The major component of the system:
1. Mode 32: the code should not cross the region. It also helps in dis-assembling the code. Why we need to dis-assemble the code? To verify some properties of the code.

The major component of Verify:
- Dis-assembly
- 32 byte blocks
- No instruction crosses block
- All jumps checked
- All loads/store checked
- No syscalls
- No funny business
- Checks and operation are in the same block

Now we have,

<table>
<thead>
<tr>
<th>Trusted Region</th>
</tr>
</thead>
<tbody>
<tr>
<td>Untrusted Region</td>
</tr>
<tr>
<td>Trusted Region</td>
</tr>
</tbody>
</table>

Address Space

We want the untrusted region should never call the trusted region. The untrusted cannot jump to the trusted code. The way that the native client does it is it creates a table as shown below:
Untrusted Code region

Saved Space

Jump to trusted_open

Jump to trusted_read

Save register, save SP

All the jumps are to be 32 byte align

[Jump to the trusted code region, restore SP, restore register, return to caller]

The caller will save the registers.

The code cannot jump to outside the range and therefore cannot jump to the data. If it jumps to the data it can write new code that the verifier has never seen. Below the code segment is the trampoline. Trampolines lives inside the untrusted region. It is a software code where the untrusted region can jump. But this untrusted code would not be provided with the untrusted code. This code will be generated at the run time.

The trusted system provides 3 APIs.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>trusted_open</td>
<td>0</td>
</tr>
<tr>
<td>trusted_read</td>
<td>1</td>
</tr>
<tr>
<td>Trusted_write</td>
<td>_</td>
</tr>
</tbody>
</table>

This particular module only read trusted_open and trusted_read and do not use all of the APIs. This module will be loaded from a file.

What preventing the untrusted code to load libraries manually?

If we want load libraries then that can be done through mmap system call. Mmap open a further directory. Before if opens a directory i.e. before executing mmap, the verifier will check that there should be no system calls then we cannot do this.
The region above the code segment is not executable but writable and the region below code segment is executable and not writable.

When we load this untrusted code the loader after verifying the code will say it came from a larger file and the larger file will say that it is using two trusted functions. Then it will assign trampoline associated with first trusted function 0 and the other trampoline to be 1. Trampoline can be a simple jump instruction. It can say that jump to trusted_open. A jump like this is 4-5 bytes but all of the jumps has to be 32 byte aligned.

There is an important property in trusting the code, as we do not want the untrusted code to jump at arbitrary point inside the trusted code. Like this trusted_open

```c
Trusted_open(char *arg)
{
  /*do something*/
  real_open(args); //other function call
}
```

Suppose real_open and trusted_open are function calls inside the trusted code. If we allow untrusted stuff to call anywhere into the trusted stuff, then it has to bypass all the safety checks. So we do not allow the untrusted code to jump to arbitrary locations. So we have to give trampoline very specific doorways i.e. we will only allow this door or that door to enter into the trusted region. These doorways are called gates or caller gates.

Why 32 bytes size block?
If you use very large blocks then you will have internal jumps and if we have small blocks then we would have jumps outside the blocks. So 32 was the ideal size after experimenting with 16, 32, 64 bytes.

If supposed we are executing untrusted region on the stack and then we change the execution to trusted region. So the stack pointer shifts to the trusted stack. Now, if the untrusted is single thread then we cannot get through it. That would mean that if no other thread is executing that region then the trusted region could use that region of the trusted code. So we have to allow more than one thread to work in a region. When we jump from the trusted open we have to set up a stack.

When we call the function with argument then it copies the arguments. Kernel has also its own stack. So the trusted_open(...) has to do bunch of stuff:

1. Switch stack:
   Means that every thread has two stacks. One is untrusted stack and second is trusted stack. It has to switch it over to the other stack.
2. Copy arguments to trusted memory region
3. Do actual work
   4. Copy result into untrusted space.
   5. Return to untrusted code

Saved Space:
It has to save the register in the stack.

Can we modify trampoline?
No, we cannot change it. The code could have a template; the only thing that needs to be filled in is the address of trusted_open.

Is the trampoline code same for every untrusted region?
There is only one untrusted region, which is the model in the native client. If we have two different untrusted regions do we have same trampoline. If suppose we provide each trampoline for every trusted code. So if there are 500 trusted functions then there are 500 trampolines. We could have pieces of untrusted code. Then we have to define some policies that say that some untrusted code are allowed to call trusted_open and some are not allowed. So we have to have different policies. The policy has to be implemented in trusted_open. Then trusted_open has another parameter (extra piece of information).

If there are different pieces of code downloaded form different websites then they would have different ids. So it is the id that we have to pass to trusted_open.

```c
Trusted_open(char *arg, int callerid)
{
    /*do something*/
    real_open(args);
}
```

Who will generate this parameter (callerid)?
The trampoline does this.

The trusted_open need to know where the call came from. We need to pass the id no for the call. For example, it has to know where the call came from, like it came from google.com.

Another problem with having multiple untrusted regions:
The number of stack will grow if we allow calls from different untrusted region. One stack for each cross-domain.

Issue to resolve: copying data from untrusted region to trusted region-
If we break it in OS then there is 5% overhead. When we make a call there are few dozen instructions that gets saved on stack. Like copying arguments. It is going to take a 100 cycles. So if we have lot of data to copy then it will take more time. So it is still faster.
How can we come back from untrusted to trusted region?
Suppose when we download the code it will have the list of APIs that it can execute. For example, the way we specify an API, url/ trusted_open (function).
It says that load the native client binary block from this url to another untrusted region and then setup a trampoline within this url to make this call to trusted_open.

We can have JavaScript in the webpage that can call function. When we download the binary it will have some metadata that says load another binary as well. Suppose, I have something like NaCl file. The code is somewhere at the bottom. An import table has url and a function. The url is like adobe.com/(something) and the function call can be init_3d. So when the NaCl plugins calls our binary then it will go to the table and before it opens this code, it has to open another binary. Then it make a trampoline that will jump.

<table>
<thead>
<tr>
<th>Imports table</th>
</tr>
</thead>
<tbody>
<tr>
<td>url</td>
</tr>
<tr>
<td>adobe.com/</td>
</tr>
<tr>
<td>code</td>
</tr>
</tbody>
</table>

Conclusion: This system we can do cross domain calls which is as cheap as a system call but it is little bit more expensive. Once we get into the trusted code we have to copy the arguments.

Can we have Mutually dis-trusting snippet of code that can call each other and we do not have to do argument copy?
Yes we can do this using shared memory.
With the concept shown in the above diagram we can implement shared memory.

So there are three related areas called capability based systems:
1. Capability based memory
2. Capability based pointers
3. Object capability model
4. Capability based OS

Capability based memory and pointers are something that people do research on. We talked about the FAT pointers. So imagine we build a hardware that does the following, each word of memory has an extra bit and the bit indicates is this word part of a FAT pointer? If it is 1 then it is part of and if 0 then not a part of.

FAT Pointer

A word
This point is always 0 mode 32

<table>
<thead>
<tr>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

We have a tag on each memory. In the CPU we have two different type of registers, one is integers registers and second is pointer registers.
1. When we store from integer register to memory then it will clear the bit.
2. When we store a pointer register to memory then it will set the three bits.
3. If we load the integer to memory then
4. If we load something in the pointer register then it will check whether all the three bits are 1.
Then if we want to dereference this pointer then CPU has to:
1. Type check memory
2. Bound check pointer.

It would be nice to make instruction in which pointer will allow us to restrict it. This is called restrict instruction. We can reduce the upper bound or decrease the upper bound.

Now suppose we have two different codes in memory that do not trust each other. One wants to call another. So what we need to do? For example, we have a function, which return vector of integers and add them up very fast. U have to another function and use vector of integers and give a copy of the vector of integers to the first function such that it does not overwrite any other part of the code. Can we do that?

So we have beginning and ending vector of integers, then I have access to vector but I cannot construct the pointer from scratch. Only I can do is that I can dereference it or restrict its access. This is the general idea of capability.

Capability:
Unforgeable reference to an object. A subject has access to an object if and only if it has capability for it. We talk about access control matrices. We have subjects, which
has access to objects. When system can catch that access and look the ACM and decide whether it is valid or not. Think about file system. The way we identify an object is separated by file system. The name of the file is one thing and the permission is controlled by ACM.

A capability based system: is sort of like if we know the name we can access it. The names are not going to same and not readable like human names. A process is allowed to access memory if it has access to pointer. So, just handling a pointer will have access to memory.

Class Concluded