2021 20th International Symposium on Distributed Computing and Applications for Business Engineering and Science (DCABES)

A Power-on Hardware Self-test Framework in Web-based OS

Hao Xu*, Long Peng*, Jun Ma, Shasha Li, Jie Yu+, Qingbo Wu

College of Computer, National University of Defense Technology, 410073, China

***Abstract*—The hardware status of mobile devices is crucial to users. If hardware fails and running application cannot perceive it, it may cause application to process wrong data and eventually make the system fall into a logically chaotic state, which may lead to severe consequences. Web-based Operating System (OS) is a mobile OS on which the application is developed by Web, but there is no mechanism for users to get hardware status in the OS. To address the above issue, we propose a hardware power-on self-test framework to deliver hardware status of Web-based OS to users. We design the framework by deep stacking three layers from bottom to top: (i) infrastructure layer, in which we register a new driver named self-test to collect the information about the hardware status from other drivers. (ii) web engine layer, in which we implement a Web API named CallCLibrary to read the status from infrastructure layer and pass it to web application; (iii) web application layer, which receives status information from lower layer and displays it to users. We implement the framework on a device equipped with Firefox OS, which is representative of Web-based OS and conduct experiments to verify the self-test framework and test the performance of CallCLibrary. Experimental results show that the proposed framework successfully captures various failure statuses of hardware and CallCLibrary outperforms native Web APIs of Web-based OS and JNI of Android.**

***Keywords*-Web-based OS; hardware self-test; device drivers; Web API;**

I. INTRODUCTION

The mobile devices are widespread nowadays, which puts forward an urgent requirements for mobile operating system (OS). The Web-based OS is a type of mobile OS and all applications in this OS are developed on Web technologies. Prevalent Web-based OSs include Firefox OS [1] [2] , Chrome OS [3] and Kai OS. Although Firefox OS has been stopped by Mozilla in 2016, it has not disappeared in the market nowadays. Kai OS, based on Firefox OS, is currently very popular in India. Data reports show that as early as the first quarter of 2018, Kai OS surpassed iOS to become the second largest mobile operating system in India, second only to Android. In 2019, the number of users of Kai OS has exceeded 100 million, making it the third largest operating system in the world. Web-based OSs are generally composed of three layers, i.e. application, web engine and infrastructure layers [4], as shown in Fig. 1.

- **Application layer**, which consists of web applications that are written in HTML5, CSS and JavaScript, is the user interface layer and provides human-machine interactions. This layer is also responsible for interacting with lower layers through exposed APIs.
- **Web engine layer** is composed of Web APIs [5], layout engine, JavaScript engine and security module. Web APIs provide the programmatic access to hardware functionality and expose it to web application. Layout engine parses the HTML and renders the content to the screen. JavaScript engine specializes in processing JavaScript scripts.
- **Infrastructure layer** is the bottom of Web-based OS, which serves as a bridge between web engine layer and the underlying hardware. This layer controls the underlying hardware and exposes hardware capabilities to Web APIs that are implemented in web engine layer. This layer contains Linux kernel, device drivers and HAL (hardware abstract layer).

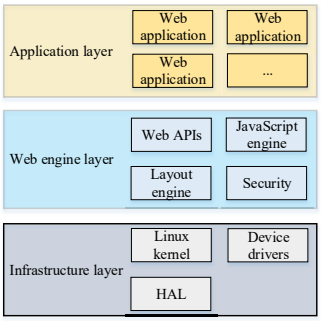

Figure 1: The architecture of Web-based OS.

System stability is an crucial indicator for an operating system, and whether the hardware works well or not is an important aspect of it. For instance, if GPS fails and device user fails perceiving it, the device may be in the wrong operating state and cause serious consequences, such as a map navigation system. At present, as far as we know, prevalent Web-based OS fails to provide hardware status information to users in the boot stage. To address above issue, we propose a power-on hardware self-test framework in Web-based OS, which can deliver hardware status to users. We implement the framework in three levels:

(i) In infrastructure layer we register a new driver named self-test to collect the status of other hardware drivers in the OS. To build the connection between the self-test driver and other hardware drivers, we utilize the notify mechanism in the Linux kernel. After the information collection, the self-test driver writes the hardware status into a file in the device.

* indicates equal contribution.

+ corresponding author. E-mail address: yj@nudt.edu.cn

2473-3636/21/$31.00 ©2021 IEEE
DOI 10.1109/DCABES52998.2021.00061

(ii) In web engine layer we implement a Web API named CallCLibrary. CallCLibrary provides the web application (HTML5) with the ability to invoke native code such as C/C++. We use CallCLibrary to read the hardware status file and deliver the message to the web application. CallCLibrary is implemented complying the rule of Web API and utilizes the callback Web API to achieve the message delivery.

(iii) In application layer, we create a web application and receive the message delivered by CallCLibrary Web API, then display the message on the login interface in the boot stage.

To validate the usability of our framework and the performance of CallCLibrary, we conduct experiments on a same smart watch device equipped with Firefox OS and Android respectively. Firstly, we simulate hardware failure and experimental results show that the proposed framework could successfully capture it and deliver it to users. Then we validate the performance of CallCLibrary. The exiting mechanisms for the application invoking the native .so library include native Web API of Web-based OS and JNI (Java Native Interface) [6]of Android, and we conduct a comparison experiment between them. The experimental results show that CallCLibrary outperforms the native Web API and JNI, with the average invocation time 2.7ms compared to 3.88ms and 3.38ms.

II. Background

In this section, we present background knowledge to be used in the following sections. Firstly, we introduce the notify mechanism in Linux kernel which is responsible for the message delivery between different drivers in our framework. Then we describe the content related to hardware driver in Linux kernel. Finally, we introduce Web API which construct the connection between the web engine layer and the application layer.

A. Notify mechanism

The various subsystems in Linux are independent, but sometimes a subsystem may be of interest to other subsystems. To build the connection between different subsystems, the Linux kernel present s the notify mechanism.

The notify mechanism is implemented by a linked list in the Linux kernel. Each node in the notification list is a data structure named `notifier_block`, a `notifier_block` contains a function that is to be executed when the `notifier_block` is matched. For instance, subsystemA uses the function `atomic_notifier_chain _register` to register a `notifier_block` in the notify list, when an event happens in subsystemB and subsystemB wants to notify subsystemA, the subsystemB invokes the `notify_call_chain` to traverse all the `notifier_block` in notify list and find the registered `notifier_block`, then invoke the function in it when matched.

B. Driver management in Linux kernel

The driver is a bridge between hardware and operating system [7]. Since Linux 2.6, a set of driver management mechanism has been introduced and most drivers in Linux use this mechanism. Device in Linux is represented as `platform_device` and driver is represented as `platform_driver`.

There is a Linux kernel at the bottom of Web-based OS. When the device equipped with Web-based OS power on, the hardware device and drivers in the system start to register. Driver would look for the device with the same name in bus and hardware device does the same. The `probe` function in the driver will complete the final job when driver matches device successfully. And the `probe` function would check the registration about the hardware device, when the hardware device fails, the function would print error information in the kernel.

While the `probe` function test hardware in kernel mode which the user program cannot attach, resulting in users cannot get the status information about the hardware. Besides, the `probe` function test a single hardware driver and the information about all drivers need to be collected.

C. Web API

Web API is an application programming interface for Web, which could manipulate the DOM (Document Object Model) using JavaScript. The DOM is a document model in the browser and represent the document as a node tree, each node in the tree represents an element. Web API generally utilizes JavaScript object as a carrier. For instance, the Web API `document.getElementById(id)`, which could get the element corresponding to the id, depending on the `document` JavaScript object.

III. Power-on self-test framework

A. Overview

In the boot stage of the device equipped with Web-based OS, the `probe` function is invoked to check the respective driver registration in the Linux kernel. To notify users about the hardware driver status, there are two works to be done: i) Collect the device driver status information from all the hardware drivers. ii) Deliver the status information to users.

However, it is not easy to accomplish these works. The device driver program runs in respective subsystem, and there is no information exchange between various drivers. On the other hand, device driver program runs in kernel mode, while the web application program, which is the interface between users and Web-based OS, runs in user mode. The bridge between web application and device drivers is to be built.

To complete the above two works, we implement our framework in three levels: infrastructure layer, web engine layer (CallCLibrary Web API) and web application layer, as shown in Fig. 2. Next, we will give detailed descriptions of the three components in the framework.

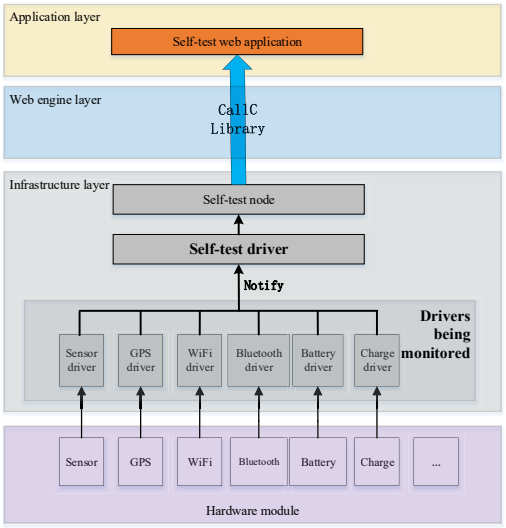

Figure 2: The architecture of the self-test framework.

B. Framework in Infrastructure layer

We register a new driver named self-test in the Linux kernel to collect information from other hardware divers through the notify mechanism and create a file to store the device status information. As a result, the information about hardware status is aggregated and the program in user mode could get the information from the device status file.

1) Register the self-test driver: The function `module_init` and `module_exit` exist in all the drivers of the Linux. `module_init` is the entry of the device driver, and when the device corresponding to the driver is deleted, the `module_exit` function will be executed. We create an entry function of the self-test driver and invoke the `subsys_initcall` instead of `module_init` to register the entry function, because the self-test driver needs to be registered before drivers to be monitored. The entry function contains function `platform _register_driver` which is used to register the self-test driver to bus.

2) Connect with other drivers through notify mechanism: The self-test driver is the crucial driver in the framework which is responsible for the information distribution and collection. On the other hand, the driver to be monitored delivers the hardware status message to the self-test driver. We implement the connection through `notify` mechanism in two aspects: the self-test driver and the driver being monitored .

Self-test driver. We use the `ATOMIC_NOTIFIER_HEAD` to initialize the notify list. Then we implement a `notifer_block` that contains the function `check_notifier_call`. The function `check_notifier _call` identifies the information delivered from drivers being monitored and set the device status file. Next we complete the implementation in the `probe` function of the self-test driver. We invoke function `atomic_ notifier_chain_register` to register the `notifer_block` mentioned above into the notify list.

Drivers being monitored. We encapsulate the function `notify_call_chain` into a function `notify_check_notifier` and add the function to the proper location of the `probe` function in the driver code. The function `notify_check_notifier` delivers "SUCCESS" or "FAIL" message to the notify list. For instance, in the `probe` function of compass driver file, we add the function `notify_check_notifier` as shown in Fig. 3.

```
kernel/linux-3.10.y/drivers/comip/sensor/akm09911.c
1467.int akm_compass_probe(struct i2c_client *client, const struct i2c_device_id *id)
1468.{
                        …
1479.   if (!pdata) {
1480.           dev_err(&client->dev, "Failed to allocate memory\n");
1481.           printk("9999===akm_compass_probe allocate memory failed \n");
1482.           notify_check_notifier(CHECK_NOTIFY_COMPASS_FAIL);
1483.           return -ENOMEM;
1484.   }
                        …
1486.   err = akm_parse_dt(&client->dev, pdata);
1487.   if (err) {
1488.           dev_err(&client->dev, "DT parsing failed\n");
1489.           printk("9999===akm_compass_probe DT parsing failed \n");
1490.           notify_check_notifier(CHECK_NOTIFY_COMPASS_FAIL);
1491.           return err;
1492.   }
                        …
1635.           dev_info(&client->dev, "successfully probed.");
1636.           notify_check_notifier(CHECK_NOTIFY_COMPASS_SUCCESS);
1637.                   return 0;
```

Figure 3: The code snippet added in the sensor driver.

When the `notify_call_chain` in the `notify_check_notifier` is invoked, the notify list will find the `notify_block` registered before and execute the function (i.e. `check_notifier_call`) in it to update the hardware status file.

C. Framework in web engine layer (CallCLibrary Web API)

In Web-based OS, web application is the interface between users and operating system, so the information in hardware status file need to be delivered to web application. We propose the CallCLibrary Web API to achieve that goal.

In this section, firstly we introduce the CallCLibrary Web API, which provide the ability to invoke the native .so library for web application. Next we present the specific application of the CallCLibrary on self-test framework.

CallCLibrary is a type of Web API, so we construct it complying the rule of Web API. The overall design of the CallCLibrary Web API is shown in Fig. 4. The implementation is divided into three parts: application layer, web engine layer and plugin (i.e. .so library).

1) CallCLibrary in application layer: In application layer, i.e., the web application, we construct a CallCLibrary JavaScript object. Three functions of CallCLibrary are `Init`, `Addlistener` and `Exec`. `Init` is used to

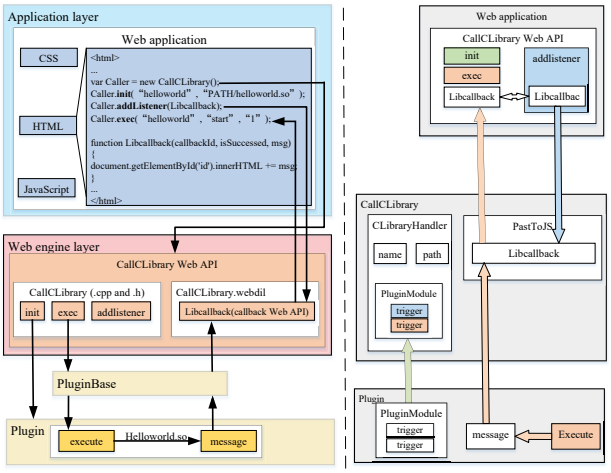

Figure 4: The architecture of CallCLibrary Web API. The left is the overview and the right is the specific implementation with data structure.

identify the corresponding .so library and complete the work for initialization; `Addlistener` adds a callback function to the web engine layer, the message about the .so library would be delivered to the web application through the callback function; `Exec` notifies the .so library to perform the specific actions and trigger the callback function added by `Addlistener` to deliver the message to web application. The parameters of `Exec` include name of the .so library ("`helloworld`"), the specific action about execution ("`start`") and the callbackid ("`1`"), as shown in Fig. 4. The three parameters of the `Libcallback` are callback id ("`callbackid`"), whether the callback function executes successfully ("`isSuccessed`"), and the delivered message ("`msg`"). All the parameters of `Libcallback` are delivered from web engine layer through the callback Web API.

2) CallCLibrary in web engine layer: The specific implementation of `Init`, `Addlistener`, `Exec` is in web engine layer and it is the major part of CallCLibrary. We construct the CallCLibrary Web API through the `CallCLibrary.webidl` and the implementation files, which build the connection between application layer and web engine layer . Then we create a class named `PluginBase` to connect the plugin .so library with web engine layer. In our CallLibrary framework, the plugin .so library only has one class, and the class need inherit the class `PluginBase`, the specific operation about the plugin class need add to the function `execute`. The work about connection with web engine layer is done by `PluginBase`, the only work for the plugin .so library is fill the function `execute`.

Similar with the regular Web API, we create the moz.build file and `CallCLibrary.webidl` as shown in Fig. 5. We define the three functions of the CallCLibrary and declare a callback Web API named `LibCallback`. The callback Web API is used to deliver the callback function in Web-based OS and the callback information would be delivered to the parameters of the function `Libcallback` in web application.

```
1 [Constructor]
2 interface CallCLibrary{
3   boolean init(DOMString lib, DOMString path);
4
5   DOMString exec(DOMString lib, DOMString function,
6           DOMString callbackid);
7
8   void addListener(LibCallback libCallback);
9 };
10
11  callback LibCallback = void (DOMString callbackId,
12          boolean success,DOMString msg);
```

Figure 5: The code snippet of CallCLibrary.webidl.

Next we will present the specific implementation about the CallClibrary. The related data structure in implementation is shown in TABLE. I. We show the implementation in three phases: Initialization of CallCLibrary, adding callback function and function execution in .so library through CallCLibrary.

Initialization of CallCLibrary. In this phase, firstly we get the name and the path of the .so library from web application through webidl. Next we use the function `dlopen` and `dlsym` to get the `PluginModule` in the `PluginBase` and store the `PluginModule` into the `CLibraryhandler`.

Adding callback function. The `Libcallback` from web application will be delivered to web engine layer through the callback Web API, then the trigger function in `PluginModule` adds the `Libcallback` to `PastToJS`.

Execution. The trigger function in `PluginModule` of the ClibraryHandler will notify the .so library to invoke the function `execute`, and execute the specific action in plugin which is implemented before. After the execution, the message to be delivered to web application is generated. The plugin notifies the `PastToJS` to get the `Libcallback` and invokes the `call` API of the `Libcallback` with the message as a parameter in it, then the message about the C/C++ plugin execution would be delivered to the web application through the `LibCallback`.

3) CallCLibrary in plugin (i.e., .so library): All the C/C++ plugins must inherit the class `PluginBase` before being compiled to .so library. The only work for the plugin is completing the function `execute`. The `PluginBase` is the bridge between the plugin and the CallCLibrary. Until now, we describe the specific implementation about CallClibrary Web API, then we will introduce the implementation on our self-test framework.

4) CallCLibrary on our self-test framework: We create a plugin named self-test complying the rule about CallCLibrary and compile it to `self-test.so`. In the self-test plugin, we use the function `fopen` and `fread` to get the message about the device status file. Then we deliver the message to self-test web application by CallCLibrary Web API.

Table I: The Data structure in the implementation about CallLibrary.

Data structures	Description
PluginModule	It exist in PluginBase and CLibraryHandler. The specific implementation is in PluginBase and it would be delivered to CLibraryHandler in the initialization. It contains trigger functions for adding callback function and contains executing function in web engine layer.
CLibraryHandler	The most important data structure in the CallCLibrary. It contains the name and the path of the .so library, and stores a PluginModule which is delivered from PluginBase in the initialization.
PluginBase	The superclass of all the plugins (.so library) in CallCLibrary. It contains a PluginModule and several utility functions, which are responsible for the connection with web engine layer.
PastToJS	In the adding callback function stage, it stores the Libcallback delivered from web application; In the execution stage, it invokes the Libcallback Web API and delivers the callback message to web application.

D. Framework in web application layer.

We develop a web application named self-test which is written in HTML5 and put it in the boot stage of the Web-based OS. The web application receives the hardware status information through the CallCLibrary Web API and displays the information to users.

IV. Evaluations

In this section, we present the experimental study of our proposed self-test framework. Firstly we introduce the setup of the experiments. Next, we conduct two experiments: (i) Verify whether the self-test framework could identify the hardware failure. (ii) Verify the performance of CallCLibrary Web API.

We run the experiments on a smart watch device equipped with Web-based OS (Firefox OS version 44.0) and Android (version 6.0.0) respectively, with its CPU 1.6 GHz, 4 GB RAM. We choose the ADB (Android Debug Bridge) [8] to control the device on a Ubuntu 18.04 desktop platform containing Intel (R) Core (TM) i7 CPU 1.8 GHz with 16 GB RAM.

A. Self-test framework verification.

1) Experiment : The hardware in the smart watch include sensor, Wi-Fi, GPS, Bluetooth, battery and so on. The hardware malfunction is not easy to be created artificially and we conduct the simulation. Take the GPS for example, we change its access permission to deter the web application from accessing it, then the GPS fails to work. We conduct the experiment on the same smart watch device with simulation and no simulation respectively, and observe the self-test framework whether could detect the failure.

2) Results : We press the power-on key, self-test framework shows the GPS works well. Then we conduct the failure simulation and the self-test framework detects the failure as shown in Fig. 6.

B. CallCLibrary performance

We conduct an experiment on the performance of CallCLibrary. The experiment compares CallCLibrary with native Web API and JNI in Android.

Figure 6: The self-test result about GPS.

1) Preliminaries: All three APIs mentioned above could invoke the function in .so library from application level. The CallCLibrary and native Web API are in Web-based OS on which the application is written in HTML5; the JNI is in Android on which the application is written in Java.

Native Web API. The Web-based OS provides Web API which could manipulate the hardware. The web engine layer handles the low-level access to the system using a C++ API that is accessible to the higher levels. For instance, when users need the device vibrate, the web application in application layer submits requests to access to underlying device via vibrate Web API, then the web engine layer submits the request to the infrastructure layer. The single request in web application layer would result in operations in infrastructure layer. In HAL of the infrastructure layer, the .so library implements the connection between the hardware driver and the web engine. The whole process is shown in Fig. 7. Compared with native Web API, the CallCLibrary Web API provides a more flexible interface for higher levels of the Web-based OS, and we could add what we want in the .so library as long as the rule of the CallCLibrary is complied. If we want to add the interface for a new hardware, we could use CallCLibrary instead of adding a new single implementation like the vibrate Web API, in other words, the CallClibrary is more general.

Android JNI. The JNI achieve the inter operation between the Java and native libraries. JNI technology enables Java program to run on JVM (Java virtual machine) [9], pass the parameters to native codes such as C/C++ and get the return values. The invocation in Java attach the .so library in HAL through JNI.

Web-based OS and Android. When developer design

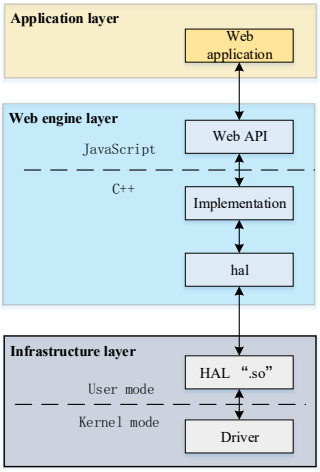

Figure 7: The native Web API in Web-based OS.

the Web-based OS, Android is well-established in the hardware-adaption, so they apply this part to the Web-based OS (i.e., Infrastructure layer) , basing on the AOSP (Android Open Source Project) [10]. Above the infrastructure layer, the Web-based OS transplants browser engine to support the HTML5 application. By contrast, Android use the JVM to support the Java application. In a word, Web-based OS and Android are very similar at the bottom layer such as device driver, which is the basis of experiment B.

2) Experimental design: In this experiment, we test three APIs, which could invoke native function in application: CallCLibrary Web API invokes the function `execute` in `self-test.so`, Vibrate Web API and Vibrate interface in Java (JNI) invoke the function `vibrate_on` in `libhardware_legacy.so`. All the .so library are under the `/system/lib/` directory of the device.

In Web-based OS, we develop a web application to test CallCLibrary and Vibrate Web API. We add a timestamp at the function `exec` of the CallCLibrary which is the entry of the invocation, then we add another timestamp to the function `execute` of the self-test.c, which is to be compiled to `self-test.so`; The Vibrate Web API could make the device equipped with Web-based OS Vibrate. We add `window.navigator.Vibrate()` to the web application, which is the invocation entry and we add a timestamp here, then we add another timestamp to the function `vibrate_on` of the vibrate.c, which is to be compiled to `libhardware_legacy.so`. The `libhardware_legacy.so` belongs to the HAL of the Web-based OS.

In Android, we test the vibrate interface in Java. We develop a vibrate android application with Android Studio (version 4.2.2) In Java application, we import the package `android.os.Vibrator` and get the object vibrator with the function `getSystemService (Service.VIBRATOR_SERVICE)`. Then we invoke the function `vibrator.vibrate(1000)` which means vibrate 1000ms on the device and add a timestamp here. Next, similar with the Vibrate Web API, we add another timestamp to the function `vibrate_on` in vibrate.c.

We conduct the invocation 50 times on each APIs (CallCLibrary Web API, native Web API, JNI). The invocation time $\triangle T = T2 - T1$, T2 is the time when the function in .so library being invoked and T1 is the time when the API in application(Web or Java) being invoked. We utilize the ADB tools to observe the experiment result and input the "adb logcat" in the terminal of PC and get the according time.

3) Results and discussions: As shown in Fig. 8, the CallCLibrary outperforms native Web API and JNI. The invocation time about CallCLibrary is smoother than others. The average time in 50 invocations for CallCLibrary is 2.7ms, 3.88ms for native Web API and 3.38ms for JNI.

The CallCLibrary directly invokes the .so library while the native Web API has to complete the work in HAL before invokes the .so library, so is the JNI, which consume extra time. We propose the CallCLibrary to provide a more flexible method for application to invoke the function in .so library, and the performance is acceptable.

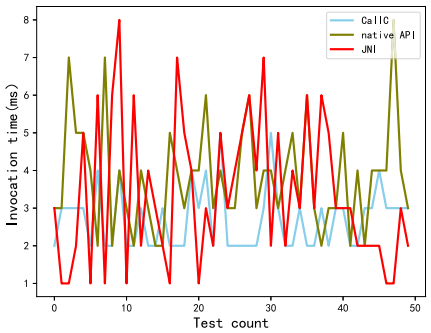

Figure 8: The invocation time about CallCLibrary, native Web API and JNI.

V. RELATED WORKS

A. Fault detection on mobile OS

We mainly focus on Android operating system, because the bottom of Web-based OS is based on AOSP. The relationship between Android and Web-based OS is introduced in Experiment B. Android fault detection consists of hardware and software detection. The software detection consists of two approaches, namely, static and dynamic analysis [11]. As for the hardware detection, the study is mainly focus on the driver analysis [12]–[14], the power-on hardware test using the driver registration is not involved.

B. C/C++ codes on Web

The CallCLibrary could invoke native code in web application which is developed in a Web environment. At present, there are two methods for Web to invoke native code such as C/C++.

Compile the C/C++ code to the language the browser could identify. The asm.js [15] is a subset of JavaScript

and is mainly used as compilation target. Developers start with an existing C/C++ application which they can then efficiently port to the web by compiling it to asm.js using Emscripten [16]. WebAssembly [17] is a new byte code designed for the web. It also provides a compilation target for languages such as C/C++. The Web Assembly cannot directly access Web APIs. The two methods mentioned above is used for run the native application which is compiled before on the web, as a comparison, we complete the invocation through CallClibrary Web API on web which is more flexible, we could invoke the native code whenever we want.

Invoke the C/C++ dynamic linked library in the browser. The NPAPI (Netscape Plugin Application Programming Interface) could make the native code run as part of the web application through invoking the C/C++ `.so` library. While the plugin developed by NPAPI runs at the same level as the browser, which is a huge security risk to the system, the NPAPI is blocked by mainstream browser since 2014. Node.js is a server-side JavaScript environment built on Google Chrome's v8 engine. There are two main ways for Node.js to use the function of the C/C++layer, the first is to call the global variables process, Buffer, etc., and the second is the function process.binding. Node.js does not apply to the scenario in this paper.

VI. CONCLUSIONS

The hardware status is a major security part of an mobile OS. Web-based OS is a type of mobile OS and there is not a mechanism for the hardware status test in the OS. So we proposed a self-test framework in the boot stage of the device equipped with Web-based OS. In the implementation of the self-test framework, we propose a Web API named CallCLibrary to facilitate the upper application invoke the native code. As a result, the hardware status information is delivered from the file in bottom of the OS to upper application through the CallCLibrary Web API. In experiments, we verify the self-test framework and the performance of CallCLibrary. The experiments show that the framework could identify the hardware failure accurately and CallCLibrary outperform other similar technologies such as native Web API and JNI.

REFERENCES

[1] M. Jadhav and K. K. Joshi, "Forensic investigation procedure for data acquisition and analysis of Firefox OS based mobile devices," 2016 International Conference on Computing, Analytics and Security Trends (CAST), 2016, pp. 456-461

[2] M. N. Yusoff, R. Mahmod, M. T. Abdullah and A. Dehghantanha,"Mobile forensic data acquisition in Firefox OS," 2014 Third International Conference on Cyber Security, Cyber Warfare and Digital Forensic (CyberSec), 2014, pp. 27-31

[3] I. Bente, B. Hellmann, T. Rossow, J. Vieweg and J. von Helden, "On Remote Attestation for Google Chrome OS,"2012 15th International Conference on Network-Based Information Systems, 2012, pp. 376-383

[4] B2G architecture, 02/06/2021. [Online]. Available: https://developer.mozilla.org/enUS/docs/Archive/B2G_OS/Architecture

[5] Web API, 03/06/2021. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API

[6] C. Qian, X. Luo, Y. Shao and A. T. S. Chan, "On Tracking Information Flows through JNI in Android Applications," 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, 2014, pp. 180-191
2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), 2021, pp. 1708-1718

[7] I. Pustogarov, Q. Wu and D. Lie, "Ex-vivo dynamic analysis framework for Android device drivers," 2020 IEEE Symposium on Security and Privacy (SP), 2020, pp. 1088-1105

[8] ADB, [online] Available: https://developer.android.com/studio/command-line/adb

[9] Chien-Wei Chang, Chun-Yu Lin, Chung-Ta King, Yi-Fan Chung and Shau-Yin Tseng, "Implementation of JVM tool interface on Dalvik virtual machine," Proceedings of 2010 International Symposium on VLSI Design, Automation and Test, 2010, pp. 143-146

[10] X. Song and C. Yang, "Mobile Device Management System Based on AOSP and SELinux," 2017 IEEE Second International Conference on Data Science in Cyberspace (DSC), 2017, pp. 417-420

[11] L. Taheri, A. F. A. Kadir and A. H. Lashkari, "cExtensible Android Malware Detection and Family Classification Using Network-Flows and API-Calls," 2019 International Carnahan Conference on Security Technology (ICCST), 2019, pp. 1-8

[12] S. M. S. Talebi, H. Tavakoli, H. Zhang, Z. Zhang, A. A. Sani, and Z. Qian, "Charm: Facilitating dynamic analysis of device drivers of mobile systems," in 27th USENIX Security Symposium (USENIX Security 18). Baltimore, MD:USENIX Association, 2018, pp. 291–307

[13] D. Davidson, B. Moench, T. Ristenpart, and S. Jha, "FIE on firmware: Finding vulnerabilities in embedded systems using symbolic execution," in Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13). Washington, D.C.: USENIX, 2013, pp. 463–478.

[14] D. Song, F. Hetzelt, D. Das, C. Spensky, Y. Na, S. Volckaert, G. Vigna, C. Kruegel, J.-P. Seifert, and M. Franz, "Periscope: An effectiveness probing and fuzzing framework for the hardware-OS boundary," in 2019 Network and Distributed Systems Security Symposium (NDSS). Internet Society, 2019, pp. 1–15.

[15] Noah Van Es, Jens Nicolay, Quentin Stievenart, Theo D'Hondt, and Coen De Roover. "A performant scheme interpreter in asm.js," in 31st Annual ACM Symposium on Applied Computing (SAC 16). New York, Association for Computing Machinery, 2016, pp.1944–1951.

[16] A. Zakai, "Fast Physics on the Web Using C++, JavaScript, and Emscripten," in Computing in Science & Engineering, vol. 20, no. 1, pp. 11-19, January/February 2018.

[17] A. Romano and W. Wang, "cWASim: Understanding WebAssembly Applications through Classification," 2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2020, pp. 1321-1325.