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

Debra L. Cook   Ricardo Baratto   Angelos D. Keromytis
Department of Computer Science, Columbia University
{*dcook,ricardo,angelos*}*@cs.columbia.edu*

Abstract

Software that covertly monitors a user's 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 by the owner of a system to spy on other users of the 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 conferencing 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 investigate the possibility of restricting the boundary of trust required to the client's GPU, and evaluate the possibility of moving decryption into GPUs. We discuss the applicability of GPU-based decryption in these two sample scenarios and identify the limitations of the current generation of GPUs. We propose straightforward modifications to future GPUs that will allow the realization of our architecture.

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 [10, 11, 34] or building a trusted system from the bottom-up, using a combination of hardware support [5, 6, 12, 42, 44, 20, 33, 38, 23, 8, 32] and operating system extensions [9]. While promising, these approaches offer only limited security against an adversary that legitimately controls the spyware-infected system, or against spyware that do 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. Specifically, sensitive content is directly passed to the GPU in encrypted form. The GPU decrypts and displays such content without ever storing the plaintext in the system's main 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 conferencing or remote desktop display access without trusting the rest of the system. Furthermore, our design allows a user to securely enter a password or PIN to a remote system without revealing it to any spyware and without requiring additional hardware. Finally we describe how, by using a suitably modified USB keyboard, it is possible to completely protect the user's communications with a remote server.

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, mostly due to the limitations of current GPU APIs, such as OpenGL. The most difficult aspect of moving decryption into a GPU is the API and the types of operations supported within the GPU. While it is possible to implement some symmetric key ciphers such as AES [3] in OpenGL, the performance is poor due to the number and types of operations required, as we demonstrated in [13]. Other ciphers cannot be implemented to run entirely within GPUs using current APIs. 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. We currently use RC4 [36] for display encryption. In the future, either a cipher suited for GPUs and/or an improved API for 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, as well as possible integration of our scheme with the Trusted Computing Group's proposed architecture.

Paper Organization The remainder of the paper is organized as follows. We give an overview of OpenGL in Section 2. We describe our motivation, architecture and prototype in Section 3. We discuss the limitations of GPU APIs and how these impacts our ability to remotely key and implement decryption in the GPU in Section 4. Section 5 presents a preliminary performance analysis.

2 OpenGL and GPU Background

We provide a brief overview of aspects of OpenGL and GPUs relevant to our experiments. A basic knowledge of the capabilities and limitations of GPUs is necessary to understand our proposed architecture and prototype. The two most common APIs for GPUs are OpenGL and Direct3D (part of the Microsoft DirectX API). We use OpenGL in order to provide platform independence (in contrast to Microsoft's Direct3D). See `http://www.opengl.org/` and [43] for a complete description of OpenGL. As explained later within the paper, operations which can be performed in the GPU are limited by the API. For the operations required of our prototype, the same limitations exist in both Direct3D and OpenGL. We choose to avoid higher level languages built on top of these APIs in order to ensure that specific OpenGL commands are being used. Examples of such languages include Cg [16] (HLSL in DirectX) and, from more recent research, Brook (the BrookGPU[4] compiler uses Cg in addition to OpenGL and Direct3D). Higher level languages do not allow the developer to specify which OpenGL commands are utilized when there are multiple ways of implementing a function via OpenGL commands and do not even guarantee the operations will be transformed into OpenGL commands but instead may transform it into C code. For example, code in a higher level language that XORs two bytes will likely be transformed into code executed in the operating system rather than converted into OpenGL commands that converts the bytes to pixels and XORs pixels.

Our prototype requires the display be set to 32-bit pixels[1]. A data format indicating such items as number of bits per pixel and the ordering of color components specifies how the GPU interprets and packs/unpacks the bits when reading data to and from system memory. The data format may indicate that the pixels are to be treated as floating point numbers, color indices, or stencil indices. When using the floating point representation and reading data from system memory, the data is unpacked and converted into floating point values in the range $[0, 1]$. Luminance, scaling and bias (all of which we do not use in the prototype) are applied per color component. The next step is to apply the color map, which we describe later in more detail. The values of

[1]When using 32 bit pixels, 1 byte is typically dedicated to each of the Red, Green, Blue and Alpha components. A format with 10 bits for each of the Red, Green and Blue components and 2 bits for the Alpha component may also be supported by the GPU.

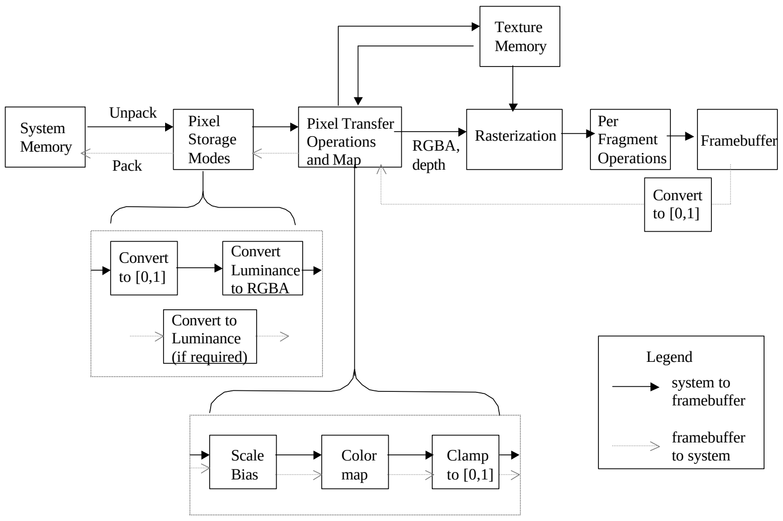

Figure 1. **OpenGL Pixel Processing Pipeline**

the color components are then clamped to be within the range $[0, 1]$. Figure 1 shows the components of the OpenGL pipeline that are relevant to pixel processing when pixels are treated as floating point values. While implementations are not required to adhere to the pipeline, it serves as a general guideline for how pixels are processed.

The OpenGL commands in our implementations consist of writing bytes from the system memory to the GPU as pixels with either a color map or the logical operation of XOR turned on. The logical operation of XOR produces a bitwise-XOR between the pixel being read in and the pixel currently in the destination, with the result being written to the destination. This is used to apply a keystream to the image. When decrypting an image, it is necessary to disable dithering to prevent pixels from being averaged with their neighbors when the image is read into the GPU. Color mapping is one of the slowest operations to perform [43]; however, we use it only for the decryption function of the asymmetric cipher used to send a secret key to the GPU. A color map is applied to a particular component of a pixel when the pixel is copied from one coordinate to another or when bytes are read in from system memory to be converted to pixels in the GPU. A color map can be enabled individually for each of the RGBA components. The color map is a static table of floating point numbers between 0 and 1. Internal to the GPU, the value of the pixel component being mapped is converted to an integer value which is used as the index into the table and the pixel component is replaced with the value from the table. For example, if the table consists of 256 entries, as in our implementation, and we apply the map to the red component of a pixel, we treat the 8 bits of the red value as an integer between 0 and 255, and update the red value with the corresponding entry from the table.

OpenGL requires support for at least a front buffer (image is visible) and a back buffer (image is not visible) but does not require support for the Alpha pixel component in the back buffer. This limits us to three bytes per pixel (the Red, Green, Blue components) when performing operations in the back buffer. It is worth mentioning that while a 32 bit pixel format is used, the 32 bits cannot be operated on as a single 32 bit value, but rather is interpreted in terms of pixel components. For example, it is not possible to add or multiply two 32 bit integers by representing them as pixels. In general, x bit pixels cannot be used to operate on x bit integers.

Due to limitations of current APIs, algorithms performing certain byte and bit-level operations are not suitable for GPUs. While simple logical operations can be performed efficiently in GPUs on large numbers of bytes, the byte and bit-level operations typically found in symmetric key ciphers, such as shifts and rotates, are not available via the APIs to GPUs. Modular arithmetic operations, which are required by AES, are also not available (we note that it is possible to implement AES as a series of copies with color maps and logical operations enabled [13]). While shifts and rotates can be performed on single bytes by defining color maps and using multiple copy commands, shifts across multiple bytes and table lookups based on specific bits, prove to be more difficult. For example, there is no straightforward way to implement in OpenGL the data dependent rotations found in RC6 [27] and MARS [14]. Also consider the DES S-Boxes [2]. The index into the S-Box is based on six key bits XORed with six data bits. Two of the bits are used to select the S-Box and the remaining four are the index into the S-Box. Masks of pixels copied onto the data can be used to "extract" the desired bits, but to merely XOR the six key bits with six data bits requires copying the pixel containing the desired key bits onto the pixel containing the mask with XOR turned on, doing the same for the data pixel, then copying the two resulting pixels to the same position. Color maps are required to emulate the S-Box. Overall, even when it may be technically possible to implement a symmetric key cipher in OpenGL, a larger number of less efficient operations are required than in a C implementation.

When using OpenGL for graphics programming, the more common aspects of vertex processing are utilized. Shapes are defined as sets of vertices, with colors or textures applied. In addition to defining the basic image, various parameters (such as the viewpoint, projection, lighting, fog and orientation) can be set. Rotations and translations can be applied to shapes to produce movement. It is these typical aspects of graphics processing that are of interest when creating a stream cipher suitable for a GPU.

3 Architecture

3.1 Motivating Applications

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 [24]. 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. For example, consider the system presented by Koller et al[21]. In this case, access to sensitive 3D data was controlled by manipulating the content sent to the remote display client. However, since the display data on the client could not be secured, a number of additional mechanisms had to be devised to prevent the actual client application from being used as an attack tool on the system. On the other hand, if the display is only in decrypted form within the GPU, we only need to block reads of the current display by other applications.

In video conferencing, we wish to prevent clients from copying the conference displays. How to secure audio is beyond the scope of this paper, although the concept we demonstrate with GPUs can also be applied to 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 [26] 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 [1]. 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 [7], 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 [41]. 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.

3.2 Architecture Overview

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. When the proxy consists of a smartcard and reader, the card reader is considered untrusted. Attacks requiring modified hardware are beyond the scope of this paper. Figure 2 depicts the 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 other processes or the operating system, 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 due to the GPU's API.

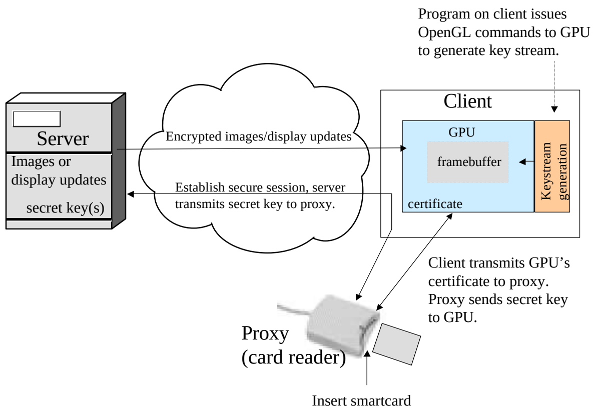
Program on client issues OpenGL commands to GPU to generate key stream.
Client
Server
Images or display updates
secret key(s)
Encrypted images/display updates
Establish secure session, server transmits secret key to proxy.
GPU
framebuffer
certificate
Keystream generation
Client transmits GPU's certificate to proxy. Proxy sends secret key to GPU.
Proxy (card reader)
Insert smartcard

Figure 2. **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 the key and key stream. 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. None of these operations require us to copy the image (plaintext) to the system's main memory.

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 certificate and private key. The certificate may be issued by the manufacturer and hardwired in the GPU. Another option is to allow writing the certificate to the GPU under circumstances when the client's OS is trusted, such as when the GPU is first being installed on a newly configured client. The first 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 certificate. 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 verification 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 3. The process of converting the key from being encrypted under 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, and that the links between these entities denote logical connections. 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. This scenario is not applicable to the real time applications we are interested in. 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. In all cases, 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 [22] and [17] 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 when sending it to the GPU. Obstacles arise due to the lack of support in GPU APIs for the operations required for public key ciphers, such as modular arithmetic for large integers, as mentioned in Section 2. Furthermore, the GPU's certificate must be placed in the GPU without exposing the private key to the operating system. We discuss the limitations of the GPU in regards to public key cryptography when describing our prototype.

3.3 Implementation

To determine the feasibility of our scheme, we implemented the general scenario with three entities: a server, a proxy and the client. We use a stream cipher, RC4, to encrypt the images (as opposed to a block cipher) 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 the GPU with an improved API as we discuss in Section 4. Specifically, computation of the keystream cannot be efficiently implemented entirely in OpenGL for a cipher such as RC4. In our description of the prototype, 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 i^{th} subset of data. These keys may be individually pre-determined, or computed through a master key using a pseudo-random function (PRF).
- A frame refers to one frame of video or one display update, depending on the application.
- 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 3 illustrates the steps for the remote keying and decryption of images in our prototype. The GPU has a certificate containing its RSA [31] key stored in its 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 to simulate having a certificate within the GPU. Entering a certificate into 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 certificate 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 certificate 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 certificate.

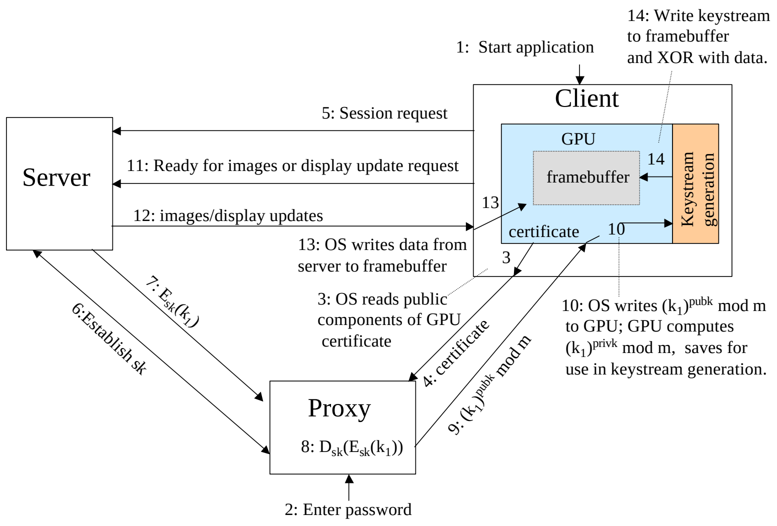
14: Write keystream to framebuffer and XOR with data.
1: Start application
Client
5: Session request
Server
GPU
14
framebuffer
Keystream generation
11: Ready for images or display update request
13
12: images/display updates
certificate
10
13: OS writes data from server to framebuffer
3
3: OS reads public components of GPU certificate
7: $E_{sk}(k_1)$
6:Establish sk
4: certificate
9: $(k_1)^{pubk}$ mod m
10: OS writes $(k_1)^{pubk}$ mod m to GPU; GPU computes $(k_1)^{privk}$ mod m, saves for use in keystream generation.
Proxy
8: $D_{sk}(E_{sk}(k_1))$
2: Enter password

Figure 3. **Remotely Keyed Decryption in GPU Protocol Shown: logical links (*i.e.*, the proxy communicates with the server through the client).**

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 certificate, and uses a single session key, sk. When contacting the proxy, the server sends a random nonce and its certificate 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 an MD5 [28] 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 specific 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 performed 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 GPUs 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.

3.4 Encryption at the Server and Client

Our prototype focuses on the securing of images sent to the untrusted client. We briefly mention here two aspects regarding encryption of images on the server and encryption of user input on the client. A complete system must include protecting any user input on the client which is sent to the server. Furthermore, in proposing to design a new stream cipher suitable for executing within GPUs, we must ensure that the cipher can also be efficiently implemented on the server. With respect to using a new cipher suited for GPUs, the server can write images to its own GPU for encryption before sending them to the client. In video conferencing applications, the images being encrypted likely appear on the monitor of the speaker and can be encrypted in the GPU before the server sends the frames to the clients of the conference's other participants. In thin-client applications, even though the server has no need to display updates to its GPU, it can encrypt the update by writing an image to its GPU and reading the result.

The user responses on the untrusted client pose a more interesting problem in that they require preventing input from the keyboard and mouse from being available to the untrusted OS. One potential solution is to encrypt the keyboard inputs inside the keyboard itself (*e.g.,* on the keyboard's USB controller). This can be done on a portable folding keyboard (as is available for several PDA devices) that connects to USB. The mouse may be directly connected to the keyboard (*e.g.,* a TrackPoint device, as is common with several laptops), or that input is taken only from the keyboard. A pin can be used as the key to the cipher used for encrypting the inputs. The pin can be of sufficient length to thwart a brute force attack. The server may either choose a pin for the user (displaying it securely to the user using our scheme) or have the user select a pin from a keypad displayed on the GPU.

If the server selects the pin, it merely sends it as an encrypted image to the client's GPU, where it is decrypted and presented to the user. The pin can be a relatively small, unpredictable area of the image. An attacker or malware attempting to modify the pin will at best have access to the encrypted image. The user can select a pin if the server displays a keypad to the user via the client's GPU. The user will select characters from the keypad by clicking on or entering a series of squares from the keypad, with the coordinates of the selections sent to the server. Even though the client's OS will see the coordinates of the user's selections (since keyboard and mouse inputs are not yet encrypted), it does not have access to the unencrypted keypad, making this information useless to an attacker. To avoid guessing attacks based on the relative locations of the mouse pointer, the keypad configuration is changed every time a digit is selected. If an attacker or malware on the client attempts to alter the coordinates sent to the server, the altered values may not correspond to valid positions on the keypad. Other possibilities include the use of graphical passwords [15, 40] and shoulder-surfing-resistant PIN-entry methods [30], which we intend to investigate in future work.

3.5 Proxy Attacks

Our scheme, as described thus far, is susceptible to a proxy attack: since the proxy, server, and client are assumed to communicate over an untrusted network (which includes the client's operating system), it is possible for an attacker to perform a man in the middle attack using another system (which has a GPU with a valid certificate) to perform the key exchange with the proxy device. The encrypted data stream can be displayed on the attacker's system (whose GPU has been given the session key by the proxy), and then transmitted to the user's system for displaying. This attack is feasible because the proxy cannot verify that the GPU it is communicating with resides on the same system that the user is using. However, the attacker cannot extract the encrypted image from their GPU's frame-buffer and thus cannot relay it to the target system, making the attack obvious to the end user. Another possibility we intend to investigate in the future is the use of packet leashes [18] in the context of the communication between the proxy and the GPU, although this would likely increase the cost of the GPU and the smartcard to unacceptable levels.

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.

As we mentioned in Section 2, GPUs are not designed to perform general arithmetic and byte-level operations. There are no API commands for common operations such as addition, multiplication, 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 (this can currently by done 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. For example, the exponents and modulus in RSA must each fit within 8 bits, making them entirely unsuitable for a security application. 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, similar to what was done for RC4, 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 fit in within a single byte, thus requiring the modulus and exponents also each fit within a single byte and resulting in key components too small to be secure, since an exhaustive search for the private key and data is easily performed. In order 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 exponent, public exponent and modulus. We used a series of 8-bit values to represent the data, in our case the secret key for RC4, encrypted with RSA. Each is encrypted with mini-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 that we intend to explore in future work is the integration of a decrypting GPU with a trusted platform module (TPM) such as the one proposed by the Trusted Computing Group. This chip could handle certificate 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. All the common stream ciphers, such as LILI [37], RC4, SEAL [25], SOBER [29], and SNOW [39], 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 specific 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], $a + b \bmod 256$ 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 i^{th} 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 b^{th} 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 in the operating system's memory.

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 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, as the computation model of modern GPUs does not efficiently support secure MAC constructs.

4.3 Data Compression

Traditionally, remote display and videoconferencing systems have made extensive use of data compression in order to maximize network utilization and allow use in bandwidth-limited environments. Since encrypted data cannot be compressed, a system that provides secure remote access must compress its data traffic before encrypting it. Clearly, this approach imposes a limitation in our architecture: In order to provide data compression, the client GPU must be able to uncompress data. Uncompressing the data within the client process would expose unencrypted display updates to the host operating system.

A straightforward solution would be to add hardware decompression abilities to the GPU. This could be accomplished by using widely available data decoding chips, such as MPEG hardware decoders; indeed, several DVD-ready GPUs contain such logic already. An alternative approach, in particular for thin-client scenarios, would be to tailor the display protocol and its compression to use operations available in the GPU. More recent thin-client systems have proposed remote display protocols that employ different types of commands and compression algorithms for different kinds of display updates [35]. The advantage of this approach derives from the characteristics of the protocol commands that provide inherent compression, negating the need for additional, specialized compression algorithms. For example, a command that instructs the client to fill a rectangular region with a particular color consumes very little bandwidth, while compressing a potentially large region of the screen. Execution of such a command is clearly within the operations available in existing GPUs. By appropriately designing the remote display protocol to utilize similar operations, we believe it is possible to improve our architecture to consume reasonable bandwidth without compromising security.

5 Experiments

To determine the practical feasibility of our architecture, we conducted a set 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-specific 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 consisted of C, OpenGL and GLUT, compiled using Visual C++ version 6.0. The processes for the server and proxy are written in JAVA, using version 1.4.2_03 with the JAVA Cryptography Extension.

Tests were performed using 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.

In our first experiment, we simulate streaming video applications, such as NetMeeting, by sending a stream of images from the server to the client. We tested with frames of sizes 320x240 and 500x500. The frames were encrypted and stored in individual files on the server prior to starting the application. A small number of unique frames were created and the server repeatedly cycled through the set. To measure thin-client performance, we used the distribution of update sizes from a thin-client system running the standard i-Bench [19] web benchmark to determine the range of and average update sizes of a typical thin-client usage scenario. The update sizes ranged from 1x1 areas to 1,007x622 areas (626,354 pixels), with an average update size of 2,112 pixels. Tests were performed using the average update size. All tests used images encoded as 24-bit RGB pixels, with 8-bits per color component.

Image Size in Pixels	Encrypted (1.8Ghz)	Unencrypted (1.8Ghz)	Encrypted (1.3Ghz)	Unencrypted (1.3Ghz)	Encrypted (800MHz)	Unencrypted (800MHz)
16x132	76.92fps	76.92fps	83.33fps	83.33 fps	100fps	100fps
320x240	17.27fps	23.87fps	21.41fps	35.71fps	11.11fps	14.93fps
500x500	6.90fps	11.24fps	9.30fps	14.86fps	6.54fps	10.39fps

Table 1. **Rates for 1.8Ghz and 800MHz PCs, and 1.3Ghz Laptop**

Image Size in Pixels	Encrypted Frames	Unencrypted Frames
16x132	76.34fps	76.34fps
320x240	8.69fps	9.95fps
500x500	3.13fps	3.66fps

Table 2. **Rates for 1.8Ghz PC on Dedicated LAN**

Image Size in Pixels	Encrypted Frames	Unencrypted Frames
16x132	76.50fps	76.50fps
320x240	8.63fps	10.20fps
500x500	2.09fps	2.35fps

Table 3. **Rates for 1.8Ghz PC on Shared LAN**

For each image size, two types of tests were run. The first 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, and GPU) were run on the same PC or laptop, and each of the three clients was tested. The results of the first set of tests are shown in Table 1.

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. Tables 2 and 3 contain 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 for both encrypted and unencrypted frames are provided. In video conferencing applications, the number of frames supported per second 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 frames per second reflects the maximum burst rate supported.

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. When testing streaming images over the LAN, it was necessary for the client to signal the server when it was ready for the next frame to avoid synchronization problems.

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. 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, since the keystream was written at once to the GPU, requiring only one write.

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 2,112-pixel 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 server and client were running on the same system to eliminate network delays and bandwidth restrictions. 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 since such updates are typically infrequent and, from a human factors perspective, are no worse than 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 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 and XORed with the current encrypted frame. The execution per frame is less then 1*ms*, indicating that any differences in the time to process encrypted frames versus the time to process 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 added approximately two seconds to the processing in each environment.

6 Conclusions

We address 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 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.

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 and certain video conferencing applications. Due to the need to generate the keystream in the client, the overhead is almost entirely due to writing the keystream to the GPU as it is computed. Designing a keystream which takes advantage of typical graphics operations to move keystream generation entirely inside the GPU will eliminate this overhead. 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 bottleneck for the performance of the system. Future work includes creating a stream cipher that runs entirely within a GPU and takes advantage of graphics operations, and developing prototypes that fully integrate the concept into thin-client applications.

References

[1] Windows 9 Media Series Digital Rights Management. `http://www.microsoft.com/windows/windowsmedia/drm.aspx`.

[2] FIPS 46-3 Data Encryption Standard (DES), 1999.

[3] FIPS 197 Advanced Encryption Standard (AES), 2001.

[4] BrookGPU. http://graphics.stanford.edu/projects/brookgpu/index.html, 2003.

[5] W. A. Arbaugh. *Chaining Layered Integrity Checks*. PhD thesis, University of Pennsylvania, Philadelphia, 1999.

[6] W. A. Arbaugh, D. J. Farber, and J. M. Smith. A secure and reliable bootstrap architecture. In *IEEE Security and Privacy Conference*, pages 65–71, May 1997.

[7] P. Biddle, M. Peinado, and D. Flanagan. Privacy, Security and Content Protection. http://download.microsoft.com/download/a/f/c/afcf8195-0eda-4190-a46d-aa%60b45e0740/Secure.ppt.

[8] E. Brickell, J. Camenisch, and L. Chen. Direct Anonymous Attestation. In *Proceedings of the* 11^{th} *ACM Conference on Computer and Communications Security (CCS)*, pages 132–145, October 2004.

[9] A. Carroll, M. Juarez, J. Polk, and T. Leininger. Microsoft Palladium: A Business Overview. White paper, Microsoft, August 2002.

[10] N. Chou, R. Ledesma, Y. Teraguchi, and J. C. Mitchell. Client-Side Defense Against Web-Based Identity Theft. In *Proceedings of the ISOC Symposium on Network and Distributed Systems Security (SNDSS)*, February 2004.

[11] M. Christodorescu and S. Jha. Testing Malware Detectors. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, July 2004.

[12] P. C. Clark. *BITS: A Smartcard Protected Operating System*. PhD thesis, George Washington University, 1994.

[13] D. Cook, J. Ioannidis, A. Keromytis, and J. Luck. CryptoGraphics: Secret Key Cryptography Using Graphics Cards. In *RSA Conference, Cryptographer's Track (CT-RSA), to appear*, February 2005.

[14] D. Coppersmith, et.al. The MARS Cipher. http://www.research.ibm.com/security/mars.html, 1999.

[15] D. Davis, F. Monrose, and M. K. Reiter. On User Choice in Graphical Password Schemes. In *Proceedings of the* 13^{th} *USENIX Security Symposium*, pages 151–163, August 2004.

[16] R. Fernando and M. Kilgard. *The Cg Tutorial*. Addison-Wesley, New York, 2003.

[17] H. Gobioff and S. Smith and J. Tygar and B. Yee. Smart Cards in Hostile Environments. In *2nd USENIX Workshop on Electronic Commerce*, 1996.

[18] Y.-C. Hu, A. Perrig, and D. B. Johnson. Packet Leashes: A Defense against Wormhole Attacks in Wireless Networks. In *Proceedings of IEEE Infocomm*, April 2003.

[19] i-Bench version 1.5, Ziff-Davis, Inc. http://www.veritest.com/benchmarks/i-bench/.

[20] N. L. P. Jr., T. Fraser, J. Molina, and W. A. Arbaugh. Copilot - a Coprocessor-based Kernel Runtime Integrity Monitor. In *Proceedings of the* 13^{th} *USENIX Security Symposium*, pages 179–194, August 2004.

[21] D. Koller, M. Turitzin, M. Levoy, M. Tarini, G. Croccia, P. Cignoni, and R. Scopigno. Protected Interactive 3D Graphics Via Remote Rendering. In *Proceedings of ACM SIGGRAPH*, 2004.

[22] M. Abadi and M. Burrows and C. Kaufman and B. Lampson. Authentication and Delegation with Smart-cards. In *Theoretical Aspects of Computer Software*, 1991.

[23] J. P. McGregor and R. B. Lee. Protecting Cryptographic Keys and Computations via Virtual Secure Coprocessing. In *Proceedings of the Workshop on Architectural Support for Security and Anti-virus (WASSA)*, pages 11–21, October 2004.

[24] J. Nieh, S. J. Yang, and N. Novik. Measuring Thin-Client Performance Using Slow-Motion Benchmarking. *ACM Transactions on Computer Systems (TOCS)*, 21(1):87–115, Feb. 2003.

[25] P. Rogaway. A Software Optimized Encryption Algorithm. In *Journal of Cryptology*, pages 273–287, 1998.

[26] R. Iannella. Digital Rights Management (DRM) Architectures. *D-Lib Magazine*, 7(6), June 2001.

[27] Rivest, Robshaw, Sidney, and Yin. RC6 Block Cipher. http://www.rsa.security.com/rsalabs/rc6, 1998.

[28] R. Rivest. The MD5 Message-Digest Algorithm. Internet RFC 1321, April 1992.

[29] G. Rose. A Stream Cipher Based on Linear Feedback Over GF (28). In *Information Security and Privacy, LNCS 1438*, page 135ff, 1998.

[30] V. Roth, K. Richter, and R. Freidinger. A PIN-Entry Method Resilient Against Shoulder Surfi ng. In *Proceedings of the* 11^{th} *ACM Conference on Computer and Communications Security (CCS)*, pages 236–245, October 2004.

[31] RSA Laboratories. *PKCS #1: RSA Encryption Standard*, version 1.5 edition, 1993. November.

[32] R. Sailer, T. Jaaeger, X. Zhang, and L. van Doorn. Attestation-based Policy Enforcement for Remote Access. In *Proceedings of the* 11^{th} *ACM Conference on Computer and Communications Security (CCS)*, pages 308–317, October 2004.

[33] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. Design and Implementation of a TCG-based Integrity Measurement Architecture. In *Proceedings of the* 13^{th} *USENIX Security Symposium*, pages 223–238, August 2004.

[34] S. Saroiu, S. D. Gribble, and H. M. Levy. Measurement and Analysis of Spyware in a University Environment. In *Proceedings of the ACM/USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, March 2004.

[35] B. K. Schmidt, M. S. Lam, and J. D. Northcutt. The Interactive Performance of SLIM: A Stateless, Thin-Client Architecture. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP)*, pages 32–47, Kiawah Island Resort, SC, December 1999.

[36] B. Schneier. *Applied Cryptography, 2nd edition.* John Wiley and Sons, New York, 1996.

[37] Simpson, Dawson, Golic, and Millar. LILI Keystream Generator. In *Selected Areas in Cryptology, LNCS 2012*, page 248ff, 2000.

[38] S. Smith. Magic Boxes and Boots: Security in Hardware. *IEEE Computer*, 37(10):106–109, October 2004.

[39] SNOW. `http://www.it.lth.se/cryptology/snow`.

[40] J. Thorpe and P. C. van Oorschot. Graphical Dictionaries and the Memorable Space of Graphical Passwords. In *Proceedings of the* 13^{th} *USENIX Security Symposium*, pages 135–150, August 2004.

[41] Trusted Computing Group. Trusted Computing Group Architecture Overview. `https://www.trustedcomputinggroup.org/home`, 2004.

[42] J. Tygar and B. Yee. DYAD: A System for Using Physically Secure Coprocessors. Technical Report CMU–CS–91–140R, Carnegie Mellon University, May 1991.

[43] M. Woo, J. Neider, T. Davis, and D. Shreiner. *The OpenGL Programming Guide, 3rd edition*. Addison-Wesley, Reading, MA, 1999.

[44] B. Yee. *Using Secure Coprocessors*. PhD thesis, Carnegie Mellon University, 1994.