Client Side Trusted Web-server Modules using Intel SGX

Ajay Brahmakshatriya and **Alexandra Zytek**
{ajaybr, zyteka}@mit.edu

Abstract

Intel's SGX enclaves allow execution of code on a client computer without risk of external processes reading or saving data within the enclave. We propose using SGX to allow arbitrary code (such as for a browser game) from a server to be executed on a clients computer without allowing the client to unfairly change the code or its outputs. This makes it possible for a small server to handle thousands of clients without a decrease in performance.

1 Motivation

Web applications are usually divided into two parts – client side components and server side components. The client side components handle aspects such as UI, capturing user inputs and actions, basic sanitation of input, and relaying input to the server for further processing. The server components handle authentication, maintaining records in databases, and interacting with other services. These modules are usually run on the server because the computation they perform has to be correctly done for the application logic. For example, if `foo.com` started checking the passwords on the client side, a malicious user could bypass this logic and simply return to the server that the user has correctly authenticated.

This model is usually fine for applications that are not latency sensitive. But with the introduction of near native speed execution in browsers with NaCl or Web Assembly, many game developers are making their games available to the users to be played directly in the browser. This allows users to quickly demo games without having to install a separate game client. Even when playing against an AI, the game logic has to run on the server so that the client cannot unfairly change game state and arbitrarily award themselves rewards or advantages. But this separation between the server and client has two main issues – Firstly, the communication between the client and the server introduces communication latencies which can range all the way from 15 ms to a few hundred ms (depending on the network quality and the number of hops between the client and the server). While this may be okay for turn based games like chess, it is completely unsuitable for real-time fast-paced games. The second challenge is that since the computation of the game state happens on the server side, the server resources (both memory and compute) have to scale linearly to the number of clients connected. For some very popular games maintained by small companies this could incur significant costs.

We attempt to solve this problem by allowing the server to send over to the client some modules of the server side logic and provide a guarantee that the logic is run exactly as it is supposed to be. Our solution comes as an API extension to the JavaScript engine in the browser that allows the JavaScript loaded from the server to create Intel SGX enclaves inside the browser, load arbitrary code and remote attest the code with the server. The JavaScript can then communicate with the running module through a defined interface while the trusted module can communicate with the server as required. The JavaScript can terminate the execution of the enclave at any time.

The game developers can now offload the game state computation to the client machine while also reducing the latency between the browser and the game server to order of a few microseconds. The game server can also trust the final state (user score, player profile changes) relayed to it from the trusted enclave and update it when the game ends. This allows a very small server to handle thousands of clients at the same time.

2 Criteria for Success

We define the following three goals for the project:

1. Successfully design an API extension to the JavaScript engine that allows the browser to create an SGX enclave, load and execute arbitrary code, and attest the code with the server.
2. Demonstrate that latency, bandwith utilization, and server utilization are improved by the design.
3. Show that the system is secure for the client.

We believe that goal 1 is the most important for a project in this context; however, goals 2 and 3 are necessary for making the infrastructure usable in real world.

3 Design

We developed a new API in JavaScript which browsers can choose to implement (depending on availability of SGX hardware and user preferences). The following types and function calls were added to the JavaScript API to facilitate creation

of the enclave module, communication with the module, and termination of the module.
The TrustedModule can be created and managed by the JavaScript using the following API:

```
@global
@module
TrustedModule = {
    @params url - URL for the image of the trusted module to be loaded inside the enclave
    @params remote_attestation_url - URL for the attestation server
    @returns the newly created TrustedModule or None if the creation failed
    create: function(url: string) : TrustedModule

    @intializes and runs the module
    run: function()

    @params message - The string message to be sent to the trusted module
    send: function(message: string)

    @params handler - Registers a handler to be called when a message is received from the trusted module
    onmessage: function(handler: function(message:string)
}
```

The trusted module has a C API that allows it to communicate with the JavaScript in the browser and the attestation server over an encrypted channel.

```

// An already initialized fd which has the browser on the other side
int websocket_fd;

// fd - the connection fd, should always be = websocket_fd
// message - a C string message to be sent to the browser
// size - size of the string to be sent
// returns 0 on success, -1 otherwise
int send_message(int fd, char* message, size_t size);

// fd - the connection fd, should always be = websocket_fd
// returns - a heap allocated string containing the message on success, NULL otherwise. The returned string if not NULL, should be free'd
char *recv_message(int fd);

// data - buffer of data to be encrypted and sent to the attestation server
// size - size of data to be sent
// return 0 on success, -1 otherwise
int send_backend(uint8_t *data, size_t size);

// size - variable to get the size of the received data in
// returns the data received from the attestation server on success, NULL on failure
uint8_t *recv_backend(size_t *size);
```

We also provide a attestation library that automatically verifies the attestation report received from the browser, exchanges cryptographic keys and provides a library by which the attestation server can exchange messages with the trusted module. The attestation library guarantees confidentiality and integrity of the messages exchanged.

4 Implementation

We implement the feature of creating enclaves in a separate service called the `enclave_service` that is started with the browser. This service waits for incoming WebSocket connections from the browser and on receiving a connection, expects 2 URLs from the browser. The first URL is the URL of the image that should be downloaded and instantiated inside the enclave. The second is a hostname:port pair to connect to for remote attestation of the enclave. The `enclave_service` then forks a new process where it creates and initializes an enclave. It then connects to the attestation server and retrieves the attestation report from the client. After the attestation server verifies the report, it tells the host service to run the enclave. After this point, the only responsibility of the host service is to relay messages between the enclave, remote server and the browser. The messages between the browser and the enclave are exchanged in clear text and the messages to and fro the attestation server are encrypted using the cryptographic keys exchanged as a part of the attestation report. The root of trust of these keys are based in the enclave because the hash of the keys is signed and included as a part of the report.

This guarantees confidentiality and integrity of messages for the server and the enclave. But we also need to make sure that the enclave cannot hijack the execution of the `enclave_service` and steal client secrets because it runs in the same address space of the host service. We are okay with the execution of the host service being hijacked as long as it cannot do anything malicious to the user's system. To guarantee that, we restrict the system calls the host service can make to just read and writes (and a few related ones for memory mapping). This is done with the use of `seccomp` 2 filters. These filters are inserted just after the enclave is created but before calling any function within the enclave. This ensures that once the enclave code starts executing, no malicious system calls can be made.
We build our host and enclave library on top of the Microsoft Openenclave framework 3. This includes the `enclave_service`, the attestation server and the actual enclave itself.

5 Testing Performance

As mentioned in Section 2, we want to evaluate our infrastructure on 2 fronts. Latency and server resource utilization.

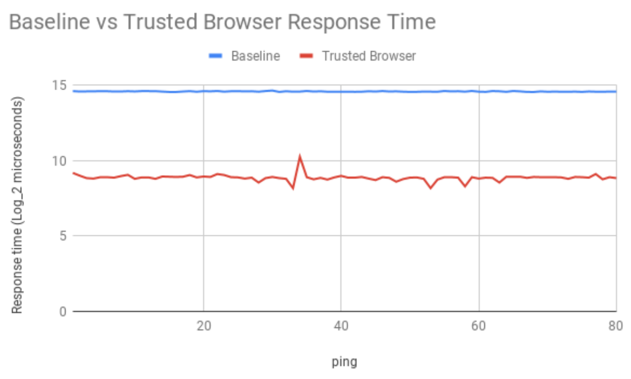

Figure 1: The response time across pings was significantly improved when using the trusted browser. Note the log scale of the y-axis.

To achieve this we created two applications and measured various parameters.

5.1 Echo Server

This is a simple ping application where the browser sends subsequent 80 ping requests to the server at an interval of 100 ms. The time for the response is measured in microsecond granularity. The server component of this application just echoes back whatever messages it receives. We build this application in two ways. One using traditional server side computation and second using trusted modules. Both the modes have different server and JavaScript components.
Figure 1 shows the latencies of the subsequent requests for both the modes on a log scale. We can clearly see that the average latencies of the traditional echo server are around 52 times higher than that of the latencies from the trusted module. This is possible because the communication is directly over the local loop back interface to the enclave running on the same machine. This drastic improvement in latencies can greatly improve the experience for browser games. Note here that the backend communication here is minimum. The attestation server simply establishes the identity of the enclave and sends it a message it to start replying to the pings.

5.2 K-means++ clustering

To measure the resource utilization of the server, we need an application that would have significant load on the server both in terms of CPU and bandwidth usage. This is characteristic of applications where the client sends over large amount of data, which is processed and sent back to the user. For example, proprietary video processing algorithms or other scientific computations wherw the server doesn't want to share it's algorithm with the user and wants to charge them for every use.
We pick one such example, the K-means++ clustering algorithm 1. This algorithm separates out a bunch of points in 2D space into user supplied K number of clusters. This being an iterative algorithm takes significant computation resources.
For our experiments we repeatedly cluster 1000 points into 11 clusters. The browser code here instead of taking data from the user, generates random coordinates for a 1000 points, serializes them and sends them over to the server. The server classifies them and reports the time for clustering. We create 10 parallel requests, each of which send a new one to replace them once a response is received.
Figure 2 shows the CPU utilization of the server for both the baseline and our framework. It is easy to see the start difference in the CPU usage. This directly would translate to higher throughputs and the server being able to handle more clients at a time. Figure 3 shows the bandwidth utilization of the server. The baseline has more network bandwidth consumption almost all the time. There is a large peak in the beginning for our framework at initiation time, because the enclave image is being downloaded. Currently the way our framework is implemented, the image size is very large because we have to statically link the entire binary. But this can be controlled by sending only the actual server code and then merging with the libraries on the client side. The signature of the entire file would still remain the same.
Reduction in both these factors enable the server maintainer to reduce the cost of the running the service while incurring very little cost to the client (cost for clients is distributed over all the clients).

6 Side notes

While implementing our technique we noticed a major flaw in the remote_attestation samples provided with Openenclave. Which was that they send the public key of the enclave to the server (with it's hash signed in the report). And then use RSA public key cryptography to exchange messages. The drawback of this technique is that it doesn't provide integrity of the messages sent to the enclave because the host also has access to the public key. We raised an issue on the Openenclave repository regarding the same. The maintainers acknowledged the bug, but decided not to fix it because a new release with full TLS support is soon coming out. But we fixed this issue while using with out system by generating an ephemeral symmetric AES-256 key, then signing with the enclaves private key and encrypting it with the remote servers public key and sending it to the server. Then we use this ephemeral key to do all the communication after that point. This guarantees both confidentiality and integrity of the communication.

References

1. K-means++ clustering. (2019, April 28). Retrieved from https://rosettacode.org.
2. Seccomp filters - http://man7.org/linux/man-pages/man3/seccomp_init.3.html.
3. Microsoft Openenclave - https://github.com/Microsoft/openenclave.

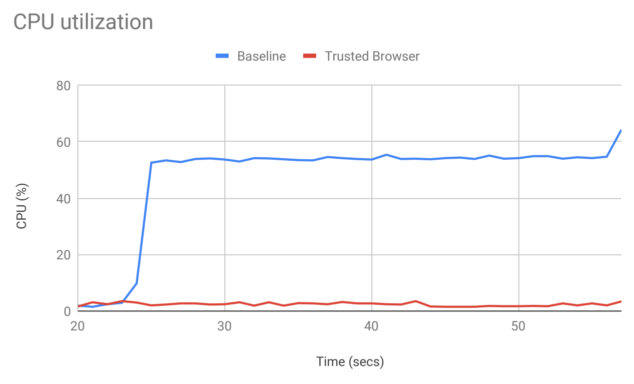

Figure 2: The CPU utilization for the server in percentage over time.

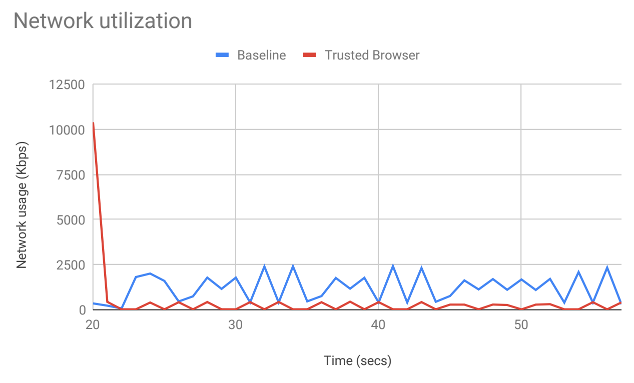

Figure 3: The Network utilization for the server in Kbps over time.