CIS 4360
Secure Computer Systems

Sandboxing:
Google Native Client

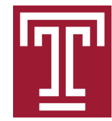
Professor Qiang Zeng

Spring 2017

Some slides are stolen from Intel docs

Previous Class

- SGX ensures confidentiality and integrity of data and code in an enclave
- Only the Intel processors and the enclave need be trusted
- Secrets are provisioned (through e.g. SSL) to enclaves only after they have been attested



When you want to run trusted code to process security-sensitive data, what can you rely on?

Intel SGX

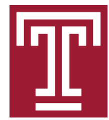
When you want to run untrusted code, what should you use?

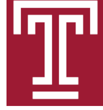
Sandboxing is a good option

Google's Native Client is an example of Sandboxing technique

What is Native Client?

- Native Client is a **sandboxing** technique for running compiled C and C++ code in the *browser* efficiently and securely, independent of the user's operating system





We already have JavaScript, which is safe and portable; why do we still want Native Client?

Performance (code is executed natively rather than interpreted)
Numerous legacy C/C++ code can be migrated to Web

•

C/C++

NaCl Module

JavaScript

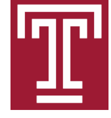
HTML

CSS

JavaScript

HTML

CSS



Life Demo

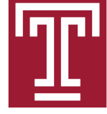
www.googleio2011.com/life/life.html

NaCl sandbox

```
sub    ebx,eax
shr    eax,1
mov    ecx,ebx
sar    ebx,1fH
and    ebx,ecx
add    eax,ebx
sub    edx,eax
```

Browser
- HTML
- CSS
- Javascript

Pepper

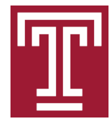


Wait! Downloading native code from Internet and running it in my browser? Are you insane?

Your concern is valid. That is exactly why we need a sandbox for running native code in the browser

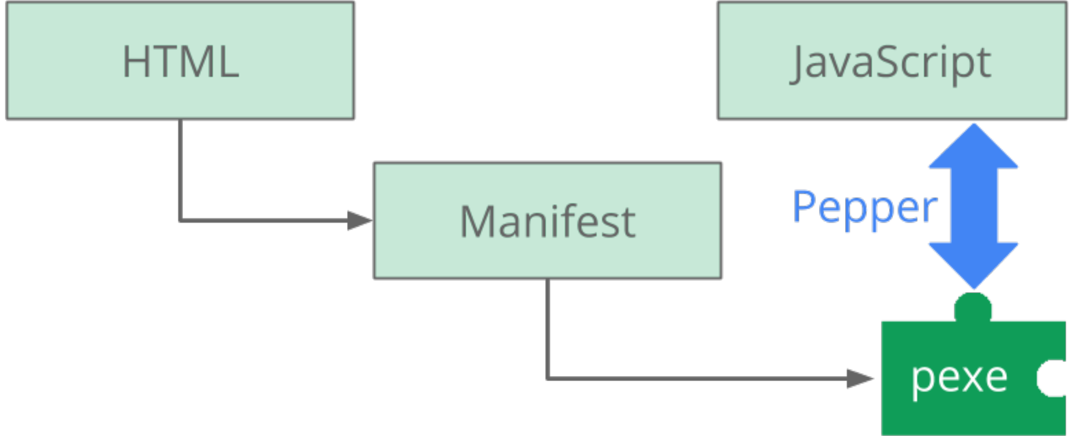
Benefits of Native Client

- Native Client gives C and C++ code the same level of **portability** and **safety** as JavaScript based on the sandboxing technique
- **Legacy code** can be easily migrated to the Web
- Meanwhile, Native Client enables high **performance** for applications such as
  - 3D games
  - Multimedia editors
  - CAD modeling
  - Data analytics
- Users use these applications **transparently** without installing plugins



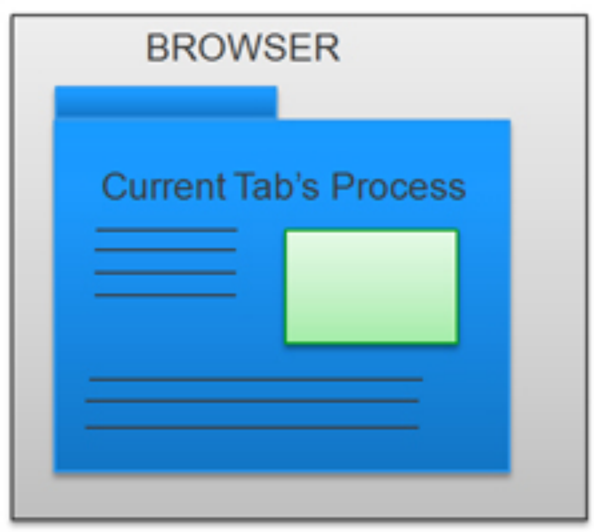
System Architecture

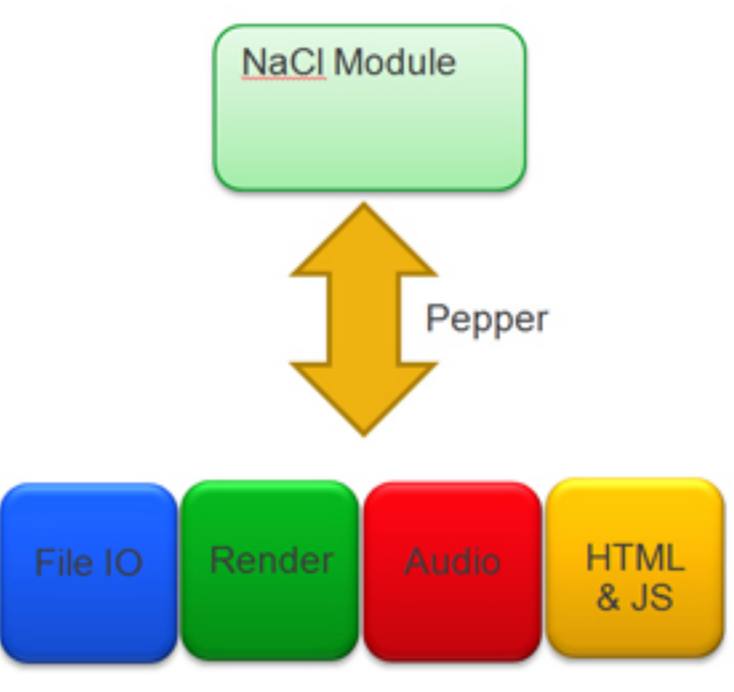
```
<embed name="mygame" src="mygame.nmf"
type="application/x-pnacl" />

{...
  ...
  "url": "mygame.pexe",
}
```

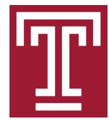


- The Native Client module cannot make OS-level calls directly
- Instead, it has to make use of a set of APIs called Pepper plug-in API (Pepper for short) to talk to the outside
  - Talking to the JavaScript code in the web page
  - Doing file I/O.
  - Playing audio.
  - Rendering 3D graphics.

Pepper Bridges NaCl Module and Others

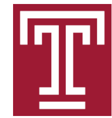






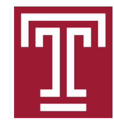
System Architecture

- Let's take a web-based photo processing app as an example
  - The untrusted user interface is implemented in JavaScript
  - The untrusted image processing code is in a NaCl module.
  - The trusted Native Client engine (provided by Google) has to be first installed as a browser plugin by users and the NaCl module will be run in a sandbox by the engine
- The JavaScript code and NaCl module run in separate processes and communicate through IMC (Inter-Module Communications)

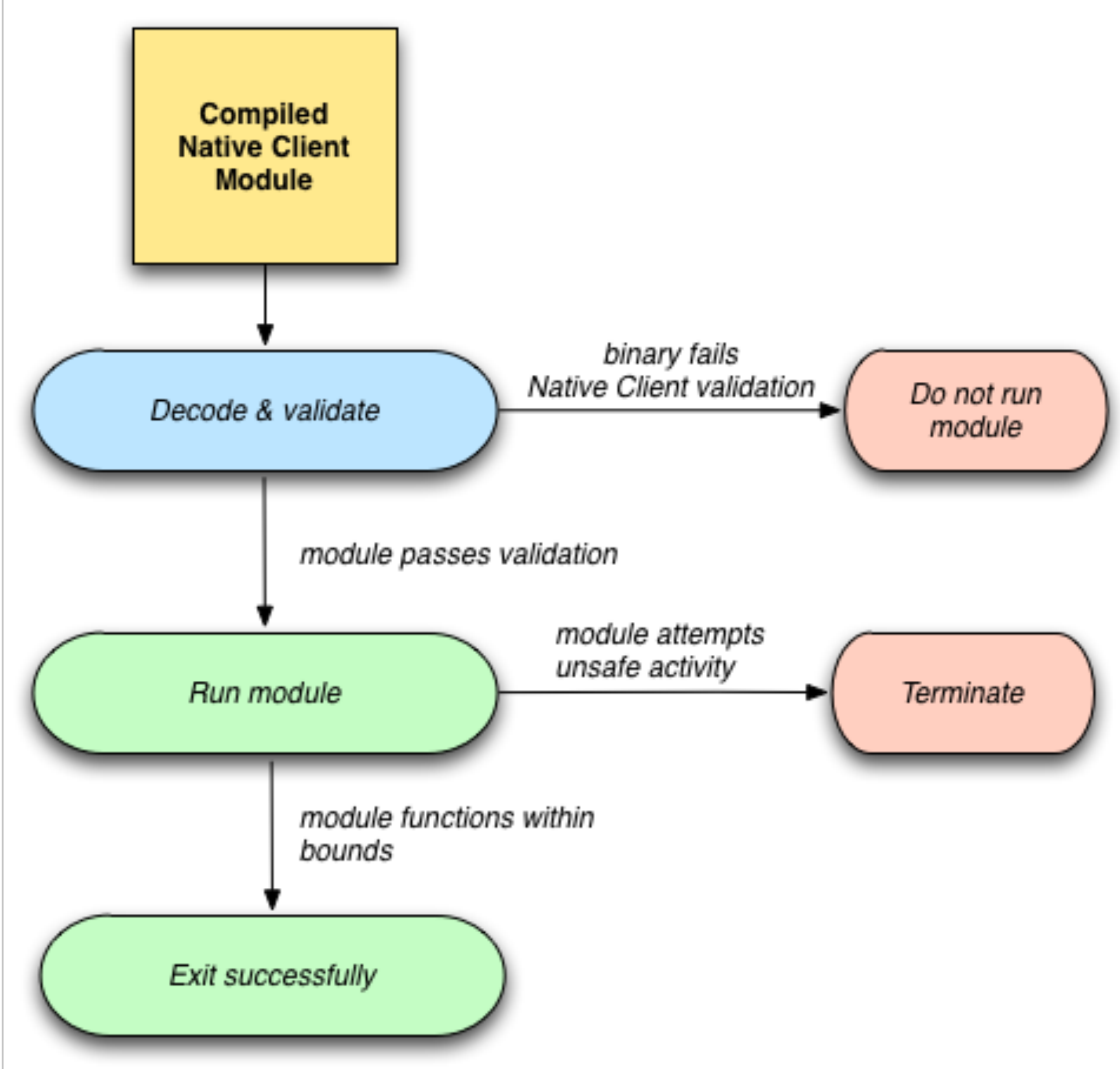


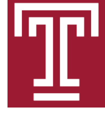
Security

- Since Native Client permits the execution of native code on client machines, special security measures have to be implemented:
  - A NaCl validator statically analyzes code before running it
  - A NaCl module can never access the outside except for through a set of APIs
  - Only a set of APIs in a whitelist is provided
- In addition to the sandbox above, the only interaction between the NaCl process and the outside world is through defined browser interface. Thus, the Chrome sandbox provides another layer of Sandbox



Static Validation + Runtime Confinement





Sandboxing through Separate Processes

Chrome Tab Process

Chrome Main Thread

Init App | *SwapBuffers(context, Draw)* | *SwapBuffers(context, Draw)* | *SwapBuffers*

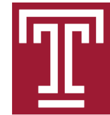
Pepper APIs | Pepper APIs

Draw | *Draw* | *Draw*

OpenGLES calls

Native Client Thread

Native Client Module Process

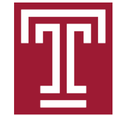


Static Validator

- The validator scans the binary code of a NaCl module from left to right
  - Any instructions in the blacklist are disallowed. E.g., syscall, lds
  - Check all direct jumps (e.g., jmp h30) and make sure the jump target is valid (e.g., not in the middle of a long instruction; not beyond the range for the NaCl module's text)
  - No instruction should be placed across the 32-byte block boundary (so padding is needed sometimes)
  - Check all indirect jumps and make sure they are

```
and %eax, 0xffffffe0
jmp *%eax
```

(Since no instruction go across the 32-btye block boundary, and the indirect jmp target is always 0 mod 32, the jmp will never go to the middle of an instruction)

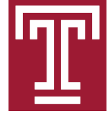


Why Jumping to the Middle of an Instruction is Hazard?

This 5-byte instruction means
AND %eax, 0x0000CD80;
It is benign

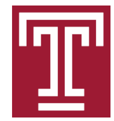
25 CD 80 00 00

But if you jump to here, “CD 80” means “int 0x80”, which can be used to issue a system call



Software Fault Isolation

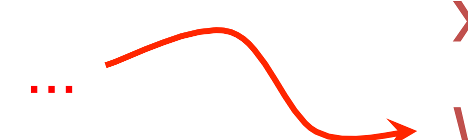
- The trusted NaCl runtime service and the untrusted NaCal module reside in the same process
- Note that the static validator does not check whether the data access or code execution go beyond the [0, 256) range for the NaCl Module
- Nor is any code instrumentation performed to enforce the range as classic SFI



Classic SFI

- Fault domain = from 0x1200 to 0x12FF
- Original code: write x
- Naïve SFI: x := x & 00FF

  x := x | 1200



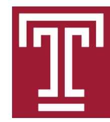
  write x

What if the code jumps right here?

- Better SFI: tmp := x & 00FF

  tmp := tmp | 1200

  write tmp



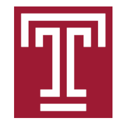
Hardware-assisted SFI

- In Intel X86, write x is executed as

  write (LDT[%ds].base + x) % LDT[%ds].len

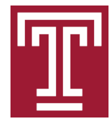
- LDT: local descriptor table; an array of entries, each having fields such as base, len etc.
- A segment register stores an index in to LDT
- Before the code in a NaCl module is executed, all segment registers (cs, ds, ss, etc.) point to an indexes whose corresponding LDT entries

  Base = 0, len = 256M (or less for cs)



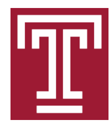
Jump between Trusted and Untrusted

- When a NaCl module invokes a pepper API, it needs to jump outside [0, 256)
- When JavaScript invokes a function of the NaCl module, the control should jump into the untrusted [0, 256)
- The [4K, 64K) range stores code for jumping between the trusted and untrusted worlds
  – Trampoline entries: it recover all segment registers to normal values so they can access $[0, 2^{32})$
  – Springboard entry: starts with “hlt”, why?
    - Invocation due to an jmp from untrusted code will crash the process
    - It can only be invoked by trusted code to jump back to the untrusted world by jumping to the instructions next to “hlt”; all segment registers will be set to “sandboxed” indexes



Summary

- Sandboxing through separate processes
  - The crash of the NaCl process will not crash your Chrome tab process
- Sandboxing through static validator
  - The control can only jump to the set of instructions that have been well analyzed
  - You can never jump to the middle of an instruction
- Sandboxing through Software Fault Isolation
  - Classic SFI is a little bit slow
  - H/w assisted SFI causes ~0 overhead



References

- Native Client Documentation
  - http://zqsmm.qiniucdn.com/data/20120628145922/index.html
- Native Client: A Sandbox for Portable, Untrusted x86 Native Code. S&P 2009
- Video by MIT
  - https://youtu.be/l0Psvvky-44