John Wood
CPSC 490 Spring 1011
Advised by Bryan Ford

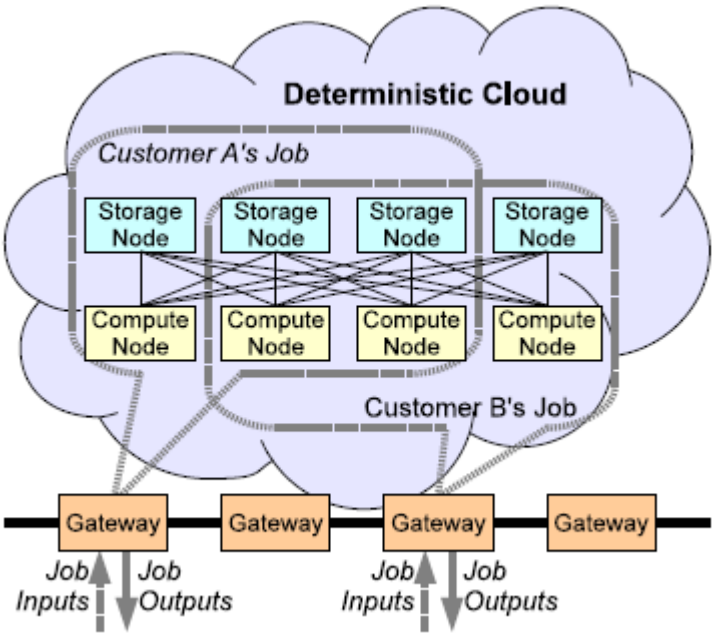
Determinating Mandelbrot:
Adding Provider-Enforced Deterministic Execution to the Cloud

Abstract

This project is based on the paper "Determinating Timing Channels in Compute Clouds" by Amittai Aviram, Sen Hu, and Bryan Ford of Yale University, and Ramakrishna Gummadi of the University of Massachusetts Amherst. The paper outlines a method of protecting against timing channel attacks in the cloud by using provider-enforced deterministic execution. I have implemented this defense using the research kernel Determinator, developed by the DEDIS group at Yale.

The basic idea of provider-enforced determinism is that you may trust the service provider (e.g. Amazon), but you don't necessarily trust fellow customers. If you are running on the same hardware as an adversary, it leaves you open to timing channel attacks. For example: Alice runs a cloud compute service where you can buy space and compute power on one of her machines. Bob opens an account with Alice's service in order to process some private financial data for his company, ACME. Eve also gets an account with Alice's service, and happens to get placed on the same machine as Bob. Bob, who is none the wiser, starts the very computationally intensive process of crunching his company's numbers. Eve, who wishes to discover information about ACME's finances, monitors the cache and CPU behavior of the machine that she and Bob share, and is able to glean some meaningful information from the results.

The solution to this problem that I have implemented is as follows: Cloud service providers such as Alice provide gateways, to which users submit job requests and all the necessary inputs for that job. The gateway sends the job off to be processed, and returns the results to the user. But the result is solely a function of the input. This means that Eve cannot possibly learn anything about Bob's job, because she must submit explicit inputs, and her result will depend only on those inputs, and not any timing information from Bob.



For this project, The gateway is an Apache server running on an Ubuntu box, which communicates with the Determinator kernel via Ethernet. The job is a Mandelbrot set viewer, which allows the user to pan and zoom into the Mandelbrot set. All computations are run on the Determinator kernel, which sends the results (via Ethernet) back to the Gateway to be displayed to the user.

1. Gateway

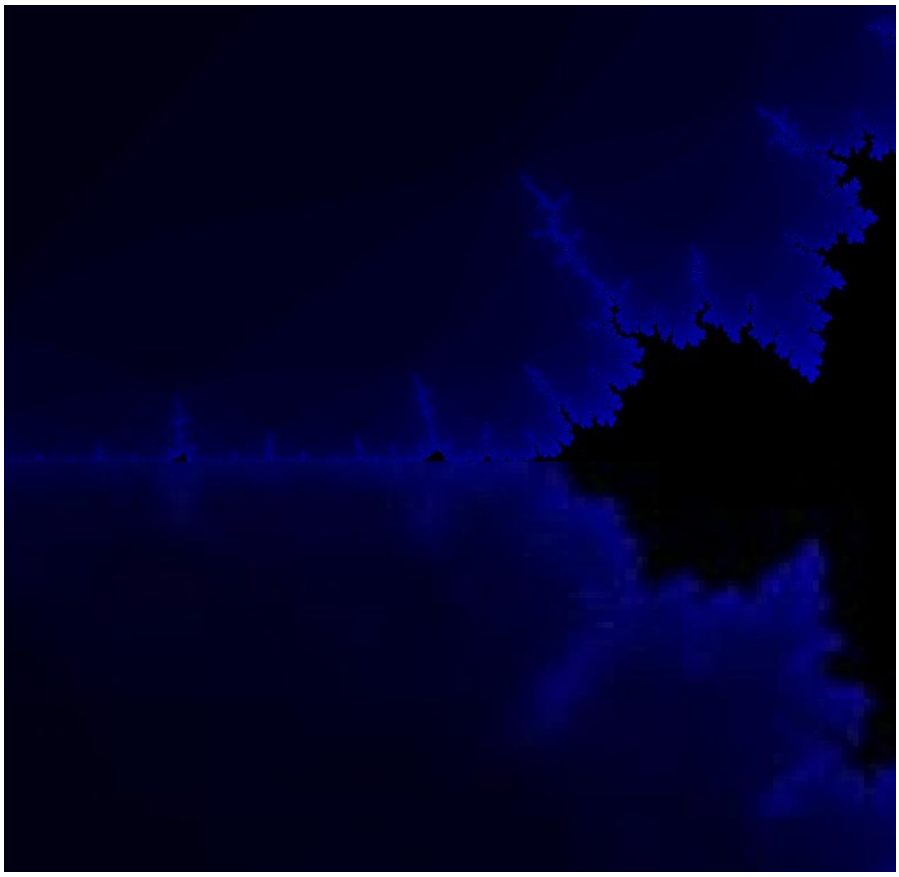
The gateway provides a front-end web interface. It is a simple Mandelbrot set viewer that allows the user to pan and zoom. When the user zooms in, a request is sent to the back-end. I have chosen a Mandelbrot set viewer to make things interesting. Some sort of computationally intensive manipulation of sensitive data probably would have made more sense given the context of timing channel attacks, but frankly the implementation would not be very exciting. Calculating the Mandelbrot set can become computationally intensive as the user zooms in closer and closer to the edge of the set, so this will actually serve as a good test in that regard. The Mandelbrot set computation can also be made extremely parallel, which makes it an excellent application to run on Determinator.

The gateway is running on an Ubuntu box with an Apache server. It has been written using PHP and JavaScript, the source code can be found here. The interface displays the set in 100px x 100px chunks, updating which chunks are displayed as the user pans.

When the user loads the page, the following code is executed:

```
1. exec("sudo php imsplit.php $zoom $color_str >> /dev/null 2>&1 &");
2.
3. $ethernet = '../ethernet';
4. exec("sudo $ethernet $zoom $res $color >> /dev/null 2>&1 &");
```

This call to `imsplit.php` splits a stock image of the Mandelbrot set into the required 100x100px chunks. The stock image is only 900px x 900px, so if zoom is >1, the image will be blurred. This is done to have "blur on zoom" functionality, so the user sees a lower resolution image while Determinator calculates the actual image. As you can see in the image to the right, the lower portion of the set is blurred and waiting to be calculated, where the top portion has been calculated already.



After the call to `imsplit.php`, `ethernet` is `exec`'d. The `ethernet` executable communicates with the Determinator kernel, sending a work request, and receiving the image chunks from Determinator. These chunks are then displayed to the user.

2. Communication

There is no TCP/IP stack in Determinator, so all communication is done via raw sockets. There were two protocols defined for this project. One is the kernel mode protocol, which will stay the same for all jobs, not just the Mandelbrot set viewer, and the other is the user mode Mandelbrot set viewer protocol, which is specific to this project. My implementation of this protocol for the gateway sending and receiving packets can be found in ethernet.c (ethernet.tar.gz), and the implementation of Determinator sending and receiving them can be found in kern/net.c and user/mandelbrot.c (determinator.tar.gz).

2.1 Kernel Mode Protocol

```
1.  #define NET_MAXPKT  1514    // Max Ethernet packet size w/o csum
2.  #define NET_ETHERTYPE_WORK  0x0800  // Claim this ethertype for our work requests
3.
4.  // Ethernet header
5.  typedef struct net_ethhdr {
6.    uint8_t   dst[6];   // Destination MAC address
7.    uint8_t   src[6];   // Source MAC address
8.    uint16_t  type;   // Ethernet packet type
9.  } net_ethhdr;
10.
11. #define NET_WORKDATA 1498 // NET_MAXPKT - sizeof(net_ethhdr) - sizeof(uint16_t)
12.
13. // Packet containing a work request
14. typedef struct net_work_pkt {
15.   net_ethhdr  eth;  // Ethernet header always comes first
16.   uint16_t data_len;
17.   char data_buffy[NET_WORKDATA];
18. } net_work_pkt;
```

2.1.1 Sending a Work Request

To send a work request to Determinator, a `net_work_pkt` is sent, and the `type` field of `net_ethhdr` must be `NET_ETHERTYPE_WORK`. The `data_buffy` field is filled with whatever user mode specific data is necessary for the job. The kernel adds `data_buffy` to a queue of work requests kept in kernel mode, and does nothing with the data. This makes the protocol extensible – if other jobs are implemented, they need only be implemented in user mode, and the kernel does not need to be recompiled.

2.1.2 Receiving a Response from Determinator

Determinator also sends it's response in a net_work_pkt (or multiple `net_work_pkts`), and once again, the kernel does not know what is in `data_buffy`. Determinator signals that it is finished sending by sending a `net_work_pkt` where `data_len` is 0.

Sending is accomplished with the system call `sys_send`:

```
1. sys_send(uint8_t *dst, void *buffy, size_t size);
```

`dst` is a pointer to the destination MAC address, `buffy` is a pointer to the data to be sent, and `size` is the size of the data pointed to by `buffy`.

2.2 User Mode Mandelbrot Protocol:

```
1. enum { BLUE, GREEN, RED };
2.
3. typedef struct mandelbrot_work {
4.   double zoom;
5.   int resolution;
6.   int color;
7. } mandelbrot_work;
8.
9. #define JPEG_BUFFY 1494 // NET_WORKDATA - 2*sizeof(uint16_t)
10. typedef struct jpeg {
11.   uint16_t x;
12.   uint16_t y;
13.   char buffy[JPEG_BUFFY];
14. } jpeg;
```

2.2.1 Sending a Work Request

In our case, we fill the `data_buffy` field of `net_work_pkt` with a `mandelbrot_work` struct. `Zoom` is self explanitory, `resolution` is a variable that changes how the set is colored, and `color` is either `BLUE`, `GREEN`, or `RED`. Lower `resolution` means that we are more strict about what is considered "inside" the set, and higher `resolution` means that we are less strict. Resolution is a bit of a misnomer, but has been kept for historical reasons.

2.2.2 Receiving a Response From Determinator

Determinator sends us `net_work_pkts` containing `jpeg` structs inside `data_buffy`. `X` and `y` are coordinates of each 100px x 100px chunk of the Mandelbrot set. When we have received all data for image (`x`, `y`), no data is send after `x` and `y`.

3. Determinator Back End

There have been both user mode and kernel mode changes to the Determinator kernel. Unfortunately, I cannot make the entire source code accessible, since Determinator springs from Pios, used to teach CPSC422, but if you are interested in obtaining the entire source, please email me at john.wood@yale.edu.

3.1 Kernel Mode Changes

Kernel mode changes to Determinator were made in 3 areas. Libjpeg was ported to run on

Determinator, networking was modified to implement the communication protocol described in section 2.1, and two system calls were added.

3.1.1 libjpeg

Much of the C library has been ported to run on Determinator, so the back end has been written with this limited subset, but libjpeg did need to be ported for this project. I ported libjpeg version 6b, which is an older version, but was easier to port than newer versions.

3.1.2 Networking

As mentioned in section 2.1, when the kernel receives a work request in kern/net.c, it is added to a queue. The work queue implementation can be found in kern/work.c. (determinator.tar.gz)

3.1.3 System Calls

Two system calls were added:

```
1. sys_send(uint8_t *dst, void *buffy, size_t size);
2. sys_getwork(void *user_buffy, uint16_t size);
```

`sys_send` has already been described in 2.1.2. `sys_getwork` removes the first item off of the work queue and copies `size` bytes of the data into `user_buffy`.

3.2 User Mode Changes

The only user mode changes were to write the application `mandelbrot` to run on the determinator kernel. The implementation can be found in user/mandelbrot.c (determinator.tar.gz).

`mandelbrot` waits for work to appear in the work queue. When work is received, it forks a child to do the computations described in the `mandelbrot_work` struct of the work request. It then begins to create a (`NORMAL*zoom`) px x (`NORMAL*zoom`) px image of the Mandelbrot set, where `NORMAL` is currently defined as 900px.

Mandelbrot calculates 100px x 100px chunks of the image in parallel and then sends them back to the gateway to be displayed.

Future Work

There are several areas of this project that could use improvement. The first of them is in the web interface. The way I have implemented panning, I track mouse movement, and then update the “src” field of each “img” tag based on the movement. The problem here is that the images change. For example, img000_000.jpg at first refers to the blurred version of the image created by imsplit.php (described in section 1). Then, when Determinator sends us the correct version, img000_000.jpg is overwritten with the new version, but the browser doesn't know that. No-cache headers are of no use in this instance either. So my solution is to append a timestamp to the end of each “src”, since when the browser sees a source that it hasn't before, it will query the server for the fresh image. The granularity of this timestamp is 3 seconds, so every three seconds, the server has to send an updated version of each image. This is a pretty big performance hit.

```
1. var seconds = Math.floor(new Date().getTime() / 3000);
2. imgs[row][col].src =
3.         "img" + origimgs[startrow+row-offset_rows][startcol+col-offset_cols] +
4.         ".jpg?seconds=" + seconds;
```

The second issue is in the back end implementation. Right now, only one gateway can access the Mandelbrot set viewer at a time, otherwise files will get overwritten on the server. A fairly easy way to remedy this is to append the host name and timestamp of each request to the image files on the server. So instead of naming them imgXXX_YYY.jpg (the current format), they would be img_ipaddress_timestamp_XXX_YYY.jpg. This would mean that no requests would conflict with one another.

And third, the implementation of the work queue needs some work before it is truly extensible. Right now, `sys_getwork` just gets the first item off of the work queue, but if we have more than one job, the queue will hold requests for many different jobs, and we won't be able to tell the difference. `sys_getwork` should take another parameter, `job_id`, and return the first work request for job `job_id`. This means that each job needs a unique `job_id`, and also suggests that a work request queue is no longer an appropriate data structure. Instead, we should use an array of queues, indexed by `job_id`, so that each job can have it's own work request queue.

Conclusion

This has been an extremely rewarding and interesting project. If you would like a demonstration, email me at john.wood@yale.edu.