Secure Computation Interfaces

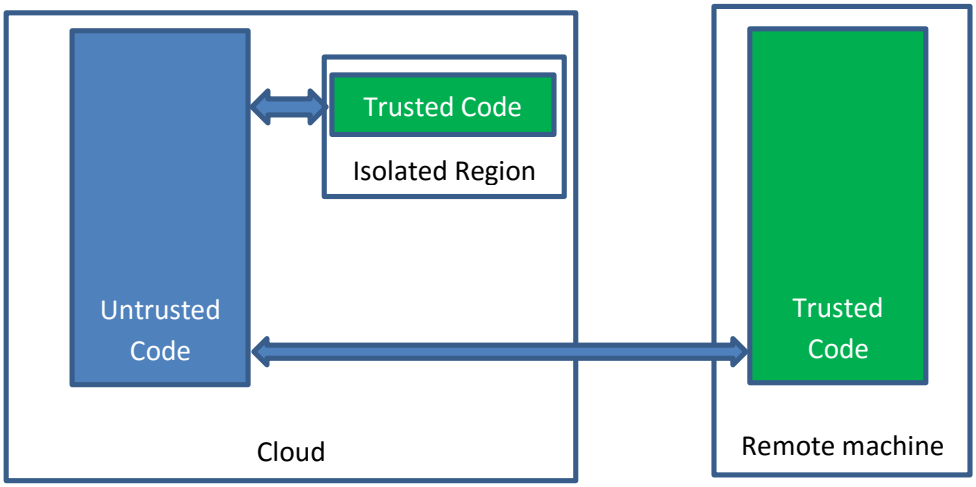
Manuel Costa, Orion Hodson, Marcus Peinado, Sriram Rajamani, Mark Russinovich, Kapil Vaswani

Introduction

Applications such as secure Hadoop [1] need to have part of their data and code isolated from privileged software (e.g., the operating system), and they need to be able to establish secure communication channels between the isolated code and remote machines. This document investigates the APIs needed to develop these applications. These APIs isolate the applications from the underlying hardware or software that provides "secure computation" [2, 3, 4, 5]. For example, the applications should work independently of whether isolation is implemented using SGX [6], or using Hypervisor-based isolation techniques [2, 3, 4, 5].

Overview

This document investigates a primitive programming model for secure computation on top of which more complex ones can bootstrap. We believe this programming model can be simple (we discuss only 8 functions). The main capability that we want to make available to applications is to create a memory region that is isolated from the operating system (i.e. the operating system cannot access the region); this is similar to an isolated process provided by virtualization infrastructures [2, 3, 4, 5], or an SGX enclave [6]. Such a region may contain both code and data; once created, the region can only be accessed by its own code. The main advantage of this feature is that even if the operating system is



compromised or operated by a malicious administrator, the attackers cannot access the data and code in the isolated region.

Figure 1 – Secure computation in a cloud provider scenario

To make the discussion concrete, we use a typical scenario for secure computation in a cloud provider, but the secure computation interfaces discussed here can be used in other scenarios. Figure 1 depicts the cloud provider scenario.

We describe how the untrusted code in the cloud creates an isolated region with some code provided by the user, and how the trusted code inside the isolated region communicates with the code outside. We also describe how the trusted code in a remote machine can establish a secure channel with the trusted code inside the isolated region.

Untrusted Code Outside the Isolated Region

Untrusted code needs to be able to create an isolated region and put some code and data in it:

```
HANDLE
IsolatedRegionCreate(
    _In_     LPCTSTR            packagePath,
    _In_     ISOLATION_PROVIDER isolationProvider,
    _In_opt_ CALL_OUT_HANDLER   callOutHandler
    )
```

This function creates an isolated region and loads the package specified by the packagePath argument into that region. The package is a container of code and data similar to the app packages used by Windows Store applications. The package also includes configuration parameters such as the size of the region. We assume the code in the package is self-contained and has no dependencies on the operating system; we implemented Hadoop with this restriction. All communication with the outside world is done through the untrusted code; if the trusted code wishes to protect secrets it encrypts them before passing them to the untrusted code. The isolationProvider identifies the underlying provider of secure computation services, e.g., VSM. The `callOutHandler` identifies a function in the untrusted code that can handle IO control codes sent from inside the region.
Untrusted code also needs to be able to invoke code in the isolated region. The simplest way to achieve this is to send an IO control code to the region:

```
IRIO_RESULT
IsolatedRegionIoControl(
    _In_      HANDLE  region,
    _In_      DWORD   callInId,

    _In_reads_bytes_opt_(inputBufferBytes)
              LPCVOID inputBuffer,
    _In_      SIZE_T  inputBufferBytes,
    _Out_writes_bytes_to_opt_(outputBufferBytes, *bytesReturned)
              LPVOID  outputBuffer,
    _In_      SIZE_T  outputBufferBytes,
    _Out_opt_ PSIZE_T bytesReturned
    )
```

The function that handles the control code inside the isolated region is identified in the package.

Finally, untrusted code can destroy an isolated region with:

```
VOID IsolatedRegionClose(_In_ HANDLE region)
```

Trusted Code Inside the Isolated Region

Code running inside the isolated region should be able to send IO control codes to code outside :

```
IRIO_RESULT
IsolatedAppIoControl(
   _In_        DWORD   callOutId,
   _In_reads_bytes_opt_(inputBufferBytes) LPCVOID inputBuffer,
   _In_        SIZE_T  inputBufferBytes,
   _Out_writes_bytes_to_opt_(outputBufferBytes, *bytesReturned)
       LPVOID  outputBuffer,
   _In_        SIZE_T  outputBufferBytes,
   _Out_opt_ PSIZE_T bytesReturned
    )
```

Code running inside the trusted environment should be able to get authentication codes or signatures for data that it wants to send outside:

```
BOOL
IsolatedAppSignMessage(
    _In_reads_bytes_(messageBytes) LPCVOID message,
    _In_ SIZE_T messageBytes,
    _Out_writes_bytes_to_(outputBufferBytes, *outputBufferBytesRequired)
         LPVOID outputBuffer,
    _In_ SIZE_T outputBufferBytes,
    _Always_(_Out_) PSIZE_T  outputBufferBytesRequired
    )
```

This function generates a signature for the buffer with secret keys available only to the provider of trusted computation. These signatures can typically be verified by code outside the isolated environment using the public keys of the provider of trusted computation.

Finally, code running inside the trusted environment should be able to get keys to encrypt data:

```
BOOL
IsolatedAppGetKey(
    _In_ KeyId keyId,
    _Out_writes_bytes_to_(keyBufferBytes, *keyBufferBytesRequired)
        LPVOID keyBuffer,
    _In_ SIZE_T keyBufferBytes,
    _Always_(_Out_) PSIZE_T keyBufferBytesRequired
    )
```

These keys allow the trusted code to encrypt data, save it in external storage, and then decrypt it in a subsequent execution.

Trusted Code on a Remote Machine

Trusted code on a remote machine needs to be able to establish secure communication channels with the isolated code. The base mechanism to achieve this is to be able to verify that a message originated from the code in the isolated region. This can be achieved with these functions:

```
BOOL
IsolatedRegionGetDigest(
    _In_ LPCTSTR packagePath,
    _Out_writes_bytes_to_(regionDigestBytes, *regionDigestBytesRequired)
         LPVOID regionDigest,
    _In_ SIZE_T regionDigestBytes,
    _Out_ PSIZE_T regionDigestBytesRequired
    )


BOOL
IsolatedRegionCheckSignature(
    _In_     ISOLATION_PROVIDER isolationProvider,
    _In_reads_(regionDigestBytes) LPCVOID regionDigest,
    _In_ SIZE_T regionDigestBytes,
    _In_reads_(messageBytes) LPCVOID message,
    _In_ SIZE_T messageBytes,
    _In_reads_(signatureBytes) LPCVOID signature,
    _In_ SIZE_T signatureBytes
    )
```

The `IsolatedRegionGetDigest` function returns a cryptographic digest that identifies the package. The digest can be passed to the `IsolatedRegionCheckSignature` function along with the identifier the secure computation provider. The function returns true if the message in the buffer was produced by the code with the given digest on an isolated region created by the secure computation provider. The contents of the signed/attested message can be used to establish shared secrets between the isolated environment and the trusted code in a remote machine in a variety of ways (e.g., Diffie-Hellman key exchange). We may want to support several of these secure channel establishment mechanisms in a library and let users choose which one to use.

Example Application: Secure Hadoop

We now describe how to implement a secure version of Hadoop based on the interfaces described above. In Secure Hadoop [1], users write map and reduce functions, we compile and encrypt those functions, and bind them together with a small amount of public code, producing a DLL called mapred.dll. We attach the user's public key to mapred.dll to allow the code to encrypt messages and send them to the user.

We also create an untrusted executable called fw.exe that interfaces with the Hadoop framework. Both mapred.dll and fw.exe are sent to the cloud; fw.exe loads mapred.dll into an isolated region with `IsolatedRegionCreate` and instructs mapred.dll to perform actions by sending it control codes using `IsolatedRegionIOControl`. We use control codes to run a key exchange protocol, and to run the map and reduce functions.

To establish a secure channel from a remote machine to the trusted code in the cloud, we run a "setup" Hadoop job that runs the key exchange protocol: when the trusted code runs, it generates a key with `IsolatedAppGetKey` and encrypts it with the user's public key. The code than signs the encrypted key using `IsolatedAppSignMessage` and outputs both the encrypted key and the signature. Code running remotely on the user's systems verifies the signature with `IsolatedAppCheckSignature` and decrypts the key. It then encrypts the keys for the secret code (the map and reduce functions) and data with the sealing key and sends the encrypted keys to the trusted code in the cloud. Communication between the user's systems and the cloud is done through the cloud file system in our implementation.

When processing data, the isolated code uses `IsolatedAppIOControl` with just two IO control codes (ReadKeyValue and WriteKeyValue) to Read/Write encrypted key-value pairs from/to the untrusted code.

Extensions

The programming model described above could be extended in several ways. First, memory management functions similar to `VirtualAlloc`, `VirtualFree`, and `VirtualProtect` could be made available inside the isolated regions to dynamically manage virtual memory. Second, functions to support threading and synchronization could also be made available. Also, note that other mechanisms based on inter-process communication primitives (shared memory, messaging interfaces, etc) could conceivably be used to communicate with the isolated regions, instead of, or in addition to, the IO control codes mechanism described in the preceding sections. Finally, note that all of these low-level communication mechanisms can be used as used as a basis to build richer communication primitives such as remote procedure calls (RPC).

We believe we should strive to keep the primitive programming model for secure computation small. More complex programming modes can be built on top of the primitive model. Keeping the primitive programming model for secure computation small is useful because not all applications will need/want the more complex models. In particular, the more complex programming models will typically require more code to be part of the Trusted Computing Base (TCB) and keeping the TCB small improves security.

Conclusion

The interfaces to support secure computation are simple. Defining these interfaces will allow us to develop trusted computing applications independently of the underlying provider of trusted computation.

References

[1] Felix Schuster, Manuel Costa, Cedric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich, VC3: Trustworthy Data Analytics in the Cloud using SGX, IEEE S&P 2015.

[2] Xiaoxin Chen , Tal Garfinkel , E. Christopher , Lewis Pratap , Subrahmanyam Carl , A. Waldspurger , Dan Boneh , Jeffrey Dwoskin , Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems, in ASPLOS 2008

[3] Owen S. Hofmann, Sangman Kim, Alan M. Dunn, Michael Z. Lee, and Emmett Witchel. 2013. InkTag: secure applications on an untrusted operating system. In ASPLOS 2013.

[4] Jonathan M. Mccune , Yanlin Li , Ning Qu , Zongwei Zhou , Anupam Datta , Virgil Gligor , Adrian Perrig, TrustVisor: Efficient TCB reduction and attestation, In IEEE S&P 2010.

[5] Device Guard Overview, https://technet.microsoft.com/en-us/library/dn986865(v=vs.85).aspx

[6] Intel Corporation, Software Guard Extensions Programming Reference, 329298-001US, September 2013, http://software.intel.com/sites/default/files/329298-001.pdf