Isolation

6.1600 Course Staff

Fall 2023

In many settings when building systems, it is useful to *isolate* different components of a computer system. For example, cloud providers such as Amazon Web Services' EC2 run multiple virtual machines, your phone runs several apps, and your browser runs many sites together. If one of these virtual machines (or apps or websites) is malicious or buggy, these platforms would like to protect the buggy component from interfering with the execution of other code in the system. We often refer to separate isolated components as running in separate *isolation domains*.

Disclaimer: This set of notes is a work in progress. It may have errors and be missing citations. It is certainly incomplete. Please let the staff know of any errors that you find.

Ideally, we could have each isolation domain run on separate physical computer. If this were the case, (modulo physical side-channel attacks) one domain would clearly not be able to touch another's state. Unfortunately, the cost of this physical "air gap" isolation is far too high for most practical applications.

Computer systems for high-stakes applications (e.g., classified government data) do in fact use this type of "air gap" isolation.

So, when creating isolation methods, we aim to make it appear as if each domain is running on a separate computer but to do so all on the same computer.

1 *Defining Isolation*

In order to define isolation, we will think about two key properties: integrity and confidentiality.

1.1 *(Weak) Integrity*

One property that we would like an isolation scheme to provide is *integrity*: one domain cannot modifying the state of another domain. To formalize this, let's consider two domains running on a single host, an adversary domain A and a victim domain V. We would like to guarantee that A cannot change the execution of V. We can define something in terms of the *state* of each domain, S_A and S_V. For any pair of starting states (S_A, S_V), after running A, we would like the new pair of states to be (S'_A, S_V):

$$(S_A, S_V) \xrightarrow{\text{run}A} (S'_A, S_V)$$

1.2 *(Weak) Confidentiality*

We would also like *confidentiality*: an adversarial domain should not be able to read a data from any other domain. We can formalize this

by considering two worlds, each with a *different* victim state. After running A in each of these worlds, we would like the resulting S'_A to be identical. That is, for all pairs of victim states S_V^1, S_V^2, running (S_A, S_V^1) results in the same adversarial state as running (S_A, S_V^2):

$$(S_A, S_V^1) \xrightarrow{\text{run}A} (S'_A, -)$$

$$(S_A, S_V^2) \xrightarrow{\text{run}A} (S'_A, -)$$

This definition of confidentiality is often called "non-leakage".

1.3 Non-interference: Strong confidentiality and integrity

In our definitions of confidentiality and integrity so far, only the adversary domain runs. In a real system, both the adversary and the victim domain will run concurrently. Ideally, we would like to ensure that our confidentiality and integrity properties hold under interleaved execution of the adversary and victim.

To achieve this stronger notion of security, we can include an interleaving of A and V in one world—we would like for the resulting S'_A to be identical whether or not V runs.

$$(S_A, S_V) \xrightarrow[A,V,V,A,A]{\text{run } A,V} (S'_A, -)$$

$$(S_A, S_V) \xrightarrow[A,A,\ldots,A]{\text{run only } A} (S'_A, -)$$

This definition is often called *non-interference*. We can similarly strengthen our definition of integrity by requiring that S'_V is identical after running V whether or not A is run.

$$(S_A, S_V) \xrightarrow[A,V,V,A,A]{\text{run } A,V} (-, S'_V)$$

$$(S_A, S_V) \xrightarrow[A,A,\ldots,A]{\text{run only } V} (-, S'_V)$$

1.4 Non-interference is difficult to achieve

To achieve a non-interference style of isolation, an adversarial process A must not be able to determine whether there is a victim V process running alongside it concurrently. The challenge, though, is that whenever the adversary and victim *share limited resources*—such as CPU, RAM, network bandwidth, hard disk space, etc.—it is almost always possible for the adversary to determine whether there is a victim process running concurrently.

The information leakage across isolation boundaries as a results of resource contention is one type of *side channel* or *covert channel*. There is a vast literature on how to construct and exploit various types of side channels that leak information from a victim to an adversary.

Example: Memory Allocation. A real system will have some bound on the amount of memory available to it. After this memory is used, the system will be unable to allocate any additional memory. Consider a system with 16GB of memory and a victim process that allocates memory based on the value of some secret:

```
int secret;
malloc(secret);
```

An adversary could repeatedly try to allocate memory until the system tells them they cannot. By keeping track of the amount of memory they were able to allocate, the adversary can learn the secret: if the adversary is able to allocate 15GB, the adversary will know that the secret is 2^{27}, as the victim must have allocated 1GB.

Example: Execution Time. Consider another victim that runs some computation that takes a variable amount of time to finish depending on the value of a secret. By keeping track of how long the adversary takes to finish, the adversary can learn how much execution time the victim running on the same system takes to finish. Using this information, the adversary may be able to learn information about the secret. This type of information transfer are often called "timing channels", and can be quite tricky to work with.

There are effectively three ways to deal with the fact that non-interference is generally impossible to achieve with shared limited resources:

- **Strictly partition resources to prevent contention.** Each isolation domain could run on a separate physical machine, or we can provision the resources on a machine are partitioned in such a way that there is never contention for resources between isolation domains.
- **Prevent isolation domains from detecting resource contention.** Another (more practical) approach is to restrict the types of programs that can runs in such a way that prevents the programs in the system from detecting contention in shared resources. For example, if all programs in an isolated system are deterministic functions with no access to the outside world—no system calls, no networking, etc.—then programs may not be able to *detect* resource contention when it exists. In most implementations of isolation (e.g., virtual machines in a cloud environment), isolation domains absolutely need access to the outside world, so this approach is rarely useful.
- **Give up on non-interference.** Most isolation mechanisms opt for this solution. Rather than trying to achieve strict non-interference,

we aim for some "good enough" notion of isolation. Linux, for example, does not attempt to achieve strict non-interference between processes running on the same physical machine.

2 *Implementing Isolation*

In principle, implementing isolation in a system involves three main steps:

1. identify the state for each domain,
2. identify operations that access state, and
3. ensure that state-modifying operations can only read/write the state within an isolation domain.

The challenge is performance. An isolation mechanism will have to perform checks to ensure that components in one isolate cannot influence another. The game in isolation is to provide strong isolation at the minimum possible cost.

We will start by considering simpler isolation mechanisms and then look at more sophisticated ones.

2.1 Emulation

One simple way to implement isolation is to have an interpreter that executes isolated programs (e.g., x86 programs). The interpreter inspects each opcode in the program one at a time, and then implements the operations on the isolated state that the opcode indicates. While processing each opcode, the interpreter enforces checks on the isolated program to ensure that it can only modify its local state.

Some JavaScript engines ("runtimes") in web browsers use emulation for isolation. The Python interpreter is another example of emulation-based isolation. The runtime for these languages is designed such that the code can access only memory that belongs to the domain.

Benefits Emulation can be simple to implement and provides "good enough" isolation for many applications.

Downsides Emulation can be slow: to execute each logical opcode in the emulated program, the emulator may have to run a large number of physical instructions. Emulation can be inflexible: emulated programs may not be able to take advantage of special-purpose hardware, unless the emulator explicitly grants access to these devices to emulated programs.

2.2 *Time Multiplexing*

Another effective isolation strategy is to only allow the code from one isolation domain to run on the hardware at once. When the isolation mechanism switches from one isolation domain to another, it writes the state of the hardware (e.g., registers, RAM) to storage, clears the state of the hardware (e.g., zeros the contents of memory), and loads the next program to run into the machine.

For example, gaming consoles implement this form of isolation: one game runs at a time and has almost full control of the hardware during this time. The state of the running game cannot tamper with the state of a non-running game.

The security of this isolation scheme requires the isolation mechanism to protect the stored state of each isolation domain, and to somehow protect the code that switches between them.

Benefits Time multiplexing is relatively simple to implement, and it can give programs in each isolation domain almost full control of the hardware (e.g., graphics hardware in game consoles)..

Downsides There can be a high cost of switching execution between between isolation domains. Since the isolation mechanism clears the state of the hardware during context switches, storing and restoring the state can be costly.

2.3 *Translation (Naming)*

Translation is another common isolation mechanism in computer systems. In a system using translation for isolation, an isolated program may not access hardware resources (such as memory or files) directly. Isolated programs can only access resources via pointers controlled by the isolation mechanism. By construction, each process in an isolation domain can only name resources inside of its isolated context.

The canonical example of naming/translation for isolation is *virtual memory*. In a system using virtual memory, the isolated domains cannot read/write physical memory directly—they can only read/write to virtual memory addresses..

To prevent one isolated program from accessing another's memory, the isolation mechanism ensures that the valid virtual addresses in each domain point to a separate portion of physical memory. Modern CPUs have special support for implementing virtual memory to make the virtual-address translation as fast as possible.

Other examples of naming/translation isolation in computer systems are: file descriptors in Linux and virtual LANs in networking.

2.4 Example: Isolation in Virtual Machines

When we run several virtual machines on one physical machine, we want each virtual machine to run as if it had its own physical CPU, memory, and devices, but we want to run all of these virtual machines on a single physical machine. For performance, we would like to run instructions from the VM directly on the host CPU—but we need to make sure, for example, that the VM does not access memory belonging to another VM. To achieve isolation and good performance, systems today use several effective techniques.

Here is how a virtual-machine monitor handles the three questions that an isolation mechanism must answer:

1. **What is the isolation domain's state?** A few important pieces are:
   - the contents of memory,
   - the values in the CPU registers,
   - the data on disk.
2. **What operations can a program perform on isolated state?** Virtual-machine monitors need to handle:
   - CPU instructions that modify register states ,
   - CPU instructions that modify memory, and
   - operations to read and write devices.
3. **How does the virtual-machine monitor ensure isolation?**
   - To handle updates to the register state, the monitor uses *time multiplexing*: one virtual machine runs on the CPU at a time and controls the CPU's registers.
   - To handle accesses to memory, the monitor uses *naming/translation* via virtual memory.
   - To handle device operations, the monitor uses *emulation*. When a virtual machine executes instructions that would cause device I/O, the CPU jumps into some dedicated code in the monitor that emulates these device operations. This is slow, but since device I/O is usually costly anyways, the overhead is isolation is tolerable.

2.5 Software interposition

A final isolation technique that we will discuss is *software interposition*. Say that an isolated program wants to run the following code:

```
var a = b[c];
var f = ....;
f();
```

When using software isolation, a compiler can insert checks into this isolated program to make sure that the memory access `b[c]` does not access out-of-bounds memory:

```
if c >= b.size: error;
load b+c -> a             (***)
```

After inserting these checks, the isolated code can run directly on hardware, since the checks ensure that the isolated code cannot touch any state outside of its isolated context.

A central challenge of using software interposition is handling function calls. When calling the function `f` in the code snippet above, the hardware might execute an instruction like:

```
jump *f
```

which would execute the code at the location stored in memory location `*f`.

If an isolated program can set the value in `*f` to the location `(***)` in the snippet above, the isolated program can skip the safety checks on the memory accesses.