Introduce LLVM from a hacker's view.

Loda chou.
hlchou@mail2000.com.tw
2012/07/02

Who am I?

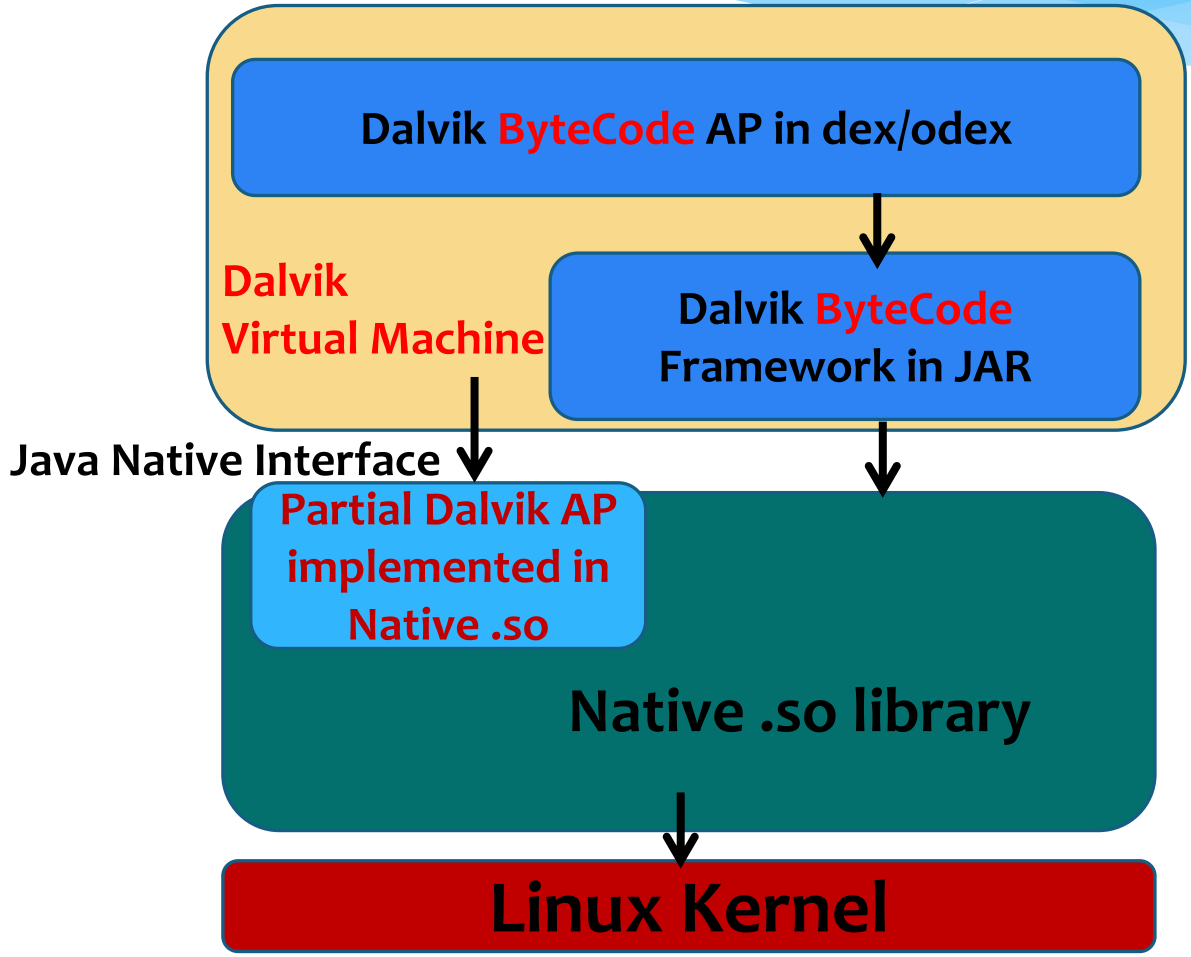
- I am Loda.
- Work for 豬屎屋 (DeSign House).
- Be familiar for MS-Windows System and Android/Linux Kernel.
  - Sometimes…also do some software crack job.
- Like to dig-in new technology and share technical articles to promote to the public.
- Motto
  - The way of a fool seems right to him ,but a wise man listens to advice. **(Proverbs 12:15)**

What is LLVM?

- Created by Vikram Adve and Chris Lattne on 2000
- Support different front-end compilers (gcc/clang/....) and different languages (C/C++,Object-C,Fortran,Java ByteCode,Python,ActionScript) to generate BitCode.
- The core of LLVM is the intermediate representation (IR). Different front-ends would compile source code to SSA-based IR, and traslate the IR into different native code on different platform.
- Provide RISC-like instructions (load/store…etc), unlimited registers, exception (setjmp/longjmp)..etc
- Provide LLVM Interpreter and LLVM Compiler to run LLVM application.

Let's enjoy it.

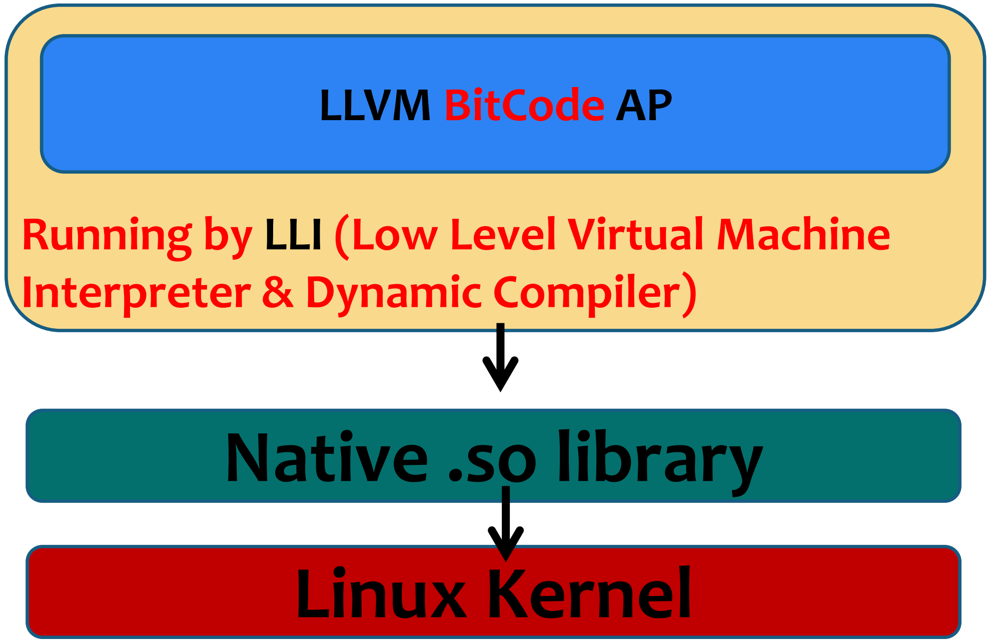
Android Dalvik RunTime


Dalvik ByteCode AP in dex/odex
Dalvik Virtual Machine
Dalvik ByteCode Framework in JAR
Java Native Interface
Partial Dalvik AP implemented in Native .so
Native .so library
Linux Kernel

The features of Dalvik VM

- Per-Process per-VM
- JDK will compile Java to Sun's bytecode, Android would use dx to convert Java bytecode to Dalvik bytecode.
- Support Portable Interpreter (in C), Fast Interpreter (in Assembly) and Just-In Time Compiler
- Just-In-Time Compiler is Trace-Run based.
  - By Counter to find the hot-zone
  - Would translate Dalvik bytecode to ARMv32/NEON/Thumb/Thumb2/..etc CPU instructions.

LLVM Interpreter RunTime

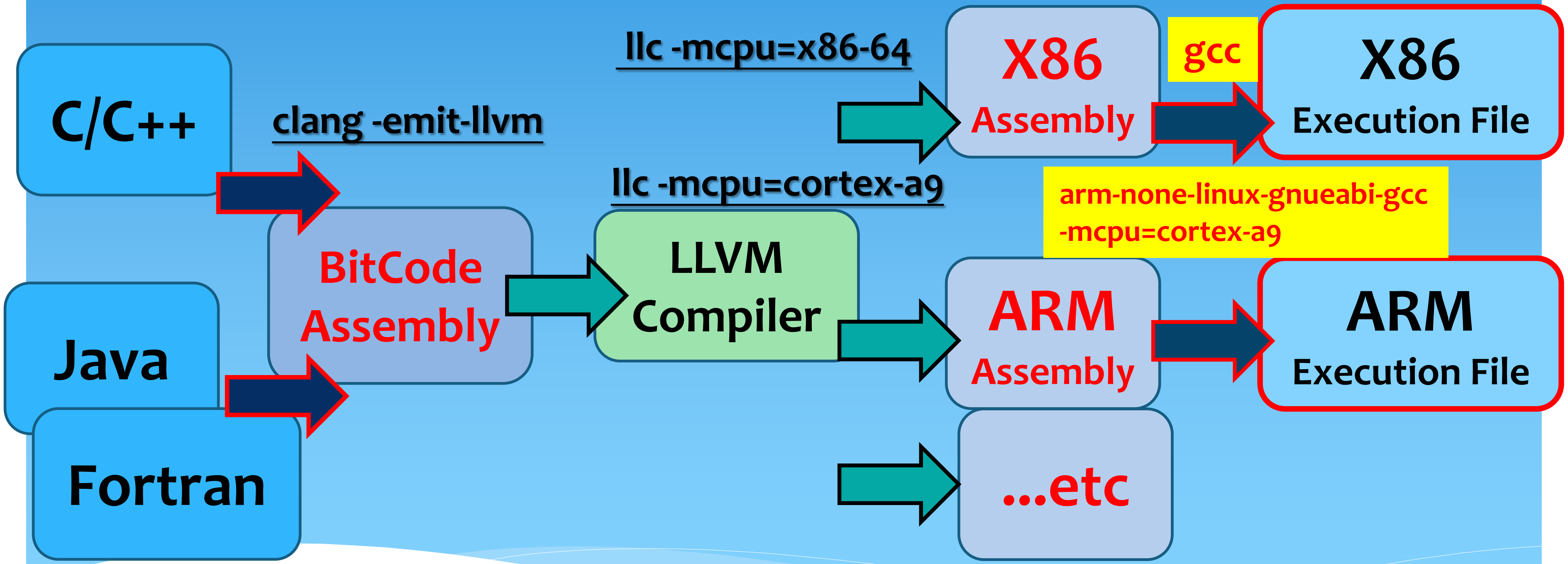


LLVM BitCode AP
Running by LLI (Low Level Virtual Machine Interpreter & Dynamic Compiler)
Native .so library
Linux Kernel

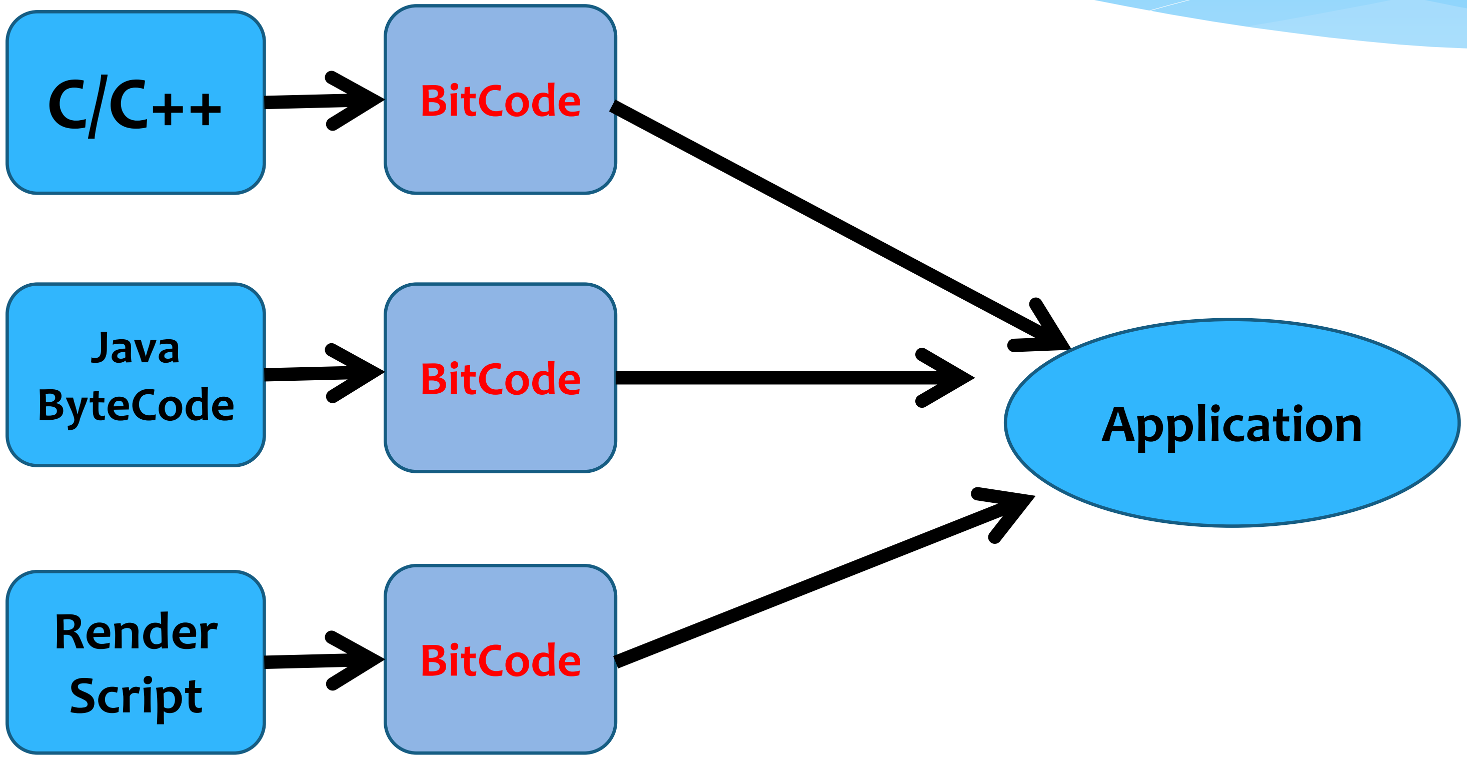
Why LLVM?

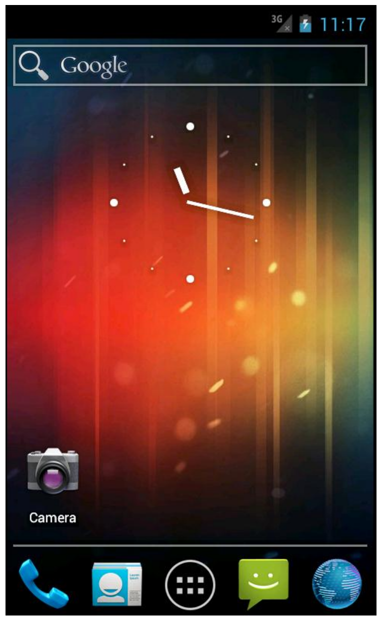
- Could run llvm-application as the performance of native application
- Could generate small size BitCode, translate to target platform assembly code then compiled into native execution file (final size would be almost the same as you compile it directly from source by GCC or other compiler.)
- Support C/C++/… program to seamlessly execute on variable hardware platform.
  - x86, ARM, MIPS,PowerPC,Sparc,XCore,Alpha… etc
- **Google** would apply it into Android and Browser (Native Client)

The LLVM Compiler Work-Flows.

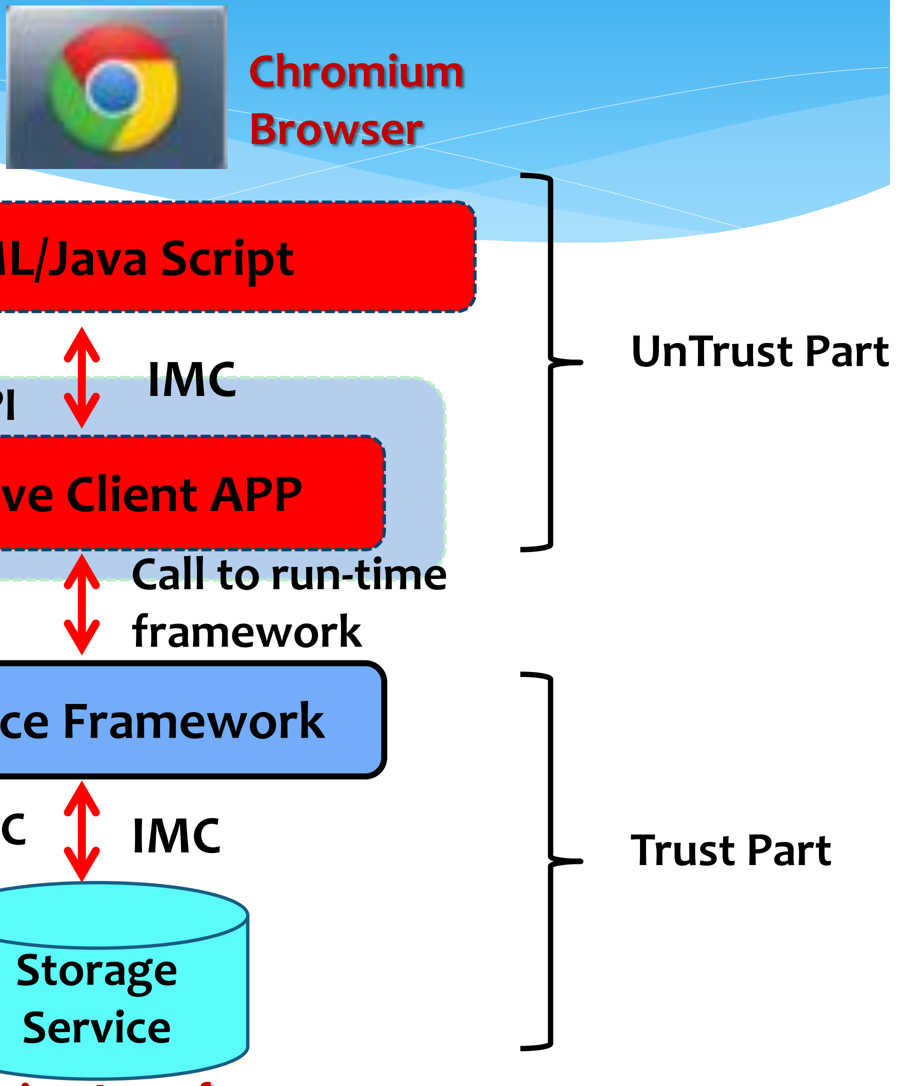


LLVM in Mobile Device





LLVM in Browser



Would passed the security checking before execution.

IMC : Inter-Module Communications
SRPC : Simple RPC
NPAPI : Netscape Plugin Application Programming Interface

LLVM Compiler Demo.

➔Use clang to compile BitCode File.

```
[root@localhost reference_code]# clang -O2 -emit-llvm sample.c -c -o sample.bc
[root@localhost reference_code]# ls -l sample.bc
-rw-r--r--. 1 root root 1956 May 12 10:28 sample.bc
```

➔Convert BitCode File to x86-64 platform assembly code.

```
[root@localhost reference_code]# llc -O2 -mcpu=x86-64 sample.bc -o sample.s
```

➔Compiler the assembly code to x86-64 native execution file.

```
[root@localhost reference_code]# gcc sample.s -o sample -ldl
[root@localhost reference_code]# ls -l sample
-rwxr-xr-x. 1 root root 8247 May 12 10:36 sample
```

➔Convert BitCode File to ARM Cortext-A9 platorm assembly code.

```
[root@localhost reference_code]# llc -O2 -march=arm -mcpu=cortex-a9 sample.bc -o
sample.s
```

➔Compiler the assembly code to ARM Cortext-A9 native execution file.

```
[root@localhost reference_code]# arm-none-linux-gnueabi-gcc -mcpu=cortex-a9 sample.s -ldl -o
sample
[root@localhost reference_code]# ls -l sample
-rwxr-xr-x. 1 root root 6877 May 12 10:54 sample
```

What is the problems for LLVM?

Let' s see a simple sample code.

LLVM dlopen/dlsymc Sample.

- [root@www LLVM]# clang -O2 -emit-llvm dlopen.c -c -o dlopen.bc
- [root@www LLVM]# lli dlopen.bc
- libraryHandle:86f5e4c8h
- puts function pointer:85e81330h
- loda

```
int (*puts_fp)(const char *);

int main()
{
        void * libraryHandle;
        libraryHandle = dlopen("libc.so.6", RTLD_NOW);
        printf("libraryHandle:%xh\n",(unsigned int)libraryHandle);
        puts_fp = dlsym(libraryHandle, "puts");
        printf("puts function pointer:%xh\n",(unsigned int)puts_fp);
        puts_fp("loda");
        return 0;
}
```

Make execution code as data buffer

- Would place the piece of machine code as a data buffer to verify the native/LLVM run-time behaviors.

```
0000000000000000 <AsmFunc>:
  0:   55                      push   %rbp
  1:   48 89 e5                mov    %rsp,%rbp
  4:   b8 04 00 00 00          mov    $0x4,%eax
  9:   bb 01 00 00 00          mov    $0x1,%ebx
  e:   b9 00 00 00 00          mov    $0x0,%ecx
              f: R_X86_64_32  gpHello
 13:   ba 10 00 00 00          mov    $0x10,%edx
 18:   cd 80                   int    $0x80
 1a:   b8 11 00 00 00          mov    $0x11,%eax
 1f:   c9                      leaveq
 20:   c3                      retq
```

Native Program Run Code in Data Segment

- **[root@www LLVM]# gcc self-modify.c -o self-modify**
- **[root@www LLVM]# ./self-modify**
- **Segmentation fault**

```
int (*f2)();

char
TmpAsmCode[]={0x90,0x55,0x48,0x89,0xe5,0xb8,0x04,0x00,0x00,0x00,0xbb,0x01,0x00,0x00,0x00,0xb9,0x4
0,0x0c,0x60,0x00,0xba,0x10,0x00,0x00,0x00,0xcd,0x80,0xb8,0x11,0x00,0x00,0x00,0xc9,0xc3};
char gpHello[]="Hello Loda!ok!\n";
int main()
{
        int vRet;
        unsigned long vpHello=(unsigned long)gpHello;
        TmpAsmCode[19]=vpHello>>24 & 0xff;
        TmpAsmCode[18]=vpHello>>16 & 0xff;
        TmpAsmCode[17]=vpHello>>8 & 0xff;
        TmpAsmCode[16]=vpHello & 0xff;
        f2=(int (*)())TmpAsmCode;
        vRet=f2();
        printf("vRet=:%d\n",vRet);
        return 0;
}
```

Native Program Run Code in Data Segment with Page EXEC-settings

```
[root@www LLVM]# gcc self-modify.c -o self-modify
[root@www LLVM]# ./self-modify
Hello Loda!ok!
vRet=:17
```

```
int (*f2)();
char
TmpAsmCode[]={0x90,0x55,0x48,0x89,0xe5,0xb8,0x04,0x00,0x00,0x00,0xbb,0x01,0x00,0x00,0x00,0xb9,0x4
0,0x0c,0x60,0x00,0xba,0x10,0x00,0x00,0x00,0xcd,0x80,0xb8,0x11,0x00,0x00,0x00,0xc9,0xc3};
char gpHello[]="Hello Loda!ok!\n";
int main()
{
        int vRet;
        unsigned long vpHello=(unsigned long)gpHello;
        unsigned long page = (unsigned long) TmpAsmCode & ~( 4096 - 1 );
        if(mprotect((char*) page,4096,PROT_READ | PROT_WRITE | PROT_EXEC  ))
                perror( "mprotect failed" );
        TmpAsmCode[19]=vpHello>>24 & 0xff;
        TmpAsmCode[18]=vpHello>>16 & 0xff;
        TmpAsmCode[17]=vpHello>>8 & 0xff;
        TmpAsmCode[16]=vpHello & 0xff;
        f2=(int (*)())TmpAsmCode;
        vRet=f2();
        printf("vRet=:%d\n",vRet);
        return 0;
}
```

LLVM AP Run Code in Data Segment with EXEC-settings

- **[root@www LLVM]# clang -O2 -emit-llvm llvm-self-modify.c -c -o llvm-self-modify.bc**
- **[root@www LLVM]# lli llvm-self-modify.bc**
- **Hello Loda!ok!**
- **vRet=:17**

```
int (*f2)();

char
TmpAsmCode[]={0x90,0x55,0x48,0x89,0xe5,0xb8,0x04,0x00,0x00,0x00,0xbb,0x01,0x00,0x00,0x00,0xb9,0x40,0x0c,0x60,0x00,0xb
a,0x10,0x00,0x00,0x00,0xcd,0x80,0xb8,0x11,0x00,0x00,0x00,0xc9,0xc3};
char gpHello[]="Hello Loda!ok!\n";

int main()
{
      int vRet;
      unsigned long vpHello=(unsigned long)gpHello;

      unsigned long page = (unsigned long) TmpAsmCode & ~( 4096 - 1 );
      if(mprotect((char*) page,4096,PROT_READ | PROT_WRITE | PROT_EXEC  ))
             perror( "mprotect failed" );
      char *base_string=malloc(256);
      strcpy(base_string,gpHello);
      vpHello=(unsigned long)base_string;
      TmpAsmCode[19]=vpHello>>24 & 0xff;
      TmpAsmCode[18]=vpHello>>16 & 0xff;
      TmpAsmCode[17]=vpHello>>8 & 0xff;
      TmpAsmCode[16]=vpHello & 0xff;
      f2=(int (*)())TmpAsmCode;
      vRet=f2();
      printf("vRet=:%d\n",vRet);
      return 0;
}
```

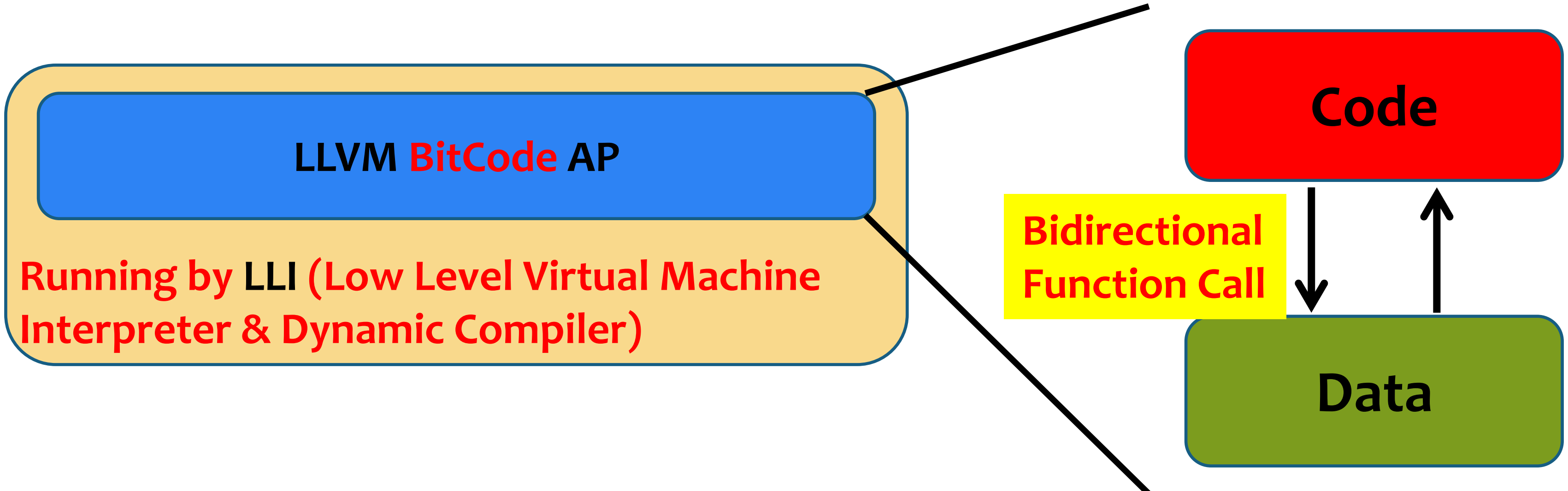
LLVM AP Run Code in Data Segment without EXEC-settings?

- [root@www LLVM]# clang -O2 -emit-llvm llvm-self-modify.c -c -o llvm-self-modify.bc
- [root@www LLVM]# lli llvm-self-modify.bc
- Hello Loda!ok! ➔ It still works!
- vRet=:17

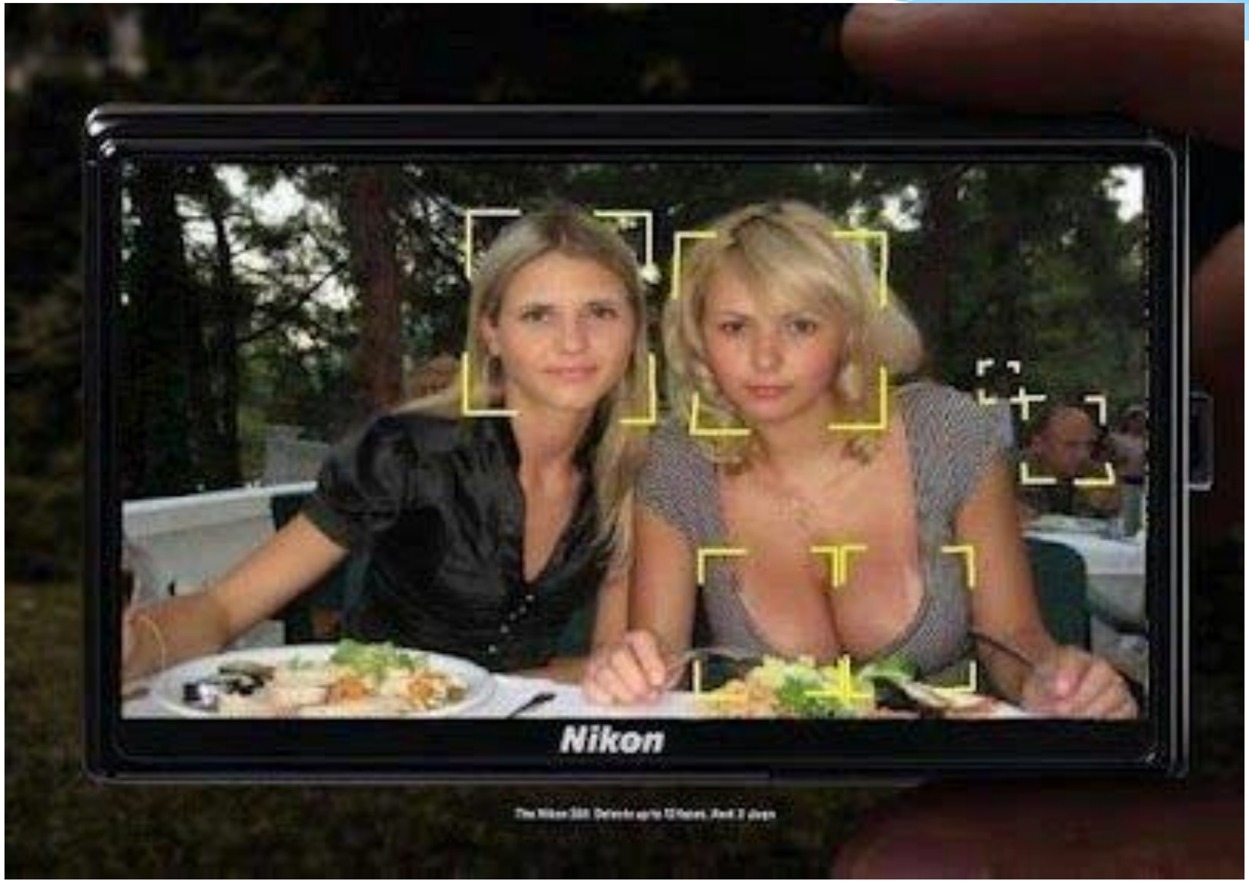
```
int (*f2)();

char
TmpAsmCode[]={0x90,0x55,0x48,0x89,0xe5,0xb8,0x04,0x00,0x00,0x00,0xbb,0x01,0x00,0x00,0x00,0xb9,0x40,0x0c,0x60,0x00,0xb
a,0x10,0x00,0x00,0x00,0xcd,0x80,0xb8,0x11,0x00,0x00,0x00,0xc9,0xc3};
char gpHello[]="Hello Loda!ok!\n";

int main()
{
        int vRet;
        unsigned long vpHello=(unsigned long)gpHello;

        char *base_string=malloc(256);
        strcpy(base_string,gpHello);
        vpHello=(unsigned long)base_string;
        TmpAsmCode[19]=vpHello>>24 & 0xff;
        TmpAsmCode[18]=vpHello>>16 & 0xff;
        TmpAsmCode[17]=vpHello>>8 & 0xff;
        TmpAsmCode[16]=vpHello & 0xff;
        f2=(int (*)())TmpAsmCode;
        vRet=f2();
        printf("vRet=:%d\n",vRet);
        return 0;
}
```

So……What we got?

* LLVM could run data-segment as execution code.
* LLVM doesn't provide a strict sandbox to prevent the unexpected program flows.
* For installed-application, maybe it is ok. (could protect by Android Kernel-Level Application Sandbox)
* How about LLVM running in Web Browser?


LLVM BitCode AP
Running by LLI (Low Level Virtual Machine Interpreter & Dynamic Compiler)
Code
Bidirectional Function Call
Data

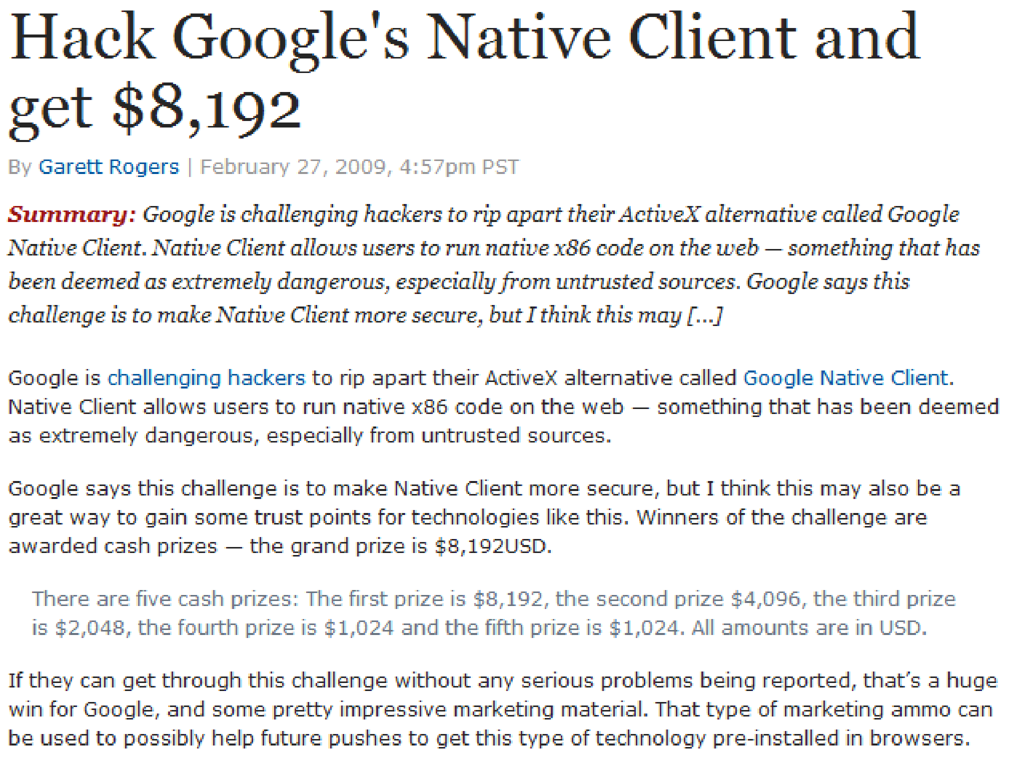
Technology always come from humanity!!!



Native Client(Nacl) - a vision of the future

- Provide the browser to run web application in native code.
- Based on Google's sandbox, it would just drop 5% performance compared to original native application.
- Could be available in Chrome Browser already.
- The Native Client SDK only support the C/C++ on x86 32/64 bits platform.
- Provide Pepper APIs (derived from Mozilla NPAPI). Pepper v2 added more APIs.

Hack Google's Native Client and get $8,192



Hack Google's Native Client and get $8,192

By Garett Rogers | February 27, 2009, 4:57pm PST

Summary: *Google is challenging hackers to rip apart their ActiveX alternative called Google Native Client. Native Client allows users to run native x86 code on the web — something that has been deemed as extremely dangerous, especially from untrusted sources. Google says this challenge is to make Native Client more secure, but I think this may [...]*

Google is challenging hackers to rip apart their ActiveX alternative called Google Native Client. Native Client allows users to run native x86 code on the web — something that has been deemed as extremely dangerous, especially from untrusted sources.

Google says this challenge is to make Native Client more secure, but I think this may also be a great way to gain some trust points for technologies like this. Winners of the challenge are awarded cash prizes — the grand prize is $8,192USD.

> There are five cash prizes: The first prize is $8,192, the second prize $4,096, the third prize is $2,048, the fourth prize is $1,024 and the fifth prize is $1,024. All amounts are in USD.

If they can get through this challenge without any serious problems being reported, that's a huge win for Google, and some pretty impressive marketing material. That type of marketing ammo can be used to possibly help future pushes to get this type of technology pre-installed in browsers.

http://www.zdnet.com/blog/google/hack-googles-native-client-and-get-8192/1295

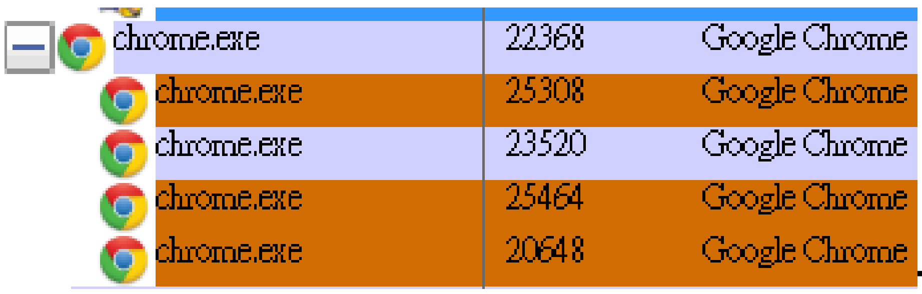
Security of Native Client

- Data integrity
  - Native Client's sandbox works by validating the untrusted code (the compiled Native Client module) before running it
- No support for process creation / subprocesses
  - You can call pthread
- No support for raw TCP/UDP sockets (websockets for TCP and peer connect for UDP)
- No unsafe instructions
  - inline assembly must be compatible with the Native Client validator (could use ncval utility to check)

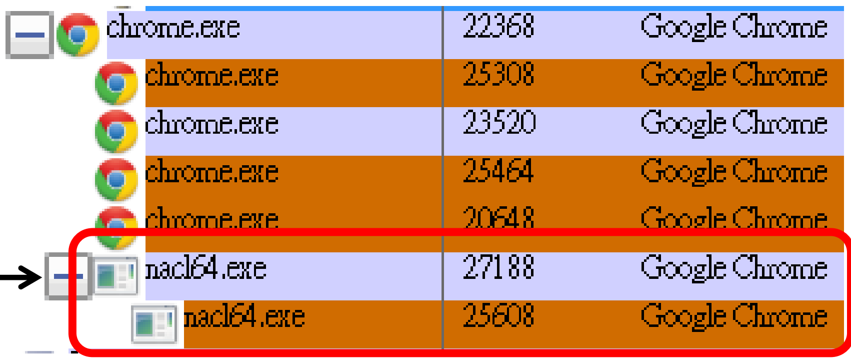
http://code.google.com/p/nativeclient/issues/list

How Native Client Work?

Chromium Browser → Browsing WebPage with Native Client. → **Launch nacl64.exe to Execute the NaCl Executable (*.NEXE) file.**



chrome.exe	22368	Google Chrome
chrome.exe	25308	Google Chrome
chrome.exe	23520	Google Chrome
chrome.exe	25464	Google Chrome
chrome.exe	20648	Google Chrome



chrome.exe	22368	Google Chrome
chrome.exe	25308	Google Chrome
chrome.exe	23520	Google Chrome
chrome.exe	25464	Google Chrome
chrome.exe	20648	Google Chrome
nacl64.exe	27188	Google Chrome
nacl64.exe	25608	Google Chrome

Main Process and Dynamic Library

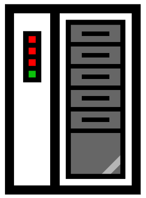
C:\Users\loda\AppData\Local\Temp

6934.Tmp (=**libc.so.3c8d1f2e**)
6922.Tmp (=**libdl.so.3c8d1f2e**)
6933.tmp (=**libgcc_s.so.1**)
6912.tmp (=**libpthread.so.3c8d1f2e**)
67D8.tmp (=**runnable-ld.so**)
66AE.tmp (=**hello_loda.nmf**)
6901.Tmp (= **hello_loda_x86_64.nexe**)

lib64/libc.so.3c8d1f2e
lib64/libdl.so.3c8d1f2e
lib64/libgcc_s.so.1
lib64/libpthread.so.3c8d1f2e
lib64/runnable-ld.so
hello_loda.html
hello_loda.nmf
hello_loda_x86_32.nexe
hello_loda_x86_64.nexe

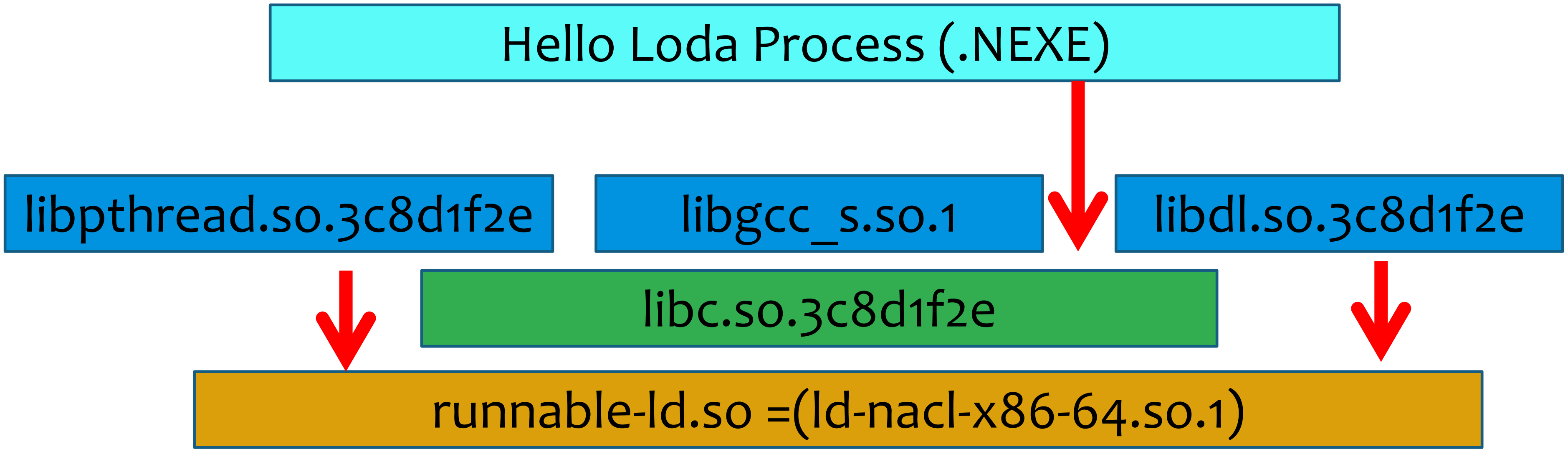
Chromium Browser

Download the main process and dynamic run-time libraries.



Server provided Native Client Page

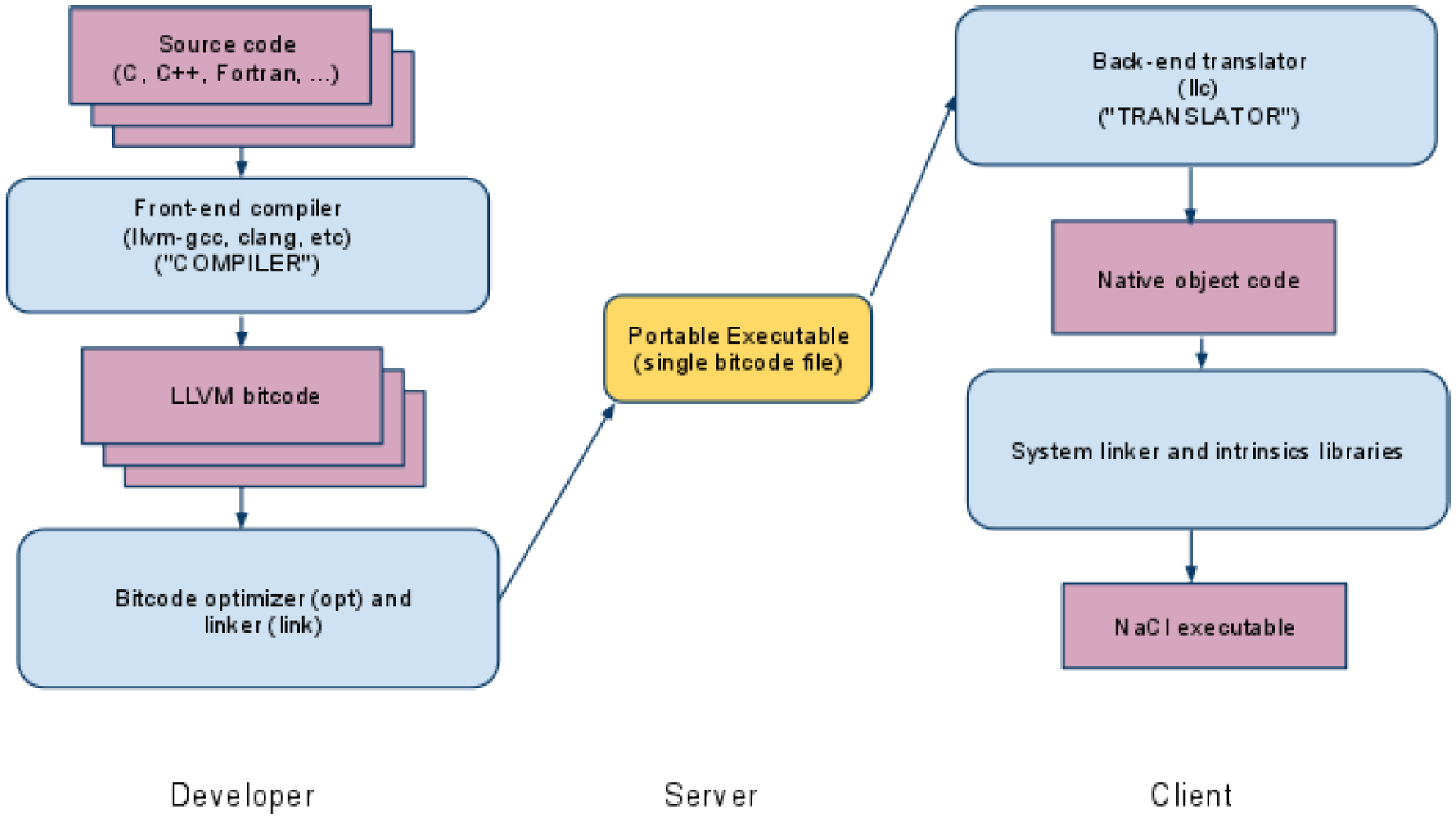
Dynamic libraries Inheritance relationship



Portable Native Client (PNaCl)

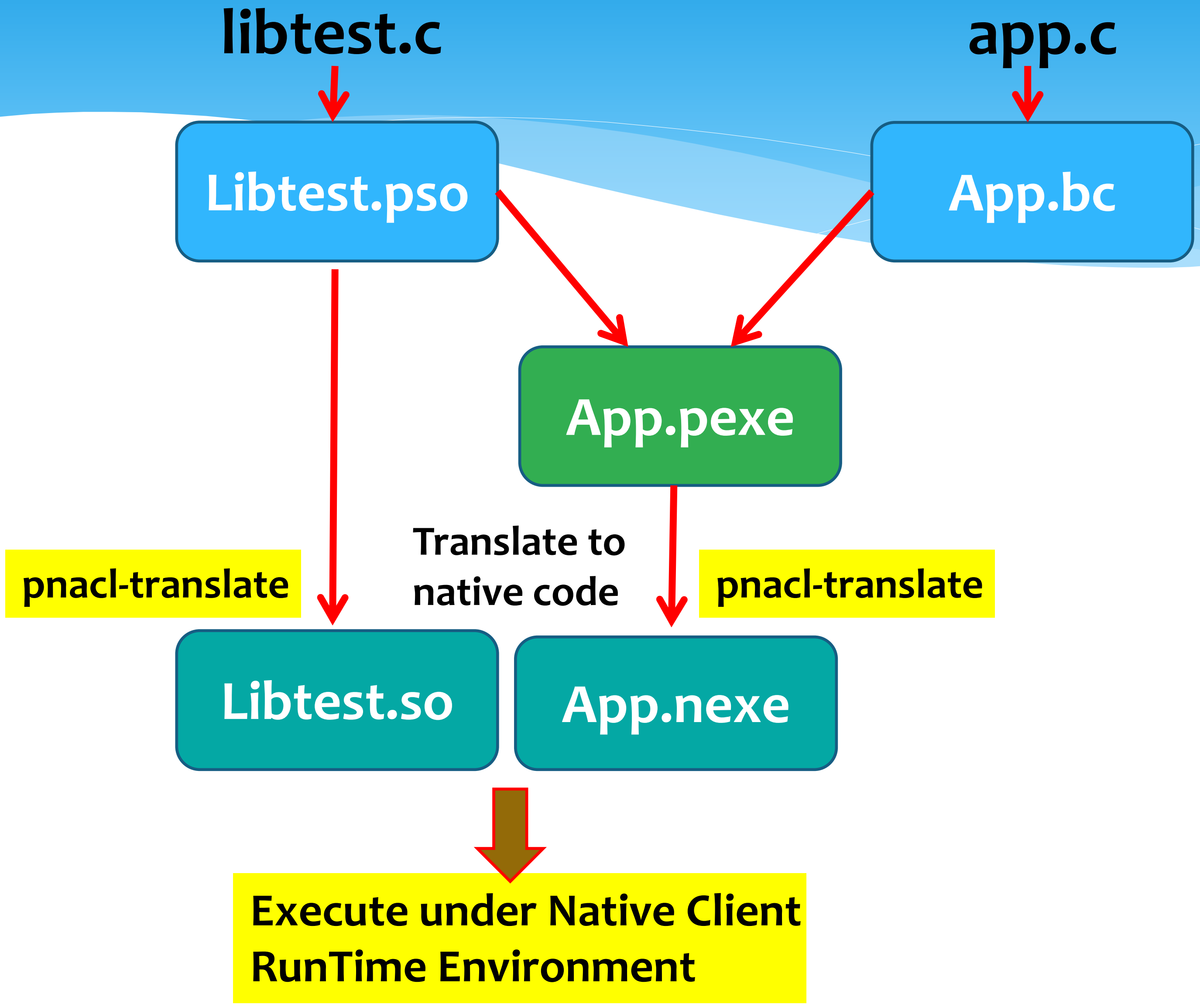
- **PNaCl** (pronounced "pinnacle")
- Based on **LLVM** to provided an ISA-neutral format for compiled NaCl modules supporting a wide variety of target platforms without recompilation from source.
- Support the x86-32, x86-64 and ARM instruction sets now.
- Still under the **security** and **performance** properties of Native Client.

LLVM and PNaCl



Refer from Google's 'PNaCl Portable Native Client Executables ' document.

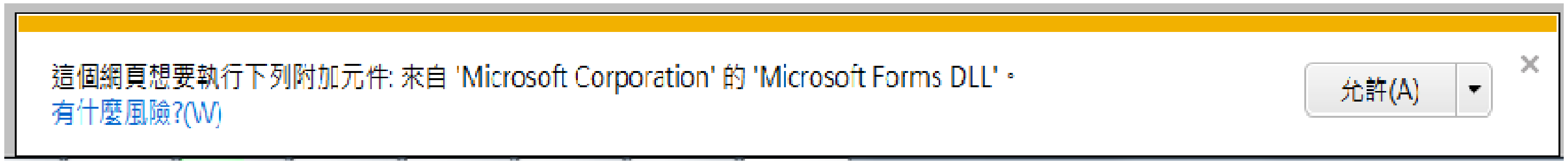
PNaCl Shared Libraries



http://www.chromium.org/nativeclient/pnacl/pnacl-shared-libraries-final-picture

Before SFI

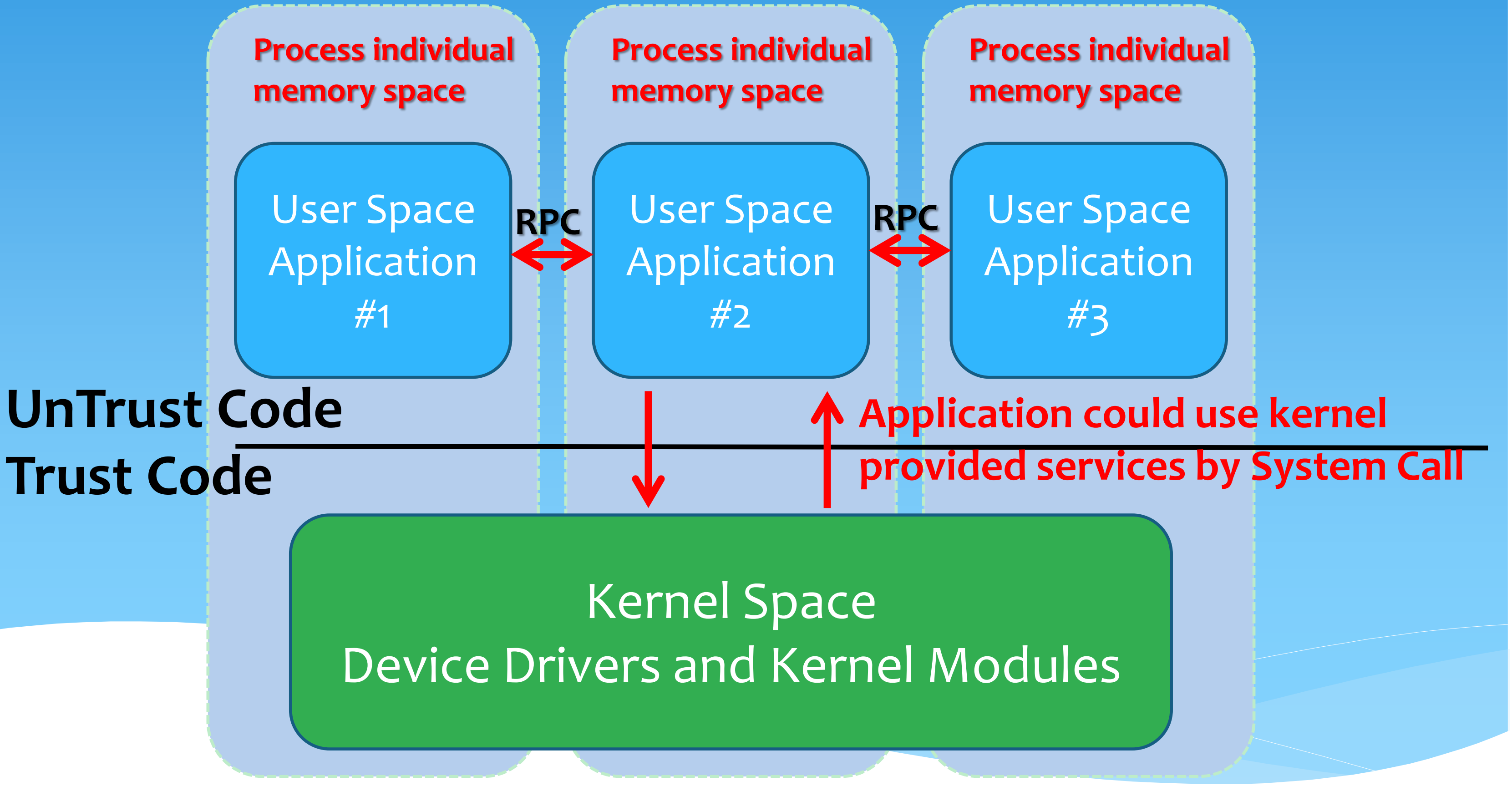
* **Trust with Authentication**
  * Such as the ActiveX technology in Microsoft Windows, it would download the native web application plug-in the browser (MS Internet Explorer). User must authorize the application to run in browser.



這個網頁想要執行下列附加元件: 來自 'Microsoft Corporation' 的 'Microsoft Forms DLL'。
有什麼風險?(W)

* **User-ID based Access Control**
  * Android Application Sandbox use Linux user-based protection to identify and isolate application resources. Each Android application runs as that user in a separate process, and cannot interact with each other under the limited access to the operating system..

General User/Kernel Space Protection


Process individual memory space
Process individual memory space
Process individual memory space
User Space Application #1
RPC
User Space Application #2
RPC
User Space Application #3
UnTrust Code
Trust Code
Application could use kernel provided services by System Call
Kernel Space
Device Drivers and Kernel Modules

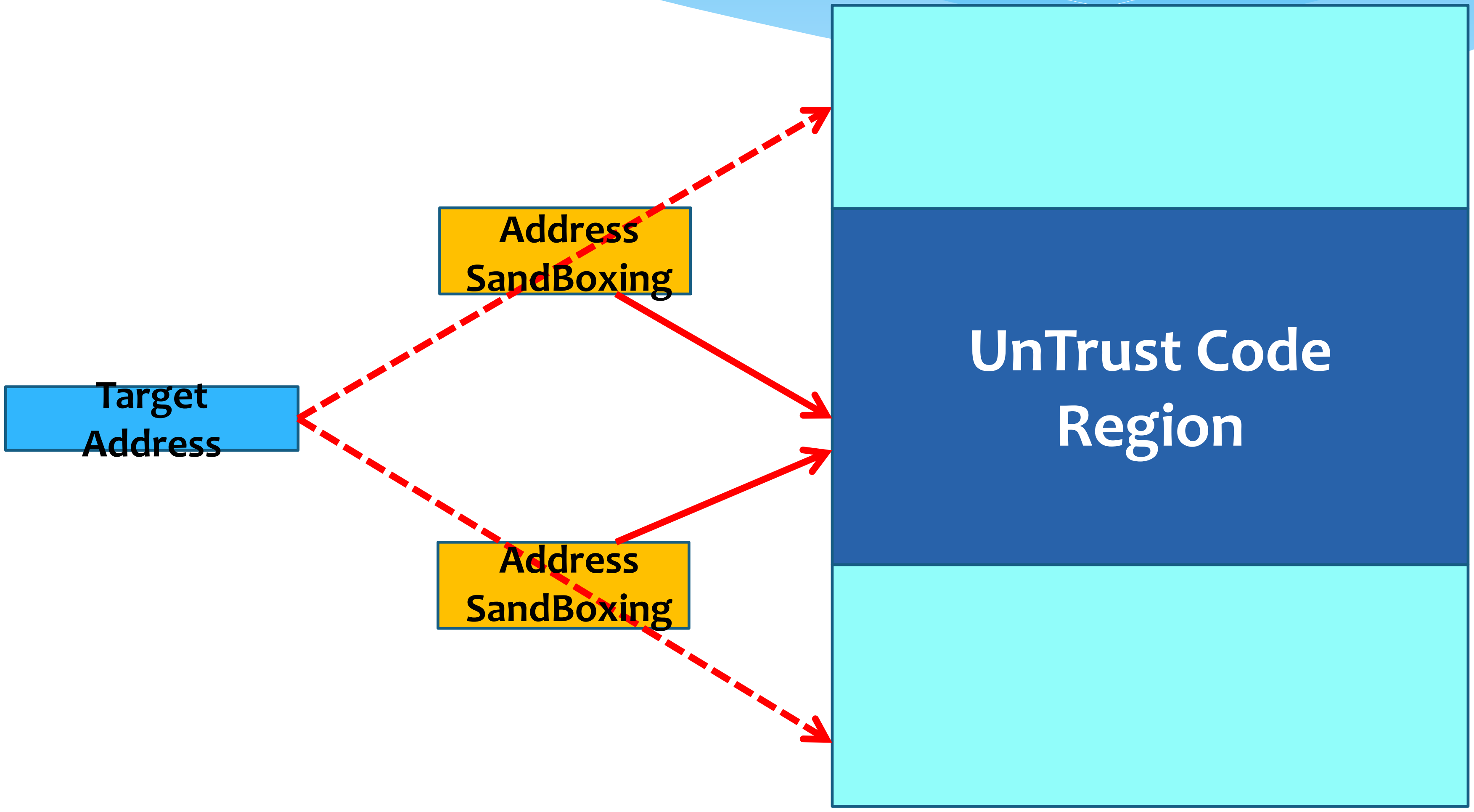
Fault Isolation

- CFI (CISC Fault Isolation)
  - Based on x86 Code/Data Segment Register to reduce the overhead, NaCl CFI would increase around **2%** overhead.
- SFI
  - NaCl SFI would increase **5%** overhead in Cortex A9 out-of-order ARM Processor, and **7%** overhead in x86_64 Processor.

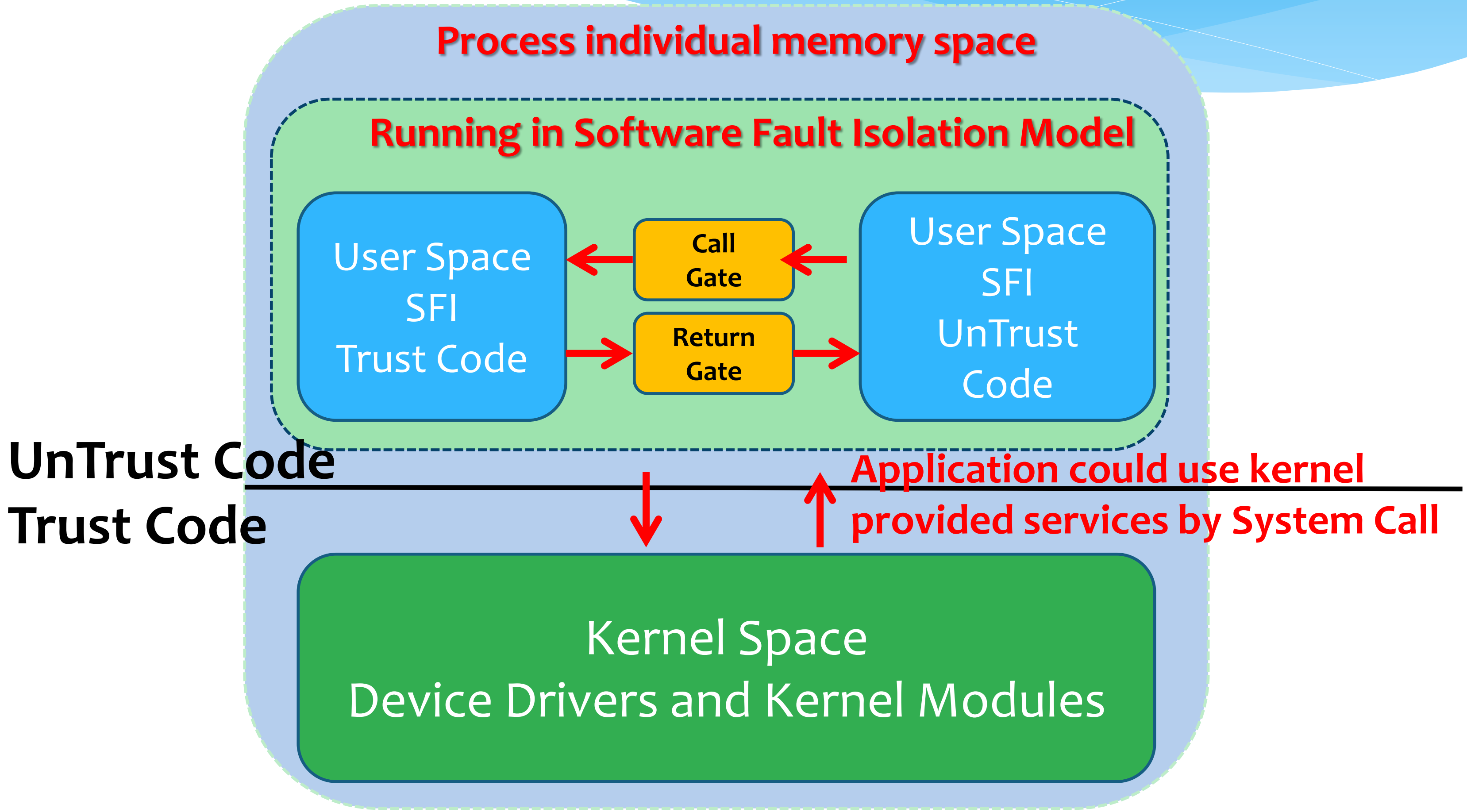
1,ARM instruction length is fixed to **32-bits** or **16bits**
(depend on ARMv32,Thumb or Thumb2 ISA)
2,X86 instruction length is variable from 1 to 1x bytes.

CISC Fault Isolation

Data/Code Dedicated Register=
(Target Address & And-Mask Register) | Segment Identifier Dedicated Register



Software Fault Isolation


Process individual memory space
Running in Software Fault Isolation Model
User Space SFI Trust Code
Call Gate
Return Gate
User Space SFI UnTrust Code
UnTrust Code
Trust Code
Application could use kernel provided services by System Call
Kernel Space
Device Drivers and Kernel Modules

SFI SandBox

- PNaCl would download the whole execution environment (with dynamic libraries)
- Would use x86_64 environment as the verification sample.
  - Each x86_64 App would use 4GB memory space.
  - But for ARM App, it would only use 0-1GB memory space.
- x86_64 **R15** Registers would be defined as "Designated Register RZP" **(Reserved Zero-address base Pointer)**,and initiate as a **4GB** aligned base address to map the UnTrust Memory space. For the UnTrust Code, **R15** Registers is read-only.
-
  -

RSP/RBP Register Operation

- The modification of 64bits RSP/RBP would be replaced by a set instructions to limit the 64bits RSP/RBP would be limited in allowed 32bits range.

```
……
10001e0: 8b 2c 24              mov (%rsp),%ebp
10001e3: 4a 8d 6c 3d 00        lea 0x0(%rbp,%r15,1),%rbp
10001e8: 83 c4 08              add $0x8,%esp
10001eb: 4a 8d 24 3c           lea (%rsp,%r15,1),%rsp
…..
```

Function Call

* The function target address would be 32 bytes alignment, and limit the target address to allowed 32bits range by R15.

```
…. ..
 1000498:     83 e0 e0          and   $0xffffffe0,%eax
 100049b:     4c 01 f8          add   %r15,%rax
 100049e:     ff d0             callq *%rax
…. ..
```

* For the internal UnTrust function directly calling, it doesn't need to filter by the R15
  * vRet=987*testA(111);

```
…. .
10004bb:     e8 c0 fe ff ff         callq 1000380 <testA>
10004c0:     69 c0 db 03 00 00      imul  $0x3db,%eax,%eax
…. .
```

Function Return

- The function return address would be 32 bytes alignment, and limit the target address to allowed 32bits range by R15.

```
…. ..
10004e8:     83 e1 e0          and    $0xffffffe0,%ecx
 10004eb:     4c 01 f9          add    %r15,%rcx
 10004ee:     ff e1             jmpq   *%rcx
…. ..
```

For Hacker’s View

Conclusion

* LLVM support IR and could run on variable processor platforms.
* Portable native client + LLVM should be a good candidate to play role in Android and Browser usage. (in SFI SandBox)
* It is a new security protection model, use user-space Sandbox to run native code and validate the native instruction without kernel-level privilege involved.

Appendix

The differences of Dalvik and LLVM (1/2)

* From compiled execution code
  * LLVM transfer to 100% native code. Dalvik VM need to based on the JIT Trace-Run Counter.
* From the JIT native-code re-used
  * After Dalvik VM process restart, the JIT Trace-Run procedures need to perform again. But after LLVM application transfer to 100% native code, it could run as native application always.
* From CPU run-time loading
  * Dalvik application need to calculate the Trance-Run Counter in run-time and perform JIT. LLVM-based native application could save this extra CPU loading.

The differences of Dalvik and LLVM (2/2)

- From the run-time memory footprint
  - Dalvik application convert to JIT native code would need extra memory as JIT-Cache. If user use Clang to compile C code as BitCode and then use LLVM compiler to compile the BitCode to native assembly, it could save more run-time memory usage.
  - If Dalvik application transfer the loading to JNI native .so library, it would need extra loading for developer to provide .so for different target processors' instruction.
- From the Storage usage
  - General Dalvik application need a original APK with .dex file and extra .odex file in dalvik-cache. But LLVM application doesn't need it.
- From the system security view of point
  - LLVM support pointer/function-pointer/inline-assembly and have the more potential security concern than Java.

NaCl Source Code

- NaCl is salt
- Download the native client source code
  - http://code.google.com/p/nativeclient/wiki/Source?tm=4
  - cd $NACL_ROOT
  - gclient config http://src.chromium.org/native_client/trunk/src/native_client
  - gclient sync

http://code.google.com/p/nativeclient/issues/list

Native Client Page Content

```
<html>
<body ...>
....
  <div id="listener">
    .....
    <embed name="nacl_module"
        id="hello_loda"
        width=200 height=200
        src="hello_loda.nmf"
        type="application/x-nacl" />
  </div>
</body>
</html>
```

```
{
 "files": {
  "libgcc_s.so.1": {
   "x86-64": {
    "url": "lib64/libgcc_s.so.1"
   },
   .....
  },
  "main.nexe": {
   "x86-64": {
    "url": "hello_loda_x86_64.nexe"
   },
   .....
  },
  "libdl.so.3c8d1f2e": {
   .....
  },
  "libc.so.3c8d1f2e": {
  .....
  },
  "libpthread.so.3c8d1f2e": {
   .....
  },
 "program": {
  "x86-64": {
   "url": "lib64/runnable-ld.so"
  },
  .....
 }
}
```

End