Remotely Keyed Cryptographics Secure Remote Display Access Using (Mostly) Untrusted Hardware

Debra L. Cook[1], Ricardo Baratto[1], Angelos D. Keromytis[1]

Department of Computer Science, Columbia University, New York, NY, USA
{dcook,ricardo,angelos}@cs.columbia.edu

Abstract. Software that covertly monitors user actions, also known as *spyware,* has become a first-level security threat due to its ubiquity and the difficulty of detecting and removing it. Such software may be inadvertently installed by a user that is casually browsing the web, or may be purposely installed by an attacker or even the owner of a system. This is particularly problematic in the case of utility computing, early manifestations of which are Internet cafes and thin-client computing. Traditional trusted computing approaches offer a partial solution to this by significantly increasing the size of the trusted computing base (TCB) to include the operating system and other software.
We examine the problem of protecting a user accessing specific services in such an environment. We focus on secure video broadcasts and remote desktop access when using any convenient, and often untrusted, terminal as two example applications. We posit that, at least for such applications, the TCB can be confined to a suitably modified graphics processing unit (GPU). Specifically, to prevent spyware on untrusted clients from accessing the user's data, we restrict the boundary of trust to the client's GPU by moving image decryption into GPUs. This allows us to leverage existing capabilities as opposed to designing a new component from scratch. We discuss the applicability of GPU-based decryption in the two scenarios. We identify limitations due to current GPU capabilities and propose straightforward modifications to GPUs that will allow the realization of our approach.
Keywords: GPUs, Encryption, Thin Clients, Video Conferencing.

1 Introduction

Spyware has been recognized as a major threat to user privacy. Especially when combined with a large-scale distribution mechanism (such as a popular web site or application, or a computer worm), the potential for large-scale security violations is considerable. Organizations increasingly spy on their employees' computer activities using the same technology, and public computers on Internet cafes are so riddled with such malware that only the most foolhardy of souls would use them for any sensitive application.

Work on addressing this problem has focused either on detection of spyware activity on a system or building a trusted system from the bottom-up, using a combination of hardware support, operating system extensions and application-specifi c logic. While promising, these approaches offer only limited security against an adversary that legitimately controls the spyware-infected system, or against spyware that does not exhibit

real-time activity (*e.g.,* consider a program that simply takes snapshots of the system's screen as the unsuspecting user is accessing some sensitive information). While images, like any data, can be sent encrypted over networks using existing protocols such as TLS and IPsec, decryption is performed by the operating system, creating the potential for the data to be copied by an untrusted client.

We propose to use the system's Graphics Processing Unit (GPU) as the only trusted component in our spyware-safe system for displays. By using GPUs, we leverage existing capabilities within a system as opposed to designing and adding a new component to protect information sent to remote displays. Sensitive content is directly passed to the GPU in encrypted form. The GPU decrypts and displays the content without ever storing the plaintext in the system's memory or exposing it to the operating system, the CPU, or any other peripherals. We use a remote-keying protocol to securely convey the decryption key(s) to the GPU, without exposing them to the underlying system. With this mechanism as our basic block, we can implement applications such as secure video broadcasts or remote desktop display access without trusting the rest of the system.

Our work is an initial step of which the main purpose is to propose the concept and determine the feasibility of GPU-based decryption. We determine that, with careful design, current GPUs allow for in-GPU image decryption at rates sufficient to support the example applications. We also identify several obstacles to fully implementing our scheme on current GPUs. The most difficult aspect of moving decryption into a GPU is the API and the types of operations supported within the GPU. [4] demonstrated that the APIs for GPUs are not designed to support operations typically found in symmetric key ciphers. As a result, we do not focus on forcing an existing symmetric key cipher to fit within a GPU in order to decrypt the data, but rather implement as many operations as possible within the GPU and confine the remaining ones to a C program in order to illustrate the concept. In the future, either a cipher suited for GPUs and/or support for additional operations in GPUs is required. We have begun work on a stream cipher designed for GPUs and include an estimate of the performance. We identify straightforward additions to future GPU designs that will allow for the realization of our scheme, and its possible integration with the Trusted Computing Group's proposed architecture.

2 Motivation

Applications to which our work is relevant include remote desktops (a thin-client scenario) and video conferencing displays. In a thin-client scenario, the client connects to a server which fulfills all of the client's computing needs [11]. Since all application logic is executed in the server, the client is completely stateless, and does little more than display updates sent by the server and forward local user input events. Current thin-client systems provide secure sessions by encrypting the display protocol before it is transferred over the network. However, in scenarios where the client terminal is untrusted, such as public computers, it may not be desirable for the host operating system to have access to the unencrypted display updates. Consider the system described in [8], wherein access to sensitive 3D data was controlled by manipulating the content sent to the remote display client. Since the display data on the client cannot be secured, a number of additional mechanisms are devised to prevent the actual client application from

being used as an attack tool on the system. In contrast, if the current display is only in decrypted form within the GPU, we only need to block reads by other applications.

In video conferencing, we wish to prevent clients from copying the conference displays. How to secure video recorded at the client and audio is beyond the scope of this paper, although the concept we demonstrate with GPUs can also be applied to digital cameras and digital signal processors. While there are existing digital rights management (DRM) architectures aimed at preventing unauthorized copying of video, the images are still decrypted within the remote and untrusted OS. DRM includes how to manage the usage and trade of material [6], and must protect against both unauthorized access and unauthorized copying. An example is Microsoft's Windows Media Player DRM 9 Series, which includes the capability of authenticating and remotely-keying the media player [10]. The images are decrypted within the operating system by the media player then sent to the GPU. This architecture's security depends on using a specific closed-source media player and no program being able to access the memory utilized when decrypting the data. Alternative models of using trusted GPUs have been considered [2], but none has been implemented to our knowledge. The Trusted Computing Group's scope includes untrusted clients but its proposed architecture utilizes distinct trusted platform modules (TPMs), which may be hardware or software, to address multiple needs and provide a generic solution [14]. For graphical applications, our approach can be considered as an alternative that avoids specialized system components, or as a companion to TPMs. In particular, one possibility is for the TPM to handle key negotiation with the remote server, and then provide the session key to the GPU. We should note that similar concerns arise when handling voice traffic, as noted in [15].

Our main goal in moving decryption of graphics into the GPU is to prevent the underlying operating system or other software from gaining access to the unencrypted data. Specifically, we consider malacious software running on the client's operating system which attempts to read or modify displays and responses transmitted between the server and the client. We do not address modifications to the client's hardware, such as altering of the GPU. Furthermore, security of the client's surroundings (*e.g,* a camera recording the client's display) is a separate problem outside the scope of our work.

3 Prototype

3.1 Architecture

Figure 1 depicts our overall architecture. A server encrypts the data and sends it to the client. The data remains encrypted until it enters the GPU where it is decrypted and displayed. The GPU's buffer is locked to prevent the display from being read by processes external to the GPU, effectively turning the frame buffer into a write-only memory. The decryption is performed via software running on the client's operating system which issues commands to the GPU (as opposed to a compiled program existing and executing entirely within the GPU's memory), with the operations performed within the GPU. This software does not have access to the keys and data contained inside the GPU; rather, it specifies the transformations (*i.e.,* decryptions steps) that the GPU must undertake. Ideally, any intermediate data produced by the decryption program, such as the keystream, are confined to the GPU. We explain in Section 4 why this is currently not possible with existing GPUs.

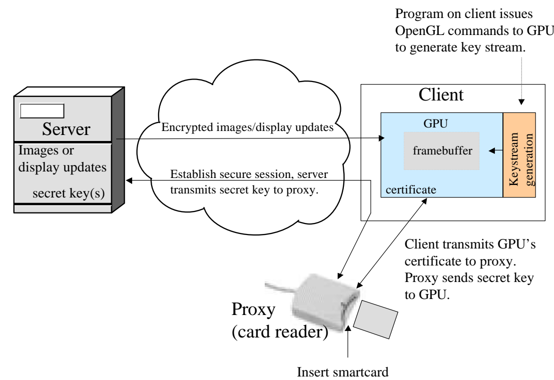

Fig. 1. Architecture for Remotely Keyed Decryption in the GPU

The decryption key changes on a per-session and application basis (and may even change within a session). Thus, the key must be conveyed to the GPU in a manner that prevents the client's operating system from gaining access to it. One way to achieve this is to remotely key the GPU and decrypt the key therein. The key is used to generate the keystream directly within the GPU, exposing neither the key nor the keystream to the OS. The decryption of the key and generation of the keystream can be performed in a non-visible buffer (back buffer) on the GPU, to avoid visually displaying them. Reading the encrypted image into the back buffer with the logical operation of XOR enabled results in the image being decrypted. The result is then swapped to the front buffer to display the decrypted image to the user.

There are a few possibilities for how the entities involved are authenticated and how the key is sent to the GPU, depending on which components are trusted. In each case, it is assumed that the GPU contains a pre-installed certifi cate and private key. The certifi cate may be issued by the manufacturer and hardwired in the GPU. Another option is to allow writing the certifi cate to the GPU under circumstances when the client's OS is trusted, such as when the GPU is fi rst being installed on a newly confi gured client. The fi rst and simplest option for authentication covers the case when the server sending the images is trusted and there is no need to verify the person viewing the images (*i.e.,* it is assumed that the fact the viewer was able to start the process on the client indicates it is safe to send the images) and/or the server is capable of authenticating a GPU based on its certifi cate. The server, either by establishing a session key with the GPU or using the GPU's public key, encrypts the secret key and sends it to the GPU via the client. The second, more general scenario, also assumes the server is trusted but requires verifi cation of the user viewing the images through a proxy entity, such as a smartcard reader. The user will activate the proxy by inserting a card into the smartcard reader attached to the untrusted system. The proxy will then establish sessions with both the server and remote system with the GPU. The server will convey the secret key to the GPU via the proxy, as shown in Figure 2. The process of converting the key from being encrypted under the server-proxy session key to being encrypted under the proxy-GPU session key requires that the key be exposed only on the smartcard. The proxy and the GPU treat the underlying system, including the OS, as part of the network connecting them to each other and the server. A third scenario assumes that neither the server nor the client OS are trusted. When the images are encrypted, the

encryption key is recorded on a smartcard. The encrypted images can then be stored on any server. To view the images on an untrusted system, the smartcard is inserted into a card reader (the proxy) or the key can be manually recorded and entered into the proxy. The proxy, using the GPU's public key, encrypts the secret key and sends it to the GPU via the client. The proxy does not have to be collocated with the client, but only has to be capable of exchanging information with the client. If a secret key only works for n blocks (such as n frames) of data, the remote keying will occur as needed to provide the key for each data segment.

The protocols used for the remote keying are not new. Refer to [1] and [5] for a discussion on authentication using smartcards. The novel component of our work is implementing one in a manner that avoids exposing the secret key outside the GPU. Any protocol used for the remote keying requires utilizing an asymmetric encryption algorithm to either encrypt the secret key directly with the GPU's public key or to establish a session key which is then used to encrypt the secret key. Obstacles arise due to the lack of support in GPUs for the operations required for public key ciphers, such as modular arithmetic for large integers. We discuss the limitations of the GPU in regards to public key cryptography when describing our prototype.

3.2 Implementation

To determine the feasibility of our scheme, we implemented the second scenario with 3 entities: a server, a proxy and the client. We use a stream cipher, RC4, to encrypt the images because of the rate of encryption required for streaming video. The prototype implemented as many operations as possible in the GPU via OpenGL, with the remaining operations restricted to a C program and which would be moved into a suitable GPU as we discuss in Section 4. Specifically, existing stream ciphers cannot be efficiently implemented entirely in OpenGL. We use the following notation:

- $K = k_1, k_2...k_n$ is the set of secret keys used to encrypt the data. k_i encrypts the ith subset of data. These keys may be individually pre-determined, or computed through a master key using a pseudorandom function.
- A frame refers to one frame of video or one display update.
- Rekeying refers to obtaining the next k_i. The interval at which rekeying occurs depends on either the number of frames displayed or the elapsed time.
- $r =$ is the number of frames or requests after which rekeying is required.
- $t =$ is the amount of time before rekeying is required.
- $sk =$ the session key used for communication between the server and proxy.
- $k^{pubk} =$ the GPU's public RSA key component.
- $k^{privk} =$ the GPU's private RSA key component.
- $m =$ the GPU's RSA modulus.

Figure 2 illustrates the steps for the remote keying and decryption of images in our prototype. A certificate containing a RSA [13] key is stored in the GPU's memory. For our prototype, a program on the client uses OpenGL to write the certificate to the GPU then deletes it from the operating system's memory. Installing a certificate in the GPU in this manner requires that the process be monitored to ensure that no program on the client gains access to the private key component of the RSA key while it is being

written to the GPU. The certifi cate includes a public parameter containing an indication that the device is a GPU. When the application is started, the client's OS reads the public information from the GPU's certifi cate and sends it in a request to the proxy. The proxy, which requires activation either by entering a one-time password or inserting a smartcard, authenticates the GPU based on the information encoded in its certifi cate.

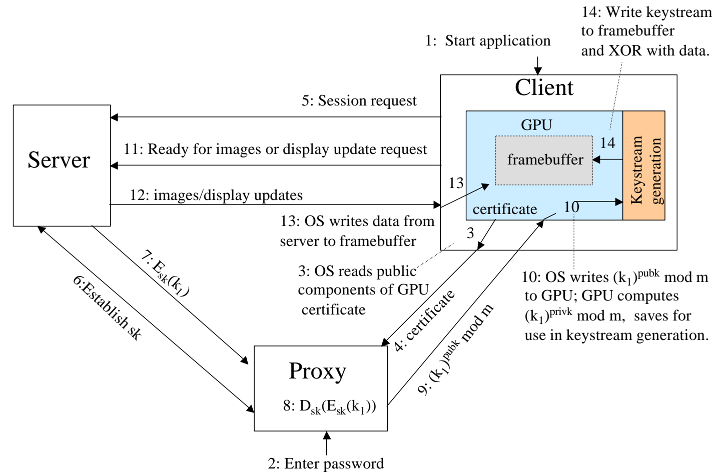

Fig. 2. Remotely Keyed Decryption in GPU Protocol Shown: logical links

The client also sends a connection request to the server. The server contacts the proxy and a secure session is established between them. This can be accomplished using any protocol designed for secure session establishment. A single session key may be used for the entire session, or the session key can be changed periodically, depending on the protocol. In our prototype, the proxy authenticates the server based on the latter's certifi cate, and uses a single session key, sk. When contacting the proxy, the server sends a random nonce and its certifi cate containing its public key for RSA. The proxy generates a random nonce, encrypts it with the server's public key and sends it to the server. The server and proxy both concatenate the two nonces and use a hash of the result as sk. The server sends k_1 encrypted with AES using key sk to the proxy. The proxy decrypts k_1, encrypts it with the GPU's public key and forwards the result, k_1^{pubk} mod m, to the client. The client issues the OpenGL command to turn color mapping on then writes the value received from the proxy to a specifi c pixel location in the GPU. The color map corresponds to x^{privk} mod m, where x is the value being written, and results in decrypting the value from the proxy to obtain k_1. The write operation is done to the GPU's back buffer to avoid visually exposing the resulting pixels (and annoy the user with unnecessary interference). As we explain later, we use a series of one-byte values for each k_i. The resulting pixels are used as the key to the stream cipher.

The client then signals to the server that it is ready to receive data or, for thin-client applications, makes a request to update a display. The server sends the encrypted data to the client. Ideally, the GPU computes the keystream, writing the resulting bytes directly

to the GPU's back buffer. As explained in Section 4, when using RC4 some C code is used to represent operations that will be performed in the GPU if improvements are made to the GPU's API. The client issues the OpenGL command to turn the logical operation of XOR on in the GPU, then writes the data received to the back buffer. The result is the data XORed with the keystream. The buffers are then swapped so the unencrypted image appears on the display. It is common practice to create an image in the back buffer then swap it to the front buffer in order to create a smooth transition between frames. After n frames or t time, the client must signal to the server that it needs the next secret key, sk_{i+1}, which is conveyed via the proxy as before.

Our prototype uses images encoded with 24 bits per pixel using 8 bits for each of the Red, Green and Blue components. No Alpha component is encoded since the image is written to the back buffer (which may not support the Alpha component) to be decrypted. The pixel format is a parameter used by certain OpenGL commands, such as the Draw command for writing data to the GPU, and can easily be changed to accommodate other pixel formats.

4 Design Decisions

We now discuss some of our design and implementation decisions that were guided by the constraints of existing GPUs. We first describe the limitations on programming a GPU to perform general keying and decryption operations, and then discuss the current inability to provide data compression.

GPUs are not designed to perform general arithmetic and byte-level operations. We refer the reader to [3] and [4] for background on GPU APIs and pixel processing, including the types of operations supported which are relevant to ciphers and the limitations of GPUs in performing byte level operations. There are no API commands for common operations such as modular arithmetic, shifts and rotates. Some operations can be performed by a sequence of other commands under certain circumstances, such as limiting values to a single byte and reading intermediate results from the GPU to the operating system to allow the result to be a parameter in a subsequent command. We describe how these limitations impact the ability to remotely key the GPU and decrypt data within the GPU, and the workarounds we used to create our prototype. We conclude that three enhancements to OpenGL are necessary to fully realize our architecture. First, a means of performing modular multiplication on values of magnitude typical of those used for public key ciphers is required to securely implement the remote keying. Second, a mechanism for using the contents of a pixel (or pixel component) as a parameter to an OpenGL command without first reading the pixel value from the GPU is required for the remote keying and keystream generation. Third, the ability to perform modular arithmetic using values less than 256 directly (*i.e.* without using color maps) is desirable to efficiently implement certain ciphers, such as RC4, within the GPU.

4.1 Remote Keying

The lack of modular arithmetic and limitations on the range of values in GPUs impacts the implementation of the asymmetric cipher used in the remote keying. The proxy conveys the secret keys to the GPU via the client's OS using an asymmetric key cipher.

Since existing public-key algorithms require exponentiation and/or modular arithmetic, the operations required cannot be emulated in the GPU with existing APIs, except when trivially small values are used, or when the values involved can be viewed as a series of 8 bits values. The remote keying of the GPU requires only that the GPU be able to perform the decryption function of the asymmetric algorithm. We note that unless the proxy and GPU share a secret key in advance, any protocol used to exchange information, whether by merely having the proxy encrypt information with the GPU's public key or by establishing a session key between them, requires use of an asymmetric cipher.

We considered two options for our prototype. First, the operations can be implemented in C code to represent a function that should be in the GPU. Second, restrictions can be imposed on the size of the asymmetric cipher's components to allow it to be implemented to run in the GPU. However, in the case of RSA this requires that plaintext and ciphertext each be restricted to fi t in within a single byte, thus requiring the modulus and exponents also each fi t within a single byte and resulting in key components too small to be secure. To illustrate the concept of decryption using public key cryptography within the GPU, we used "toy" values less than 256 in the prototype for the private and public exponents and the modulus. We used a series of 8-bit values to represent the data, *i.e.,* the secret key for RC4, encrypted with RSA. Each is encrypted with RSA by the proxy and sent to the GPU. When using RC4 as the keystream generator, up to 256 single-byte values can be in the series for RC4's secret key.

A third possibility is the integration of a decrypting GPU with a TPM such as the one proposed by the Trusted Computing Group. This chip could handle certifi cate storage and handling, as well a remote attestation and key negotation. Our GPU can then handle image decryption using the TPM-negotiated session key.

4.2 Decryption of Data in the GPU

To decrypt the images received from the server, the GPU on the client must run a symmetric key cipher. As we described previously, we use a stream cipher. We consider two options for the stream cipher: using an existing stream cipher and designing a stream cipher suitable for a GPU. With respect to running an existing cipher within a GPU, operations typically found in symmetric key ciphers make this infeasible either due to the nature and number of OpenGL commands required to emulate the operations or due to the infeasibility to convert the operations to execute within the GPU given limitations of the API [4]. Existing stream ciphers, such as LILI, RC4, SEAL, SOBER and SNOW, are unsuitable for implementation in a GPU. We chose to use RC4 because it is possible to implement using OpenGL, though not practical due to the specifi c OpenGL commands required resulting in poor performance. The use of irregularly clocked feedback shift registers in LILI and SOBER, and 32-bit words in SNOW and SEAL, among other operations such as 9-bit rotations in SEAL, make these either less attractive than implementing RC4 or impossible to implement in OpenGL.

The operations in RC4 consist entirely of adding two bytes, modulo 256 and swapping two bytes. Thus, the only operation required of RC4 which is lacking in a GPU is modular arithmetic. Since the modulus is 256, all values can be represented by single bytes and can be stored as individual pixel components. Given two integers, a, b in the range [0,255], $a + b$ mod 256 can be computed using a color map. This requires

knowing either a or b in advance to determine which color map to activate. For each integer, a, in the range [0,255], create a color map where the ith entry corresponds to $a + i \bmod 256$. To compute $a + b \bmod 256$, b is stored as a pixel component, the color map for a is activated, then the pixel containing b is copied to a new location. The result written to the new location will be the bth entry of the color map. This poses two problems. First, while OpenGL is used, the command to activate a color map must be issued by a program running on the operating system, requiring a to be exposed to the operating system. While this does not expose the keystream to the OS, it does provide partial information to the operating system, which may be helpful in determining keystream values. Second, the copying of pixels between locations in the buffer is one of the slowest operations within GPUs. In addition to the copy needed to compute the sum, copies are needed to update the indices and move bytes into the appropriate pixel components and locations. As a result, implementing RC4 in OpenGL is not a practical option. Therefore, we opted to implement the keystream generator of RC4 in C to represent a function that will eventually be moved into the GPU. The keystream bytes are written to the GPU as they are computed. This requires the C function computing the keystream to read the secret key from the GPU. We initially wrote each byte of output from RC4 directly to the GPU as it was generated. However, the number of writes required (750,000 for a 500x500 image) resulted in poor performance. We changed our prototype to compute the keystream bytes for an entire row of pixels before writing them to the GPU, reducing the number of writes to the height of the image with the tradeoff that a segment of the keystream is temporarily stored outside the GPU.

Due to the inability to efficiently generate a keystream within a GPU by using an existing stream cipher, we are investigating designing a stream cipher utilizing graphics operations for which GPUs are designed. We briefly describe the concept here. By mapping a texture exhibiting sufficient randomness to a continuously morphing image while changing certain variables, such as viewpoint and lighting, and extracting pixels from the image, a keystream is generated. The keystream is never within the client's system memory in this case. We experiment with an initial version in order to estimate the time to compute the keystream, with the results shown in Section 5. We point out that while creation of a new stream cipher suitable for current GPUs is feasible (and in fact may have wider applicability than our applications), the same is not true for public-key ciphers, since this would require devising a new one-way function that does not require exponentiation and modular arithmetic on numbers larger than a single byte.

While the proposed approach protects the secrecy of the images sent to the untrusted system, the integrity of these images is not protected. This could allow an attacker to change parts of the image, although this would be immediately detectable by the user, as it would produce corrupt output on the screen (since the attacker does not know the session key). Adding a message authentication code (MAC) to our scheme is not currently feasible due to the limits of current GPUs.

5 Experiments

We conducted two sets of experiments to measure the ability of current GPUs to sustain decryption rates compatible with our example applications. We used OpenGL as

the API to the graphics card driver. We did not use any vendor-specifi c OpenGL extensions, making our prototype GPU-independent. We used GLUT to open the display window. The only requirement is that the GPU must support 32-bit "true color" mode, as the routine for decrypting the secret key requires representing bytes in a single-pixel component. The code for the client consists of C, OpenGL and GLUT, compiled using Visual C++ version 6.0. The processes for the server and proxy are written in JAVA.

The experiments utilized three different clients in order to test different GPUs. The environments were selected to represent a fairly current computing environment, a laptop and a low-end GPU. In all cases, the display was set to use 32-bit true color with full hardware acceleration. The clients are:

1. A Pentium IV 1.8 GHz PC with 256KB RAM and an Nvidia GeForce3 Ti200 graphics card with 64MB of memory, running MS Windows XP. The GPU driver uses OpenGL version 1.4.0.
2. A Pentium Centrino 1.3 GHz laptop with 256KB RAM and an ATI Mobility Radeon 7500 graphics card with 32MB of memory, running MS Windows XP. The GPU driver uses OpenGL version 1.3.425.
3. A Pentium III 800 Mhz PC with 256KB RAM and an Nvidia TNT32 M64 graphics card with 32MB of memory, running MS Windows 98. The GPU driver uses OpenGL version 1.4.0.

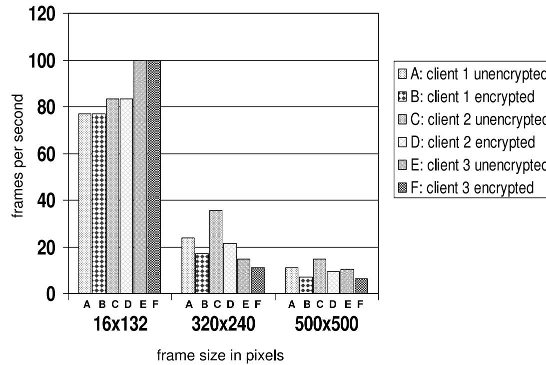

Fig. 3. All Entities on a Single System

We simulated streaming video applications, such as NetMeeting, by sending a stream of images from the server to the client. We tested with frame sizes of 320x240 and 500x500 pixels. The frames were encrypted and stored in individual fi les on the server prior to starting the application. To measure thin-client performance, we used the average update size of 2,112 pixels (a 16x132 pixel area) from the standard i-Bench [7] web benchmark for thin-clients. The update sizes in i-Bench range from 1x1 areas to 1,007x622 areas (626,354 pixels). All tests used images encoded as 24-bit RGB pixels, with 8-bits per color component.

For each image size, two types of tests were run. The fi rst set of tests determined the delay due to the additional computation needed for the remote keying and decryption, compared to sending unencrypted images. In these tests, all three entities (server, proxy,

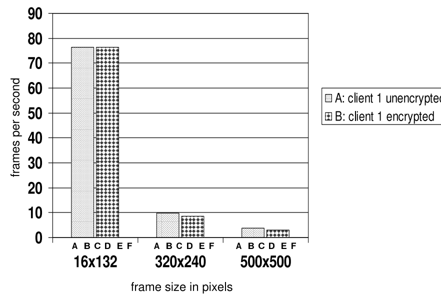

Fig. 4. Dedicated Lan and Client 1

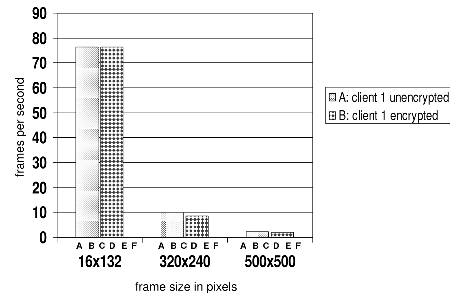

Fig. 5. Shared Lan and Client 2

and GPU) were run on the same PC or laptop. Each of the three clients was tested. The results of the first set of tests are shown in Figure 3.

The second set of tests involved running each entity on separate systems on a LAN to determine the overall performance when the data arrival rate was impacted by network delay. The first client with the Nvidia GeForce3 GPU was used for these tests. Figures 4 and 5 show the results of these experiments. Two tests were run using two different LANs. In one case, the server and proxy were dedicated to the experiment and there was no traffic leaving the server and proxy aside from that due to our experiment. In the second case, we ran our tests on shared servers used for general purpose computing. In both cases, each element had a 100Mbps connection to the LAN. There were three hops between the client and server, and between the client and proxy; there are two hops from the proxy to the server.

For all tests, the number of frames per second (fps) for both encrypted and unencrypted frames are provided. In video conferencing applications, the fps supported is important: a minimum rate of 10 fps is required to obtain tolerable video and is typical in such applications, with 24 fps and higher rates required for better quality. In contrast, the rate of updates in thin-client applications is dependent on user requests and will be sporadic. The fps reflects the maximum supported burst rate.

We note that it was not our intention to build a robust streaming video application using RTP which accounted for delay, rate of transmission and lost packets, but rather we focus on the remote keying and decryption within the GPU, and determine the resulting overhead. Therefore, TCP was used for all communication between the entities.

At least 99% of the delay when decrypting frames with RC4, compared to using unencrypted images, is due to the writing of the keystream bytes to the GPU. The keystream was written to the GPU one row at a time. When the test is run with the write eliminated (all other operations for the decryption are still performed), the average time is the same as that for the unencrypted images. The actual computation of the keystream per frame, enabling the logical operation of XOR in the GPU and swapping of buffers takes less than 1*ms* for the 500x500 frames on all clients. When testing the average thin-client display size update (2,112 pixels), the times for the encrypted updates were the same as for the unencrypted updates because the keystream required only 16 writes to the GPU. In contrast, the 320x240 and a500x500 pixel frames required 240 and 500 writes per frame, respectively.

The limiting factor in the processing of the 2,112-pixel updates is the time for the server to create the update (read the update from a file in our experiment). To determine the rate at which the client can process such updates if creation of updates is not a limiting factor, an array containing 2,112 pixels was stored in memory on the server and repeatedly sent to the client. The client can process over 500 updates per second on each of the three platforms, indicating that decryption overhead and the GPU are not limiting factors for small updates. For larger updates in thin-client applications, we do not consider an increased delay, *e.g.,* when the entire display changes, to be an issue; such updates are infrequent and, from a human perspective, are no worse than the loading of some web pages or opening of applications.

When sending images over a LAN, the decreased rate for the 320x240 and 500x500 pixel frames compared to the case when all processes were on the same PC is due to the rate at which images are sent from the server to the client being limited by the bandwidth. Even if no bandwidth is consumed by protocols, a maximum of 16.66 uncompressed 500x500 RGB frames can be transmitted per second on a 100Mbps interface.

To estimate the time required for computing a keystream designed for the GPU as described at the end of Section 4, we loaded an initial image in the GPU and measured the time to execute all of the OpenGL operations under consideration. After each series of executions, the resulting image is the keystream XORed with the current encrypted frame. The execution per frame is less then $1ms$, indicating that any differences in the time to process encrypted *vs.* unencrypted frames will be imperceivable.

The time for the remote keying is mainly dependent on the time to enter the password or insert the smartcard into the proxy, and may take up to a few seconds if a password must be entered. Aside from this, the time is dependent on the protocol used and on the transport delay between the entities. Using a public-key encryption algorithm, generating random nonces and encrypting the secret key with AES requires approximately two seconds in each environment.

6 Conclusions

We addressed the feasibility of decrypting images and displays within a graphics processing unit as a way of combating the rising threat of spyware. Our primary insight is that a suitably modified GPU can serve as a minimal trusted computing base for displays in certain types of widely used applications, such as video conferencing and remote desktop display access. The main mechanism in our scheme is decryption of frames exclusively inside the GPU, without storing either the key material or the plaintext on the system's main memory. Our technique can protect against many types of spyware, as well as several attacks aimed at the human interface layer [9].

We explained why this scheme cannot fully be realized due to current limitations of GPU APIs. We identified three straightforward enhancements to GPU APIs that can overcome these limitations. With our prototype, we demonstrated that the concept is feasible for thin-client applications and the video broadcast in conferencing applications. Designing a keystream which runs entirely in the GPU and takes advantage of typical graphics operations will eliminate overhead and improve performance. To further improve performance in these applications, image compression facilities will need to be implemented inside the GPU, a trend which is already occurring. In addition, our

numbers show that for typical video conferencing frame rates and web browsing using thin-clients, the lack of compression is not a performance bottleneck.

Our prototype focused on the securing of images sent to an untrusted client. Some additional items must be considered in a complete system that protects all inputs. For example, protecting any user keyboard and mouse inputs on the client which must be conveyed to the server. Also, depending on the application, audio may need to be encrypted in a manner that prevents the OS from accessing the plaintext. The types of operations supported by programmable DSPs make extending our concept to audio relatively easy. We refer the reader to the extended version of this paper [3] for a complete discussion. Other items discussed in [3] include proxy attacks in relation to our model, data compression when using GPU based decryption and server-side encryption when using a GPU based stream cipher. Future work includes developing prototypes that fully integrate the concept into thin-client applications and expanding the prototype to include encryption within DSPs.

References

1. M. Abadi, M. Burrows, C. Kaufman, B. Lampson, Authentication and Delegation with Smart-cards, *Theoretical Aspects of Computer Software*, 1991.
2. P. Biddle, M. Peinado and D. Flanagan, Privacy, Security and Content Protection, http://download.microsoft.com/download/a/f/c/afcf8195-0eda-4190-a46d-aa60b45e0740/Secure.ppt
3. D. Cook, R. Baratto and A. Keromytis, Remotely Keyed Cryptographics - Secure Remote Display Access Using (Mostly) Untrusted Hardware (Extended Version), Columbia Univeristy Computer Science Technical Report CUCS 050-04, 2004.
4. D. Cook, J. Ioannidis, A. Keromytis and J. Luck, CryptoGraphics: Secret Key Cryptography Using Graphics Cards, *RSA Conference, Cryptographer's Track (CT-RSA)*, LNCS 3376, Springer-Verlag, pages 334-350, 2005.
5. H. Gobioff, S. Smith, J. Tygar and B. Yee, Smart Cards in Hostile Environments, *2nd USENIX Workshop on Electronic Commerce*, 1996.
6. R. Iannella, Digital Rights Management (DRM) Architectures, *D-Lib Magazine*, http://www.dlib.org/dlib/june01/iannella/06iannella.html vol. 7 (6), June, 2001.
7. i-Bench version 1.5, Ziff-Davis, Inc, http://www.veritest.com/benchmarks/i-bench/.
8. D. Koller, M. Turitzin, M. Levoy, M. Tarini, G. Croccia and P. Cignoni and R. Scopigno, Protected Interactive 3D Graphics Via Remote Rendering, *ACM SIGGRAPH*, 2004.
9. E. Levy, Interface Illusions, *IEEE Security & Privacy*, vol. 2 (6), 2004, pages 66-69.
10. Microsoft Windows 9 Media Series Digital Rights Management, http://www.microsoft.com/windows/windowsmedia/drm.aspx, 2004.
11. J. Nieh, S. Jae Yang and N. Novik, Measuring Thin-Client Performance Using Slow-Motion Benchmarking, *ACM Transactions on Computer Systems*, vol. 21 (1), pages 87-115, 2003.
12. OpenGL Organization, http://www.opengl.org.
13. RSA Laboratories, PKCS #1: RSA Encryption Standard Version 1.5, November, 1993.
14. Trusted Computing Group, Trusted Computing Group Architecture Overview, https://www.trustedcomputinggroup.org/home, 2004.
15. T. J. Walsh and D. R. Kuhn, Challenges in Securing Voice over IP, *IEEE Security & Privacy Magazine*, vol. 3 (3), May/June 2005, pages 44-49.
16. M. Woo, J. Neider, T. Davis and D. Shreiner, *The OpenGL Programming Guide*, 3rd edition, Addison-Wesley, Reading, MA, 1999.