objects created*	Execution Time (secs)		
			PHPCrawl	**Micro-App PHPCrawl**	**MicroApp PHPCrawl without session ID switching**
Sam-ple 1	***1211***	***141***	7.61	30.46 (400%)	10.75 (141%)
Sam-ple 2	***5771***	***621***	35.86	417.85 (1165%)	51.66 (144%)
Plurk	***16231***	***4044***	71.52	6014.48 (8535%)	153.98 (215%)
Java	***15626***	***4177***	91.73	6829.66 (7446%)	140.5 (153%)

1) Execution Time Overhead

TABLE III shows the execution time of running the original PHPCrawl, running the MicroApp version of PHPCrawl, and running the no session ID switching MicroApp version of PHPCrawl. The field "# of function calls" shows the number of function calls (including object constructors and object member function calls) invoked during the execution. Note that the function calls in the MicroApp Runtime Library are not included. The field "# of objects created" shows the number of objects created. Again, object creations in the MicroApp Runtime Library are not included.

The execution time of MicroApp PHPCrawl is noticeably longer. The execution time overhead increases with the number of function calls and the number of object creations. A breakdown of the execution time overhead of MicroApp PHPCrawl is shown in Fig. 9. We can see the session ID switching constitutes most of the overhead. This is because each function call and each object manipulation will have to involve several session ID switching between the default PHP session and the MicroApp session. Each switching involves flushing the in-memory session data to the disk and loading back the session data from the disk to the memory. We can also see the percentage of session ID switching overhead grows with the number of object created. This is because more objects will have to be kept in the session, and that increases the cost per session ID switching. Going back to TABLE III, we can see that the execution time of MicroApp PHPCrawl without session ID switching is smaller than that of MicroApp PHPCrawl. However, this optimization is not generally applicable. It requires the target application not to use PHP session. A more general solution could be implementing a dedicated MicroApp Session mechanism for PHP application rather than relying on the built-in PHP session mechanism.

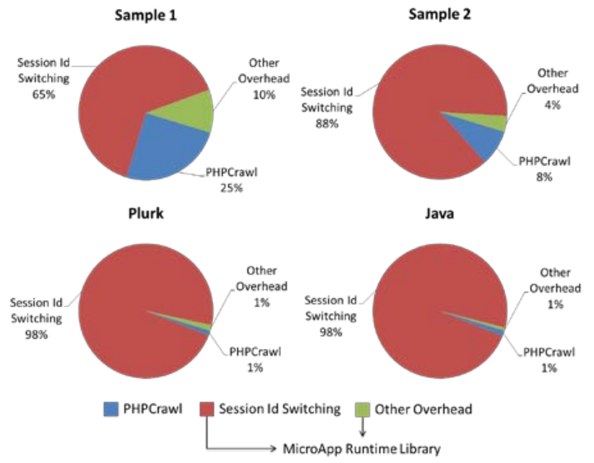


Fig. 9. Split of MicroApp PHPCrawl execution time

TABLE IV. Memory usage of original PHPCrawl vs. MicroApp PHPCrawl

Work-load	***# of objects created***	**Memory Usage (Bytes)**		
		PHPCrawl	**MicroApp PHPCrawl**	**MicroApp PHPCrawl without switch-ing session id**
Sam-ple 1	***141***	2,971,800	13,264,048 (446%)	12,571,016 (423%)
Sam-ple 2	***621***	3,472,264	38,939,032 (1121%)	36,104,824 (1040%)
Plurk	***4044***	11,103,680	112,342,096 (1011%)	116,652,888 (1050%)
Java	***4177***	4,712,424	117,379,760 (2490%)	100,547,480 (2134%)

1) *Memory Usage Overhead*

The memory usages of PHPCrawl and MicroApp PHPCrawl are shown in TABLE IV. PHP has garbage collection mechanism and the memory space of unreferenced objects will be automatically released. But the in the current implementation of MicroApp, a micro application has to retain all the objects in its MicroApp Session until the end of its whole-life time because it cannot be certain if an object in the MicroApp Session is referenced in an ongoing life-time of another micro application. Therefore, the memory usage of the MicroApp PHPCrawl is much higher than the original PHPCrawl. With more objects created, the memory usage overhead also grows higher.

To reduce the memory usage, we will have to implement a garbage collection mechanism for MicroApp Session. Whenever an object is no longer referenced by any other micro applications, the garbage collection mechanism can release the memory space of the object.

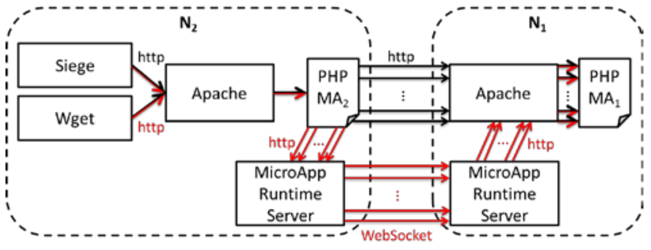


Fig. 10. The experiment environment of reducing communication overhead with MicroApp Runtime Server

B. *Reducing RPC Communication Overhead with MicroApp Runtime Server*

In Section IV.C, we mentioned that PHP MicroApp RPC is rather inefficient because each RPC call requires establishing a new TCP connection. It is a very time consuming process especially when the connection is made across a long-latency network such as the Internet. This can be improved by tunneling the RPC calls through a WebSocket persistent connection between the MicroApp Runtime Servers as shown in Fig. 10. In this experiment, we evaluate the RPC communication overhead of direct HTTP connections vs. WebSocket persistent connections.

As shown in Fig. 10, the test target is a simple PHP MicroApp consisting of two micro applications MA_1 and MA_2; MA_1 is essentially a computation server that takes requests from MA_2 and performs corresponding addition operations of two numbers. MA_1 runs on node N_1, and MA_2 runs on node N_2. The two micro applications make frequent RPC calls to each other, and most of the execution time is consumed by the MicroApp RPCs.

We used two benchmark programs to drive MA_2. The first is Siege[22] benchmark, which is a HTTP benchmarking tool. The second is GNU Wget. In each experiment run, we repeatedly used the benchmark tool to load MA_2, which in turns made a couple of RPC calls to MA_1. Corresponding, we created three measurement points corresponding to 8, 44, and 404 RPC calls between MA_2 and MA_1 as presented in TABLE V. The loading time is the end-to-end time it took to complete each corresponding number of RPC calls.

The experiment results in TABLE V indicates a steady improvement of at least 85% on the RPC communication latency due to the use of WebSocket tunnels between MicroApp PHP Runtime Servers. The improvement is expected as with WebSocket tunnel, only the first MicroApp RPC has to be penalized by the overhead of three-way handshake while the direct HTTP connection approach requires each individual RPC call to undergo a three-way handshake.

TABLE V. Total Loading Time

Benchmark		**Siege 2.68**			**GNU Wget 1.12**		
Number of RPCs		***8***	***44***	***404***	***8***	***44***	***404***
Total loading time (secs)	**Direct HTTP**	1.14	6.04	55.9	1.13	5.76	53.07
	WebSocket	0.18	0.88	7.58	0.18	0.86	7.63
Improved Percentage (%)		84.19	85.43	86.43	84.22	85.09	85.63

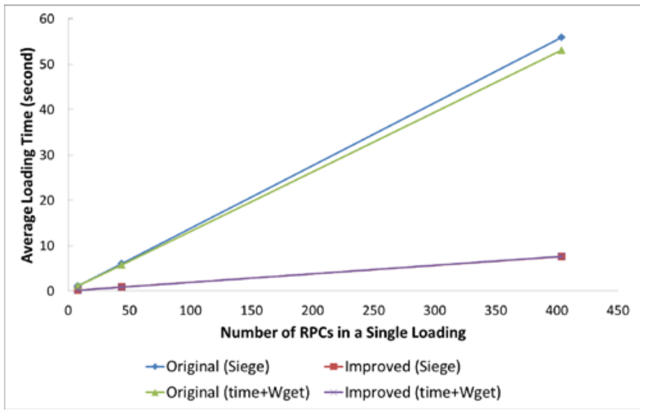


Fig. 11. Comparison of the loading times of MicroApp RPC with and without MicroApp Runtime Server WebSocket Tunneling

In Fig. 11, we can see that the average loading time grows linearly with the number of RPCs, which indicates the use of MicroApp Runtime Server WebSocket tunneling not only reduces the RPC communication overhead but is also a scalable mechanism in itself.

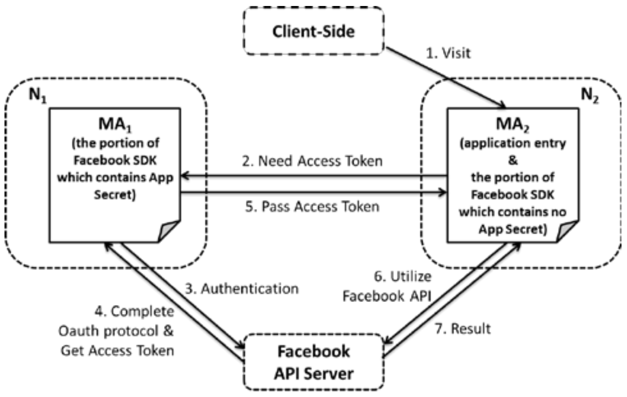
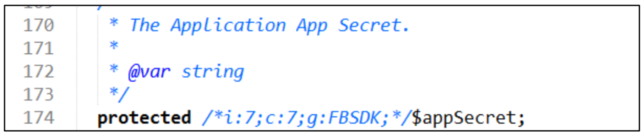
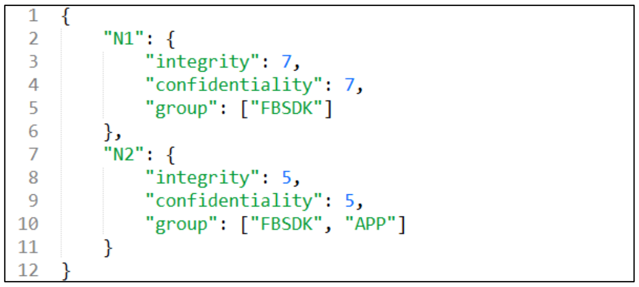

Client-Side
1. Visit
N_1
N_2
MA_1
(the portion of Facebook SDK which contains App Secret)
MA_2
(application entry & the portion of Facebook SDK which contains no App Secret)
2. Need Access Token
5. Pass Access Token
3. Authentication
4. Complete Oauth protocol & Get Access Token
6. Utilize Facebook API
7. Result
Facebook API Server

Fig. 12. An application using MicroApp Facebook SDK for PHP



```
 * The Application App Secret.
 *
 * @var string
 */
protected /*i:7;c:7;g:FBSDK;*/$appSecret;
```

Fig. 13. Confinement label for protecting app secret (in *base_facebook.php*)



```
{
    "N1": {
        "integrity": 7,
        "confidentiality": 7,
        "group": ["FBSDK"]
    },
    "N2": {
        "integrity": 5,
        "confidentiality": 5,
        "group": ["FBSDK", "APP"]
    }
}
```

Fig. 14. Confinement labels of the infrastructure nodes for MicroApp Facebook PHP SDK (in *microapp.json*)

C. *Case Study: Facebook SDK for PHP*

Facebook SDK for PHP[23] provides a set of PHP APIs for interacting with Facebook. A web application can use Facebook SDK to access Facebook functionality on behalf of a Facebook user. The application first needs to acquire authorization from the user through OAuth[24] protocol together with a predefined *app secret*. The application will then be given an *access token*, which allows the application to access Facebook functionalities on behalf of a Facebook user for the time period during which the *access token* is valid. Generally speaking, the *app secret* must be placed on a node that can be trusted by the application developer[25], while the application itself may be placed on a different, possibly less trustworthy, node. This requirement can be fulfilled by applying MicroApp on the Facebook PHP SDK.

As shown in Fig. 12, an application may be partitioned into two micro applications MA_1 and MA_2 by MicroApp Parser; MA_1 will be located on N_1, and MA_2 will be located on N_2. A user, who wants to use the application, will first visit the entry point of the application, which is encapsulated in MA_2 located at N_2. Later, when the user application code at MA_2 needs to access Facebook services on behalf of the user, it needs to acquire an *access token* from Facebook. The *access token* is provided by Facebook and requires an authentication process that depends on the *app secret* and user credential supplied through Oauth protocol. The *app secret* is hold by MA_1, so MA_2 will have to make a MicroApp RPC to request MA_1 to acquire a new access token from Facebook. MA_1 will initiate the OAuth protocol with Facebook to acquire an *access token* and then return the *access token* to MA_2. After receiving the *access token*, MA_2 can then interact with Facebook services with the user's identity until the token is expired.

As the *app secret* requires high confidentiality, the developer can put a confinement label on the variable that holds the *app secret* and set up the confinement labels for the nodes in the MicroApp configuration file. Fig. 13 shows an example of the confinement label for the variable *$appSecret* (line 174 in *base_facebook.php*), and Fig. 14 shows an example of the confinement labels for node N_1 and N_2.

VI. Conclusion and Future Work

We propose MicroApp, a web application architecture to address non-uniform trustworthiness of cloud infrastructures. MicroApp places the portions of application data and code that require high security and integrity assurance on more trustworthy nodes while allowing the remaining portions of the application to leverage as much of the resource provided the cloud for elasticity and scalability.

An application developer would use the confinement labels to specify the security and integrity requirements of each application code and data segment at the source code level. On the other hand, each node in the cloud infrastructure also carries a confinement label indicating the trustworthiness of the node in terms of the security and integrity levels that it can guarantee. MicroApp will partition the application into a collection of micro applications, each with homogeneous security and integrity requirements, and place the micro applications onto appropriate infrastructure nodes with matching confinement labels.

We implemented a MicroApp prototype supporting LAMP-based web applications and server-side Node.js applications. We have demonstrated that the MicroApp architecture can effectively protect sensitive application code and data and allow the application to pursue elasticity and scalability as enabled by the use of cloud infrastructures. MicroApp architecture is shown to be applicable to real-world applications with minimal modification efforts. The performance overhead due to session ID switching in PHP is significant. We will implement a dedicated MicroApp session mechanism for PHP to eliminate the time-consuming session ID switching process. We also plan to implement a garbage collection mechanism for the MicroApp Session to reduce the memory footprint of MicroApp Runtime Library.

Acknowledgement

This study is conducted under the “ICT Security and Data Protection Technology Development Project” of the Institute for Information Industry which is subsidized by the Ministry of Economy Affairs of the Republic of China. The study is also partially supported by the Ministry of Science and Technology of the Republic of China under grant number 101-2221-E-009-076-MY3.

References

[1] M. D. G. Jed Liu, K. Vikram, Xin Qi, Lucas Waye, Andrew C. Myers, "Fabric: A Platform for Secure Distributed Computation and Storage," in *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, 2009, pp. 321-334.

[2] *Mashup (web application hybrid) - Wikipedia, the free encyclopedia*. Available: http://en.wikipedia.org/wiki/Mashup_(web_application_hybrid)

[3] A. L. Daniel B. Giffin, Deian Stefan, David Terei, David Mazières, John C. Mitchell, Alejandro Russo, "Hails - Protecting Data Privacy in Untrusted Web Applications," in *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, 2012, pp. 47-60.

[4] M. D. G. Owen Arden, Jed Liu, K. Vikram, Aslan Askarov, Andrew C. Myers, "Sharing Mobile Code Securely with Information Flow Control," in *Proceedings of IEEE Symposium on Security and Privacy*, 2012, pp. 191-205.

[5] S. Chong, J. Liu, A. C. Myers, K. V. Xin Qi, L. Zheng, and X. Zheng, "Secure Web Applications via Automatic Partitioning," in *Proceedings of the 21st ACM Symposium on Operating Systems Principles*, 2007, pp. 31-44.

[6] A. C. Myers, "JFlow: Practical Mostly-Static Information Flow Control," in *Proceedings of the 26th ACM Symposium on Principles of Programming Languages*, 1999, pp. 228-241.

[7] *Remote procedure call - Wikipedia, the free encyclopedia*. Available: http://en.wikipedia.org/wiki/Remote_procedure_call

[8] *Same origin policy - Wikipedia, the free encyclopedia*. Available: http://en.wikipedia.org/wiki/Same_origin_policy

[9] *XMLHttpRequest - Wikipedia, the free encyclopedia*. Available: http://en.wikipedia.org/wiki/XMLHttpRequest

[10] *node.js*. Available: http://nodejs.org/

[11] *PHP: Sessions - Manual*. Available: http://www.php.net/manual/en/book.session.php

[12] *JSON*. Available: http://www.json.org/

[13] *Cluster Node.js Manual & Documentation*. Available: http://nodejs.org/api/cluster.html

[14] *HTTP Node.js Manual & Documentation*. Available: http://nodejs.org/api/http.html

[15] *Worlize/WebSocket-Node*. Available: https://github.com/Worlize/WebSocket-Node

[16] *WebSocket - Wikipedia, the free encyclopedia*. Available: http://en.wikipedia.org/wiki/WebSocket

[17] *window.onbeforeunload - Web API reference | MDN*. Available: https://developer.mozilla.org/en-US/docs/Web/API/window.onbeforeunload

[18] *process Node.js Manual & Documentation*. Available: http://nodejs.org/api/process.html

[19] *PHPCrawl webcrawler/webspider library for PHP - About*. Available: http://cuab.de/

[20] *PHP: microtime - Manual*. Available: http://php.net/manual/en/function.microtime.php

[21] *PHP: memory_get_peak_usage - Manual*. Available: http://php.net/manual/en/function.memory-get-peak-usage.php

[22] *Siege Home Page*. Available: http://www.joedog.org/index/siege-home/

[23] *Facebook for PHP SDK Reference - Facebook Developers*. Available: https://developers.facebook.com/docs/reference/php/

[24] *OAuth Community Site*. Available: http://oauth.net/

[25] *Login Security - Facebook Developers*. Available: https://developers.facebook.com/docs/facebook-login/security/#appsecret