Cobalt: Separating content distribution from authorization in distributed file systems

Kaushik Veeraraghavan, Andrew Myrick, and Jason Flinn
Department of Electrical Engineering and Computer Science
University of Michigan

Abstract

How should a distributed file system manage access to protected content? On one hand, distributed storage should make data access pervasive: authorized users should be able to access their data from any location. On the other hand, content protection is designed to *restrict* access — this is often accomplished by limiting the set of computers from which content can be accessed. In this paper, we propose a new method for storing content in distributed storage called Cobalt. Rather than grant access to data based on the computer that reads the data, Cobalt grants access based on the physical proximity of authorized users. Protected content is stored encrypted in the distributed Blue File System; files can only be decrypted through the cooperation of a personal, mobile device such as cell phone. The Cobalt device is verified by content providers: it acts as a proxy that protects their interests by only decrypting data when policies specified during content acquisition are satisfied. Wireless communication with the device is used to determine the physical proximity of its user; when the Cobalt device moves out of range, protected content is made inaccessible. Our results show that Cobalt adds only modest overhead to content acquisition and playback, yet it enables new forms of interaction such as the ability to access protected content on ad hoc media players and create playlists that adapt to the tastes of nearby users.

1 Introduction

The complexity of managing digital content continues to increase. Many people use a wide variety of computing and consumer electronics devices to access their content — PCs, laptops, MP3 players, and DVRs are just a few examples. While users typically access their content on a handful of well-known devices, inevitably some scenarios arise in which they would like to access their content on *ad hoc devices*, which we define to be devices that they do not own and do not commonly use. For instance, a visiting family member might wish to display photos using a living room DVR, or a party-goer might wish to share their taste in music by playing MP3 files on a friend's stereo.

Ad hoc access to digital content is challenging for several reasons. First, the ad hoc media player must locate the content that a user wishes to access. Currently, this requires the user to copy their content to portable storage or specify a location in a distributed storage system from which the content can be read. Second, if the content is not compartmentalized into a specific subtree of the portable or distributed storage, the media player must search through many files to locate relevant media. Over a wide-area link, such searches can be extremely time-consuming. Third, configuring a media player to locate the content might be challenging since each new device presents a different user interface. Finally, users may have to enter a password to grant the media player access to their content — however, they have no assurance that any entered password will not be abused.

Digital rights management introduces another dimension of complexity. Content providers who wish to ensure that users will not share their products in an unauthorized manner typically use some form of digital rights management. While many providers such as Yahoo [30] and the iTunes Music Store [1] allow a user to view content on multiple media players, the user must explicitly authorize the device on which content is viewed. In order to play protected content, the user must enter his userid and password. To revoke access to his content, the user must later deauthorize the ad hoc device. Potentially, such authorizations compromise privacy by informing the content provider of the movements and activities of users who have purchased their content.

In this paper, we introduce Cobalt, a mobile solution for ad hoc content access. Cobalt is implemented as an extension to the distributed Blue File System (BlueFS) [22]. Cobalt runs on a *token*, which is defined to be a mobile device such as a cell phone or PDA that is nearly always carried by its user. Cobalt improves usability and security by automating:

- **content location.** A Cobalt token automatically discovers media players in its immediate vicinity. When its user decides to access her content on an ad hoc player, she specifies her intentions as a semantic query. The token translates the semantic specification into a specific list of files to share. For example, a user may share all her MP3 files with a media player or just her recent vacation photos. The token provides the media player with the network address of the BlueFS server from which these files can be fetched, as well as a list of shared content.
- **authorization.** Cobalt uses techniques from Zero-Interaction Authentication [5] to limit the content that is shared with each media player. Content is protected with per-file keys that are encrypted with a key-encrypting key (KEK) known only to the token. Thus, the media player must contact the token to obtain the per-file key for all shared content. Unless the user has explicitly authorized sharing of the content with the media player, the Cobalt token denies access by refusing to decrypt the per-file key. Cobalt leverages Trusted Platform Module (TPM) hardware to ensure that decrypted per-file keys are not leaked by any media player to which they are provided. Currently, Cobalt limits ad hoc devices to read-only access to shared content.
- **digital rights management.** The token acts as a proxy for digital rights management that protects the interests of a content provider. During content acquisition, a provider uses the TPM to verify the integrity of the software running on the token. It then sends the token the key used to encrypt content along with a policy describing how that content can be played. The token encrypts the content key and a secure hash of the policy with its KEK. It stores these encrypted values along with the policy and encrypted content in BlueFS. Subsequently, it only decrypts the content key for media players that satisfy the policy. The use of a proxy improves usability because it eliminates the need to register and deregister media players with a content provider.

Our experimental results show that Cobalt adds only minimal overhead to content acquisition and playback. Further, this overhead does not substantially depend on the size of data being acquired or played. We also present results from a case study that shows how Cobalt can be used to enable new applications such as adaptive playlists that adjust the selection of music being played to match the tastes of people located nearby.

2 Design goals

In this section, we outline the goals that we followed in the design of Cobalt.

2.1 Usability

The primary design goal for Cobalt is usability. Cobalt minimizes the amount of effort required to access content. When user input is required, Cobalt strives to provide the interface on the token, with which the user is familiar, rather than on an ad hoc media player, which may have an unfamiliar interface.

Consider the effort that a typical user must currently exert to access protected content on an ad hoc media player. The user must authorize the new device with each content provider. Then, the user must provide the content to the device either with portable storage or by specifying a network address of a Web or distributed storage server from which data can be read. If the content is copied over the network and is not publicly accessible, the user must specify a password with which the media player can access the content. The user may need to manually create a playlist in order to specify what content should be shared with the ad hoc media player. Many of these interactions must be done using the unfamiliar media player interface.

For example, while iTunes allows content to be streamed to ad hoc computers, protected content can only be accessed after the userid and password of the person who owns the streaming computer is provided. Streaming is only permitted between computers on the same local network. While proxies exist that allow unpermitted access, setting up such proxies requires substantial configuration.

In contrast, Cobalt minimizes the actions required to share content with an ad hoc media player. It leverages a distributed file system, BlueFS, to share content seamlessly with the media player — this automatically provides prefetching and caching of content to improve the quality of playback. Since each token is associated with a specific BlueFS server, the user need not enter network addresses and other technical information. A Cobalt token allows its user to specify content to be shared as a semantic query rather than an explicit list of files; query results are automatically updated when a user adds content or modifies existing files. Finally, the token protects a user's content without the need to enter a password by decrypting only files that match the specified query.

2.2 Protection for content providers

The second design goal for Cobalt is to protect the interests of content providers. Content should not be leaked to unauthorized users or devices. Files should only be accessible on media players that satisfy the policy specified when content was acquired.

Figure 1 shows the Cobalt trust model. The token and ad hoc media player have Trusted Platform Modules. The

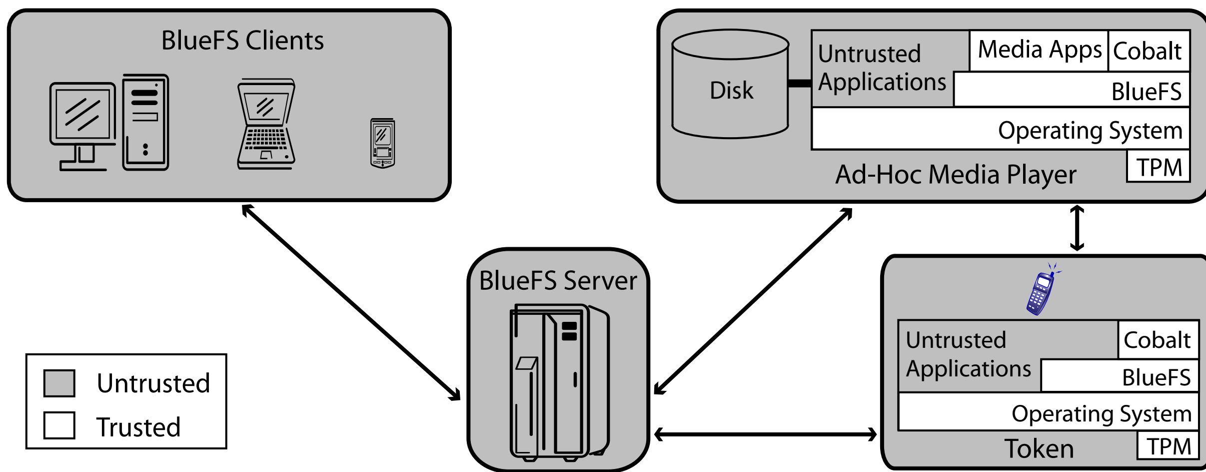


Figure 1. Cobalt trust model

TPM on each platform is used to verify the operating system, BlueFS client, and applications involved in content acquisition and playback. The TPM allows a content provider to verify the software running on the token — this verification is vital because the token acts as a proxy that protects the interests of the provider.

Using the media player's TPM, the token verifies the integrity of software on the player before it allows content to be decrypted. BlueFS, with the cooperation of the operating system, ensures that the decrypted content is only provided to the trusted application and not to other applications running on the media player. The operating system caches decrypted content in the buffer cache to improve performance. The BlueFS daemon may also cache content on disk; however, protected content on disk is always encrypted.

In contrast, the BlueFS server and its other clients are not trusted devices and do not need TPM hardware. The token has a symmetric key-encrypting-key (KEK) that is never exposed externally. Protected content is stored encrypted in BlueFS and cannot be decrypted without the KEK that is known only to the Cobalt token. This architecture allows unverified clients to prefetch and cache content on their local storage to improve performance. The Cobalt token only decrypts the content key for prefetched files after it has verified the trustworthiness of the media player.

Cobalt also allows users to protect their own content. For instance, a user might store all of her photos in BlueFS and protect them using Cobalt. When she visits a friend's house, she can then make specific subsets of those photos, e.g., vacation pictures, available on her friend's television. Photos that she does not choose to share will not be decrypted by her token.

A substantial advantage of this trust model is that the platforms required to have a TPM are typically closed. For example, the token may be a cell phone, and the media player may be a DVR or car stereo. In contrast to a general-purpose computer, these closed-platform devices typically run a much smaller set of software that may be more tightly controlled by their manufacturers. The task of verifying these platforms becomes easier given that the set of possible software and hardware combinations is limited. In contrast, the number of software and hardware combinations on a general-purpose computer is much larger, meaning that it might be more difficult to verify the integrity of other entities such as the BlueFS server.

2.3 Privacy

A Cobalt token preserves its user's privacy. The use of a mobile wireless device naturally raises concerns because the presence of such a device can be used to track the movements and activities of its user.

A Cobalt token discovers new media players in its vicinity without exposing its identity. The discovery protocol reveals to a media player that a token is located nearby; however, the discovery request is generic and reveals no information that can be used to identify a particular token. The user explicitly authorizes an interaction with a particular media player by selecting it from a menu on the token. Alternatively, the user may authorize future interactions with a media player by adding it to a list of pre-approved players. The token does not reveal its identity to unauthorized media players.

In contrast, if a user must register and deregister media players, the content provider has considerable information about the movement and activities of that person. In addition, if content is protected, then the user must disclose a password to the media player before accessing

content. A malicious media player could use this information to access unauthorized information.

Cobalt currently has a privacy limitation. In order to let media players efficiently generate playlists from shared content, the type-specific metadata of shared content, e.g., ID3 tags, are public and unencrypted. We plan to address this limitation in the future by including the metadata in the list of files generated by the token and specified to ad hoc media players. Once such metadata are available through other means, they can be encrypted in content files without adversely affecting performance.

3 Threat model

Cobalt is designed to protect the interests of both its users and content providers. For individual users, Cobalt prevents unauthorized access to their data. An attacker may try to subvert Cobalt to obtain access to content that the user has not decided to share. For content providers, Cobalt prevents access to content in violation of the specified policy. An attacker may try to subvert Cobalt to gain access to content in a manner that violates the policy or attempt to make a copy of decrypted content to gain unfettered future access.

We assume that attackers are capable of monitoring all communication between parties such as the Cobalt token, content provider, media players, and BlueFS server. Cobalt uses the station-to-station protocol [6] to establish a session key to encrypt communication; we assume that the private keys used in this protocol are known only to the respective parties and that trusted mechanisms exist for obtaining the public keys of other parties. We further assume that the encryption used by Cobalt is strong enough to provide confidentiality and authentication.

We assume that an attacker cannot compromise the hardware on the Cobalt token or media players. An attacker who subverts these mechanisms can gain unfettered access to content. Similarly, we assume that an attacker cannot subvert software that has been certified as trusted. A software exploit in a token or media player could record content keys to give the attacker access to protected content. Keys might also be discovered through covert channels such as power analysis. Finally, the current TPM standard is known to be vulnerable if attackers can modify software after it has been loaded and verified by the TPM but before it is used to access content.

We assume that an attacker may gain access to data stored on disk. Content and keys written to persistent storage are always encrypted. On the token, we assume that the TPM's sealed storage mechanisms prevent an attacker from obtaining the encryption keys. An attacker who compromises the BlueFS file server does not gain access to content since all content is encrypted. However, the attacker may mount a denial-of-service attack by deleting content or causing the server to respond to queries with incomplete information. Techniques that have been developed to deal with untrusted file servers [14] might address the latter problem.

An attacker might attempt to gain unauthorized access to content through a wormhole attack [11] in which a computer near a media player forwards packets to a remote Cobalt token over the Internet and returns the token's replies to the media player. Cobalt defends against wormhole attacks with a protocol that requires tokens to periodically respond to challenge messages within a threshold time period. If a number of challenges are missed, a media player refuses to play content. We assume that a threshold value exists that allows nearby tokens to respond in time and that is also small enough to prevent responses to be received from remote tokens.

An attacker may gain the ability to play content by obtaining possession of a token. This does not give the attacker more privilege than the Cobalt user, so an attacker possessing a stolen token cannot make unauthorized copies of content. If a token is lost or stolen, a user may deauthorize the token by re-keying content stored in BlueFS. However, any content previously fetched and cached by an attacker would still be viewable. Some cell phones have locking mechanisms that require a user to enter a PIN or password before using the device — such mechanisms, while not required by Cobalt, could potentially reduce the damage caused by a lost token.

4 Background

Cobalt leverages prior work in distributed file systems and trusted computing. In order to put Cobalt in its proper context, we briefly describe the relevant details of this prior research.

4.1 Blue File System

BlueFS is a server-based distributed file system that is designed to meet the storage needs of small groups of individuals such as a family [23]. The BlueFS file server, which is assumed to have a static IP address, might reside in the family's home or with its ISP. BlueFS clients include traditional computers such as desktops and laptops, as well as consumer electronics appliances such as MP3 players, cell phones, DVRs, and digital cameras. BlueFS supports disconnected operation [12] for mobile clients. When clients are connected to the server, the BlueFS consistency model is similar to Coda's weakly connected mode [21]. Prior to this work, BlueFS clients were tightly bound to a single server; they could only read or write data stored by that server. In Section 5.4.2,

we describe how we have extended BlueFS to support read-only sharing of data stored on different servers.

Cobalt exploits a novel feature of BlueFS called *persistent queries*. As described previously [23], a persistent query notifies standalone applications about modifications to data stored in the distributed file system. An application running on any client that is interested in receiving such notifications specifies a semantic query (e.g., all files that end in ".mp3") and the set of events in which it is interested (e.g., file existence and new file creation). The query is created as a new object within the file system. The BlueFS server evaluates the query and adds log records for matching events. For instance, in the above example, the server would initially add a log record to the query for every MP3 file accessible to the user who created the query and then incrementally add a new record every time a new MP3 file is created. Since the query and its results are a file system object, the underlying cache consistency mechanisms of BlueFS notify the application about changes to the query. Prior results have shown that persistent queries are fast to create since they are evaluated at the server, which keeps a metadata database to speed processing.

4.2 Trusted computing

Rather than propose a new model for trusted computing, Cobalt leverages previous work in this area [9, 15, 17]. It assumes that both the token and the ad hoc media player have a Trusted Platform Module, as defined by the Trusted Computing Group (TCG) [28]. In this section, we summarize only the portions of TPM that are relevant to our work: McCune et al. [17] provide a good introduction to TPM for those who wish more details.

A TPM implementation includes hardware support for cryptography primitives. At a minimum, Cobalt assumes that the TPM module incorporates two unique keys: the Attestation Identity Key (AIK) which is an RSA signing key pair and a symmetric Key-Encryption-Key (KEK). Any value signed with the AIK can be verified by an external entity using the public RSA key. The public RSA key may be exported as a certificate with an embedded chain that can be followed back to the manufacturer of the hardware. Thus, signing a value with the AIK allows any external entity to verify the identity and manufacturer of the device that generated the signature. To avoid computationally-expensive public key cryptography, the symmetric KEK is used to encrypt large data items.

Cobalt relies on TPM support for attestation. When requested, the TPM generates a *manifest* that explicitly lists the software loaded on the system, as well as a *Platform Configuration Registers (PCR) quote*, which is an AIK-signed hash of the manifest that can be used for verification. Upon request, a device provides its TPM-generated manifest and PCR quote to an external entity. The external entity verifies the manifest using the PCR quote and confirms that the software running on that device meets its approval. Additionally, if the entity doubts the integrity of the TPM on the device, it can verify the AIK signature on the PCR quote and trace the certificate chain to ensure that the device was manufactured by an entity that it trusts.

In Cobalt, content providers use the TPM to verify the integrity of a token. The token, in turn, uses the TPM to verify that media players meet the approval of policies specified by the provider during content acquisition.

The Cobalt token uses the KEK to encrypt the content key and hash of the policy that are given to it during content acquisition. Since the KEK never leaves the token, content keys cannot be subsequently decrypted without the cooperation of the token. We use the KEK to encrypt these data items because the performance of symmetric key encryption is substantially better than that of public key encryption. Our experimental results using a cell phone as the token confirm that public key operations can require several seconds to complete, whereas symmetric key operations require only a few milliseconds.

5 Implementation

5.1 Overall model

The Cobalt token is a small, mobile device. The token should be powerful enough to run a BlueFS client, yet small enough to always be carried by its user. Additionally, there should be a strong association between a token and its user so that the presence of the token can be taken to mean that its user is present. Cell phones are ideal Cobalt tokens because they meet all of the above criteria. Consequently, we use a Motorola E680i phone [20] for our token implementation in this paper. We have also ported the token to other platforms, such as the HP iPAQ PDA.

We assume that ad hoc media players run a BlueFS client. Given that platforms such as the TiVo DVR run the Linux operating system and have APIs that allow them to be extended with novel applications, we do not think this requirement will be onerous in the future. Our prior work [23] has also investigated how general-purpose computers can extend the functionality of consumer electronics appliances when those platforms are too closed to run a BlueFS client. Similar techniques could be applied in Cobalt if necessary (although the TPM verification would need to be extended to include the general-purpose computer).

Cobalt functionality can logically be separated into two phases: content acquisition, which is described in the next section, and content playback, which is discussed in Section 5.3. During content acquisition, the Cobalt token coordinates the acquisition of new protected content from a provider. The content, encrypted with a per-file content key, is stored in BlueFS. The content key, in turn, is encrypted with the token's KEK and also stored in BlueFS. Optionally, the content provider may supply a playback policy that is stored in BlueFS.

During playback, a token verifies that a file requested by an ad hoc media player has been authorized for access by its user. It checks that the media player meets the specifications provided by the content provider in the playback policy. It only decrypts the content key (allowing the ad hoc media player to decrypt the content) if both checks pass.

5.2 Acquiring content

The Cobalt token manages the acquisition of content. We have implemented content acquisition as a library routine that takes as parameters the network address of the content provider and a unique identifier for the specific content being acquired. Currently, we use a menu-driven user interface. However, our library design would make it trivial to substitute a more user-friendly GUI for directing content acquisition.

After the user selects the provider and content to acquire, the token opens a network socket to the specified provider. The token and provider mutually confirm each other's identities using the station-to-station protocol. Specifically, Cobalt uses the Full-STS variant that includes an exchange of public-key certificates. Each party provides the other with a certificate chain that vouches for their respective public keys. While Cobalt does not assume that the token and content provider have prior knowledge of each other, it does assume that there is at least one certificate authority trusted by each party that can vouch for the other. As a by-product of the station-to-station protocol, the token and the provider establish a symmetric session key that is used to encrypt further communication during content acquisition.

The provider verifies that the token is running software that it trusts. This is necessary since the provider will give the token sufficient information to decrypt the content. The provider therefore needs to verify that the token will only release the content in accordance with the specific policy for that content. For example, the content provider needs to assure itself that the token will not leak the content to an unauthorized third party.

The token uses its TPM's AIK as its public key during the station-to-station protocol. This allows the content provider to ascertain that the token has a valid TPM. The token then uses its TPM to generate a manifest and PCR quote. The content provider verifies the signature on the PCR quote and makes sure that the quote is a valid hash of the software manifest. The provider then verifies that the manifest entries are known versions of programs that can be trusted not to leak content.

The provider next encrypts the requested content with a symmetric content key. To improve performance, save battery lifetime, or deal with closed software environments in which the BlueFS client code cannot be run, the token typically enlists the cooperation of another *helper* computer during content acquisition. The helper is a BlueFS client that is known to the token; for example, it might be the user's workstation or BlueFS server. The helper need not be trusted by the content provider since it is never provided with the key necessary to decrypt the content. When a helper is used during content acquisition, the provider sends the encrypted content to the helper, which stores the data as a new file in BlueFS. If a helper is unavailable, the provider sends the content to the token. The token writes the data to BlueFS itself.

The provider then sends the content key and the policy for playing the content to the token. The policy for playback can be thought of as a set of requirements that the provider wishes to enforce on any device that tries to access the content. For example, the provider might request that the device be deployed on a TPM platform whose PCR quote corresponds to one of several known and trusted software configurations — this helps ensure that the content is not played on a device that leaks the content in an unauthorized manner. Alternatively, the policy might allow the provider, the manufacturer, or other trusted entities to vouch for software running on media players by digitally signing a certificate which lists the software. The media player could present a copy of this certificate to the token, which would verify that the certificate is signed by an entity trusted by the policy.

The token next generates a *protector* for the content being acquired. The protector is a concatenation of the content key and a SHA-1 hash of the policy specified by the provider. The token encrypts these values using its KEK and stores them in the BlueFS metadata of the newly-created content file. Cobalt uses AES Output-Feedback Chaining [7] when generating the protector—this ensures that the content key cannot be decrypted successfully if the policy hash is tampered with by an external entity. The token writes the policy, which may be of arbitrary length, to a separate file in BlueFS. The token deletes the content key from its memory after generating the protector. Although all our current tokens, the Motorola E680i cell phone and HP iPAQ, are BlueFS clients, Cobalt supports closed-platform tokens that cannot run

the client code by allowing the helper to store the protector and policy on their behalf.

If a Cobalt user is protecting his own content, the above process is simplified. The user specifies files to protect using his token — these files can be in a private directory within BlueFS. The token generates a content key for each file, encrypts the data with that key, and stores the encrypted data in a new file in a publicly-accessible BlueFS directory. The protector is generated and stored as described above.

5.3 Playing content

The Cobalt token runs a wireless discovery protocol to learn about media players in its immediate vicinity. This protocol can be run periodically or on-demand when the user starts a media sharing application on the token. Periodic discovery reduces user-perceived latency, but on-demand discovery saves battery energy on the token by only performing discovery when necessary. During discovery, the token sends a broadcast message over the local network (currently, we use 802.11b in ad hoc mode for discovery). All media players on the local area network that are willing to access content respond to the token. If the media player is a TPM platform, its response includes its public key (AIK), the manifest of software running on the media player, and the signed PCR quote necessary to verify the manifest.

The token presents the user with a list of all media players that responded. When the user selects one of these players, the token and the media player mutually authenticate and establish a secure communication channel via the station-to-station protocol. Cobalt allows media players to reject connections from unknown tokens if they wish; however, our design does require the media player to disclose its identity to such tokens.

The user next specifies which content he is willing to share with the media player. This content is stored as encrypted, publicly accessible files in BlueFS. Rather than requiring the user to specify a lengthy list of files, the Cobalt token allows its user to semantically specify which content should be shared as a persistent query. For example, he can create a persistent query that matches all MP3 files or refine the query to specify only music files from a certain artist. To share content, the user may browse through current outstanding persistent queries and select one that is most appropriate. Alternatively, the user may create a new query that matches the content that they currently want to share. In either case, the user specifies the query using the interface of his token, i.e., one that he knows and is comfortable with, rather than an unfamiliar interface presented by an ad hoc media player After authenticating the media player and establishing a secure session, the token sends the media player the IP address of the user's BlueFS server and the unique 96-bit identifier of the persistent query that specifies the content to be shared.

The media player next contacts the user's BlueFS server and fetches the persistent query object. We have implemented a federation mechanism, described in the next section, that allows BlueFS clients to mount third-party servers as read-only directories within their distributed namespaces. The persistent query contains the unique identifier of all files that match the associated semantic string. Rather than search the entire BlueFS namespace for relevant files, the media player can read the query results to determine the exact set of files that are being shared. Then, the media player can read the metadata associated with these files and add them to its list of available content. The media player may prefetch and store locally some of this content to improve performance. However, until the token provides the content key for a file, the media player cannot decrypt cached content.

When a Cobalt-protected file stored in BlueFS is accessed, the file metadata indicates that the content is encrypted. On the first access to the file, the BlueFS client on the media player contacts the token to obtain the content key. Over the secure session established previously, the BlueFS client sends the token the protector encrypted with the KEK that includes both the policy hash and the content key. The BlueFS client on the media player also sends the token the policy for the file it wishes to read. The token decrypts the protector, hashes the specified policy, and verifies that the computed hash matches the one stored in the protector.

The token next verifies that the media player and its current software environment is in accordance with the policy for the specified content. As described by McCune et al. [17], the token can independently compute the PCR quote from the manifest and confirm that its computed quote matches the value supplied by the media player. If the software environment specified by the PCR quote is in accordance with the policy dictated by the content provider, then the token sends the decrypted content key back to the media player over the secure session. The media player uses this key to decrypt and play the content. Having verified the media player platform, the token trusts it not to leak decrypted content to a third party.

To limit interactions with the token, the media player caches content keys while the token is located nearby. Once a session is established, the player sends a challenge to the token every 30 seconds (the period is configurable). A prompt response to the challenge informs the media player that the token is still located within wireless communication range. If a number of consecutive challenges (currently, one) are not met, the media player

assumes that the token has left its vicinity and destroys any keys cached on its behalf. Any decrypted content is flushed from the kernel buffer cache on the media player. This stops playback of the content.

The media player may continue to cache encrypted content on its local storage. This behavior improves performance by eliminating the need to refetch data from the BlueFS server if the token returns. It also potentially enables prefetching strategies (that we have not yet implemented) where a user's content can be prefetched by a media player in anticipation of her arrival and stored encrypted on a local disk for future use.

5.4 File system changes

5.4.1 Support for encryption

In order to support Cobalt, we made several changes to BlueFS. First, we added support for token-encrypted content. The metadata of each file stored in BlueFS can optionally contain two new fields: the protector that stores the encrypted policy hash and content key, as well as the unique BlueFS identifier of the token that can decrypt the protector.

Cobalt metadata fields can be set by any application executing with a userid that has write permission for a content file. The token first creates the file and writes the encrypted content using normal file system calls, then uses an IPC to add the metadata. Encrypted content is publicly readable — this allows an ad hoc media player to fetch and cache the content. However, the media player cannot decrypt the content without the cooperation of the token, nor can it modify the content. As mentioned in Section 2.3, file-specific metadata such as ID3 tags is unencrypted in our current implementation.

The BlueFS daemon only decrypts content immediately before providing it to the kernel as the result of a file read. If an encrypted content key is specified for a file, BlueFS verifies that it has established a secure session for the associated token. If no such session exists, it returns an error. Otherwise, it asks the token to decrypt the content key as described in Section 5.3. The content key is then used to decrypt the content.

To improve performance, a BlueFS client maintains a cache of decrypted keys for each connected token. All keys associated with a token are flushed from the cache if the specified number of consecutive challenge responses are not received from the token. At that time, the daemon also makes an upcall into the kernel to instruct the kernel module to flush any decrypted content associated with the flushed keys. Since content cached on storage devices is encrypted, no action is taken to destroy or evict on-disk files (however, files may be evicted later due to capacity constraints).

5.4.2 Support for read-only federation

Ordinarily, a BlueFS client is tightly bound to its server. A client registers with its server and receives a unique identifier. When a client caches a file, the server maintains a callback, which is a promise to notify the client if the file changes. Callbacks are maintained even when a client is disconnected to eliminate the need for full disk scans on reconnection.

The binding between an ad hoc media player and a BlueFS server must necessarily be more loose. We envision that the client running on the media player will be associated with the server of its owner. However, it must be able to read content from other servers in order to play the content of different users. We enable this loose binding through read-only federation.

We added a `federate` IPC to the BlueFS client that takes as input the IP address or hostname of a BlueFS server. When this function is invoked, the client connects to the server and assigns it a temporary *volume identifier* (like AFS [10] and Coda [12], BlueFS reserves the high-order 32-bits of its file identifier for an administrative volume identifier). The client maintains a mapping between the actual volume identifier that is permanently chosen by the server with which it is federated and the temporary volume identifier that has been assigned. After reading data from the server, the client changes the actual volume identifier to the temporary one. It makes the reverse change when sending requests to the server.

The server maintains callbacks for its federated clients so that it can maintain cache consistency when files change. However, in contrast to permanent clients for which it maintains persistent callbacks, the server immediately drops all callbacks held on behalf of a federated client once that client disconnects. Consequently, a client evicts all files that it has cached from a federated server as soon as it disconnects. Users may trigger an explicit disconnection with a `defederate` IPC.

Currently, federated clients may not modify files. While this may not suffice for all applications, read-only federation fits the needs of ad hoc content access. Media players can read public but encrypted content from federated servers and access the content with the cooperation of a Cobalt token. File updates are rare for media files. For those rare updates, e.g., a change to a song rating, a Cobalt user can make the change directly using his token, which is a permanent client of his server.

5.4.3 Support for other distributed storage

Although Cobalt was developed as an extension to BlueFS, one could potentially modify other distributed storage systems to use Cobalt. Cobalt has less than 5000 lines of source code, most of which could be reused.

The most significant change that would be required is for clients that run on ad hoc media players to be made Cobalt-aware. They need to establish secure sessions with mobile tokens and use the TPM to attest to the integrity of software running on the player. Such clients should restrict distribution of content to only authorized software components. A distributed storage solution must be able to associate Cobalt metadata (the policy, protector, and token identifier) with each file. It should also support a search mechanism, similar to persistent queries, that allows users to specify content to share.

For example, Cobalt could potentially be used with data stored on a Web server. One would need to build a Cobalt-aware Web client and provide a mechanism for a token to specify files to share. Cobalt metadata could be embedded in HTTP headers.

6 Evaluation

Our evaluation answers the following questions:

- What is the overhead of using Cobalt during content acquisition?
- What is the overhead of using Cobalt during content playback?
- Can Cobalt enable new applications?

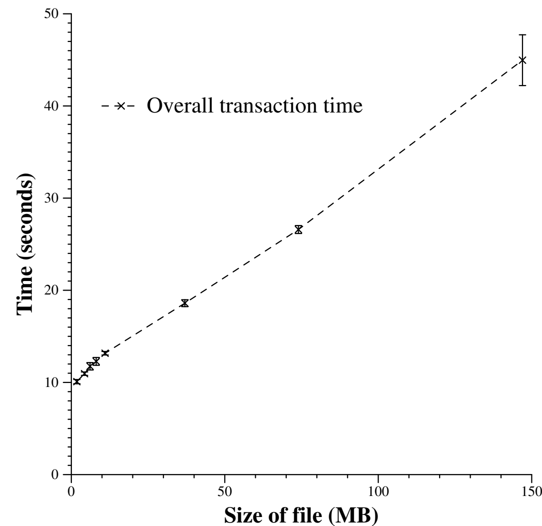
6.1 Methodology

In the following experiments, the Cobalt token is a Motorola E680i cell phone with a 300 MHz XScale processor. The cell phone is a BlueFS client that runs MontaVista Linux Consumer Electronics Edition 3.0. It communicates with the other computers via an SD/MMC 802.11b card. The BlueFS server runs on a Dell GX620 desktop with a 3.4 GHz Pentium 4 processor and 2 GB of DRAM. The desktop also runs a BlueFS client that is used during content acquisition. We use an IBM X40 laptop with a 1.2 GHz Pentium M processor and 784 MB of RAM as both the content provider and ad hoc media player. The laptop and desktop run Fedora Core 4 (Linux kernel 2.6.15) and are connected by 100 Mb/s Ethernet.

6.2 Content acquisition

We first measured the overhead that Cobalt adds to content acquisition. In this experiment, the token initiates the acquisition of a media file from the content provider. The BlueFS client running on the desktop is used as a helper during content acquisition, as described in Section 5.2.

Figure 2 shows how the total acquisition time varies with the size of the content being acquired. From the graph, it can be seen that there is an approximately 10 second



This graph shows the time to acquire content for audio and video files of various sizes. Each result is the mean of 5 trials — the error bars are 90% confidence intervals.

Figure 2. Time to acquire content using Cobalt

fixed cost for acquiring a file plus a variable cost that is roughly linear with the size of the file. Table 1 shows detailed results for 2 MP3 audio files and 2 MP4 videos. For the smaller audio files, the majority of the acquisition time is used to establish a secure session between the token and the content provider. In these experiments, we assume that no prior relationship exists between these two parties; thus, each must send the other a certificate signed by a root authority to establish its identity.

The second column of Table 2 provides further detail about session establishment by detailing the time for the token to perform the individual components of the station-to-station protocol. This step is especially time-consuming since it requires the limited processor of the cell phone to perform public-key cryptography. For reference, Table 2 shows that if we replace the cell phone with the X40 laptop, session establishment requires an order of magnitude less time. Thus, as cell phone processors continue to improve, this component of the acquisition cost will diminish.

Once a secure session has been established, a symmetric session key is used for all further communication. Therefore, the session establishment time does not increase with the size of the file being transferred. Further, if multiple files are being acquired, the session need only be established once.

In contrast, Table 1 shows that the time to encrypt the content, transfer it to the helper, and store the data in BlueFS is roughly proportional to the size of the file being stored. For the larger video files, these activities comprise the majority of the time spent acquiring the content.

Operation	1.8 MB MP3	11 MB MP3	37 MB Video	147 MB Video
Secure session established	7.6 (±0.2)	7.5 (±0.3)	7.2 (±0.2)	7.5 (±0.3)
Content encrypted by provider	0.3 (±0.0)	1.8 (±0.0)	4.6 (±0.2)	14.3 (±0.1)
Content fetched and stored by helper	1.3 (±0.0)	3.0 (±0.2)	5.8 (±0.4)	22.3 (±2.6)
Metadata stored by token	0.9 (±0.1)	0.9 (±0.0)	1.0 (±0.2)	0.8 (±0.1)
Total acquisition time	10.1 (±0.2)	13.2 (±0.2)	18.6 (±0.4)	45.0(±2.8)

This table shows the time (in seconds) to acquire content using Cobalt for files of varying sizes. Each result is the mean of 5 trials — 90% confidence intervals are given in parentheses. The first row must be performed once per session, while the remaining rows are per-file costs.

Table 1. Time to acquire content using Cobalt

Operation	Cell phone (seconds)	Laptop (seconds)
Diffie-Hellman parameter generation	2.49 (±0.11)	0.14 (±0.00)
Preparation of signed exponentials and certificate encryption	3.17 (±0.17)	0.21 (±0.01)
Verification of certificate and signed exponentials	1.29 (±0.01)	0.06 (±0.00)
Other (exponential exchanges, key derivation, network, etc.)	0.60 (±0.01)	0.11 (±0.01)
Total session establishment time	7.56 (±0.19)	0.51 (±0.01)

This table shows the time to establish a secure session using the station-to-station protocol between the provider (Dell GX620) and both, the token (Motorola E680i) and the helper (IBM X40 laptop). Each value is the mean of five trials — 90% confidence intervals are given in parentheses.

Table 2. Detailed breakdown of the time to establish a secure session

However, these activities are not Cobalt-specific, as most existing methods for acquiring protected content must encrypt data, transmit it over the network, and store it on a destination computer. In fact, this experiment underestimates the network cost of content acquisition since the BlueFS desktop and the content provider communicate via local Ethernet. For instance, if the server's connection to the network were a 5 Mb/s cable link, transmitting the 147 MB file would take approximately four minutes. Thus, as network speeds decrease, Cobalt overhead becomes a smaller proportion of the total acquisition time.

The final activity, metadata creation by the token, consists of generating the protector and storing it in BlueFS. Since the metadata size is independent of the content size, the time to perform this activity is constant.

Overall, we are encouraged by these results since the overhead added by Cobalt (establishing the secure session and storing metadata) is less than 9 seconds and does not increase significantly with the size of the content being acquired. The vast majority of Cobalt overhead results from the need to establish a secure session between the content provider and token — this overhead will decrease as cell phone processors become more powerful.

6.3 Content playback

We next evaluated the time to access content using Cobalt. In this experiment, we use the X40 laptop to represent an ad hoc media player. The laptop runs a BlueFS client. It uses xmms to play MP3s and VLC to play video. To specify which files are shared with the media player, we created a persistent query that matched 1500 MP3 files stored on the BlueFS server.

Table 3 shows the time for a token to associate with a media player and specify the content that will be shared. The majority of the time is required to establish a secure session between the token and player. The secure session is necessary to confirm the identity of each party, since a media player may only wish to accept content from known sources, and the token must verify that the media player is a trusted platform that meets the security policy for the content being shared. As part of session establishment, the token receives and caches the media player's PCR quote and software manifest, as provided by the player's TPM hardware. Since our laptop does not have TPM hardware, the laptop transmits precomputed values (a SHA-1 hash for the PCR quote and 1 KB file for the manifest) to the token on request.

Cobalt takes 4 seconds to create a persistent query specifying the content to be shared — most of this time is spent resolving the path names associated with the media files in the query. Currently, each path resolution requires multiple remote procedure calls to the server — based on these results, we are currently considering methods for pipelining these operations to reduce latency.

Table 4 shows the per-file costs for playing content once the token has associated with an ad hoc media player. When the first 4 KB data block is read from a Cobalt-protected file, the BlueFS client running on the media player asks the token to decrypt the content key. Thus,

Operation	Time (seconds)
Secure session establishment	7.9 (±0.2)
Media player selection and TPM verification	0.2 (±0.0)
Persistent query creation and content path resolution	4.0 (±0.2)
Playlist creation	0.3 (±0.1)
Total association time	12.3 (±0.3)

This figure shows how long it takes a Cobalt token to associate with a nearby media player and create a playlist with 1500 MP3s. Each result is the mean of five trials — 90% confidence intervals are given in parentheses.

Table 3. Time for the token to associate with a media player

Operation	Time (seconds)
First block decryption time	0.273 (±0.013)
Subsequent block decryption time	0.001 (±0.000)

This table shows the time to decrypt content when playing it on an ad hoc media player. The first 4 KB block decryption time includes the time for the token to verify the policy and decrypt the content key. Decryption of subsequent 4 KB blocks is much quicker since the media player caches decrypted content keys. Each result is the mean of five trials — 90% confidence intervals are given in parentheses.

Table 4. Decryption time

the first block decryption time includes the time taken by the token to decrypt the protector, verify the policy hash, check the policy against the media player's PCR quote and manifest, and return the content key if all checks pass. Cobalt decrypts the first file block in 273 ms, while it takes only 1 ms to decrypt subsequent blocks since the content key is cached.

We have experimentally verified that Cobalt successfully prevents the media player from decrypting content stored on the federated BlueFS server that is not explicitly shared by the persistent query (e.g., in the above experiment, the media player is unable to play the video files since only music files were shared). We have also verified that the token prevents the media player from decrypting content when its manifest does meet the policy specified for a given file. Finally, our results show that when the token leaves the vicinity of the media player, playback of the video ceases after approximately 30 seconds due to the absence of challenge responses.

6.4 Decryption CPU load

We used the `top` utility to measure the CPU consumption of Cobalt while decrypted content is accessed. During playback of the video, Cobalt consumes 2.1% of the CPU on the laptop while the VLC application uses 10.9% of the CPU. When MP3 audio files are played, Cobalt consumes 0.7% of the CPU while xmms uses 0.3% of the CPU. These results match our expectations: since video files have a higher data rate than music files, Cobalt must decrypt more data per second for the videos.

6.5 Case study: Adaptive playlists

We also explored a potential new application that Cobalt enables. Typically, when visiting a friend's house, the only content available is that which resides in the friend's music collection. However, with Cobalt, guests can pool their content to create a more diverse set of music. With this greater pool of potential content comes a problem: not all music may be enjoyable to everyone in the room.

To address this scenario, we built a Cobalt application that uses persistent queries to play only content that is mutually enjoyable to all people located nearby. Each user's Cobalt token sends a persistent query that lists their most highly rated songs to the media player. The media player compares the results of all queries and creates an adaptive playlist that consists only of songs that are in a specified number of query results (currently, all of them). This application assumes that there is uniformity in labeling MP3s, which seems reasonable given the availability of ID3 repositories such as freedb [8].

Table 5 shows the time for the media player to create an adaptive playlist. In this experiment, the owner of the media player specifies a query that matches on 650 songs. The owner of the Cobalt token specifies a query that matches on 1500 songs. When these queries are combined, the adaptive playlist consists of 650 songs that were included in both query results. Comparing the results in Table 5 with those in Table 3, it is apparent that creating the adaptive playlist takes only about a second longer than creating one based solely on remote content. The extra time is required to create a persistent query for the media player's owner.

Operation	Time (seconds)
Secure session establishment	7.7 (±0.2)
Media Player selection and TPM verification	0.3 (±0.0)
Local persistent query creation and path resolution	1.0 (±0.0)
Remote persistent query evaluation and path resolution	4.1 (±0.0)
Merged playlist creation	0.2 (±0.0)
Total time to create a new adaptive playlist	13.2 (±0.2)

This table shows the time needed by a guest's token and an ad hoc media player to create a new adaptive playlist with 650 MP3s from a collection of 1500 MP3s. Each value is the mean of 5 trials — 90% confidence intervals are given in parentheses.

Table 5. Time to create an adaptive playlist

7 Related work

To the best of our knowledge, Cobalt is the first system to use a mobile token to assist in the secure playback of protected content on ad hoc media players. Specifically, Cobalt separates content distribution from authorization by utilizing a distributed file system (BlueFS) as the distribution channel and a mobile device such as a cell phone to perform authorization on behalf of a content provider.

Content secured by Cobalt is protected by a digital container as described by Sibert et al. [25]. Cobalt extends this scheme by storing the content key in a trusted mobile device that can be used to decrypt the content when requested by the user.

Cobalt builds on Zero-Interaction Authentication [5]. ZIA introduced the notion of proximity-based encryption in which users carry a wearable token that announces their presence to their mobile computer. The token exchanges periodic messages with the computer to confirm its presence. If a user moves away from her computer, ZIA encrypts sensitive data stored on disk and in memory. When the user returns, ZIA decrypts the data so that it can again be accessed.

Cobalt differs from ZIA by focusing on scenarios where a single user does not own all computers. Cobalt uses TPMs to let content providers verify the integrity of a token and to allow the token to validate the integrity of a media player. Further, Cobalt tokens can dynamically associate with ad hoc media players. In contrast to ZIA which decrypts all files in a user's presence, Cobalt users can scope the specific files that they wish to decrypt using persistent queries.

More generally, Cobalt is an example of *splitting trust* [3, 26], in which a small, trusted device performs certain critical functions while a more resourceful computer executes the more demanding part of an application.

Pierce and Mahaney [24] have advocated using cell phones to perform additional functionality for usability reasons; Cobalt follows their advice in that it allows its user to interact with the system via their phone rather than through the interface of an ad hoc media player.

Cobalt assumes that tokens and ad hoc media players are deployed on a trusted computing platform that meets the Trusted Platform Module standard [28] defined by the Trusted Computing Group [27]. Currently, this standard is being extended to better support mobile devices such as the Cobalt token [19]. BitE [17] extends the TPM architecture by showing how a software manifest and PCR quote can be used to verify the integrity of a trusted device. Cobalt leverages these ideas when it checks the integrity of the token and media player. BitE provides a secure channel through which a user can enter sensitive data using their phone. Since Cobalt focuses on integrating protected content and distributed storage, the manner in which it uses the manifest and PCR quote is different from how they are used in BitE. We have tried to make Cobalt as agnostic as possible with regard to the trusted platform on which it runs; potentially, this could enable Cobalt to run on alternative architectures such as XOM [15] and Terra [9].

Commercial systems such as Apple's iTunes Music Store, Yahoo's Music Unlimited and Microsoft's Zune also deploy content protection mechanisms. Subscription-based services, such as Yahoo! Music Unlimited [30], allow users to play content on ad hoc media players after explicitly authenticating with a userid and password. Apple's iTunes Music Store [1] allows a user to purchase protected content and stream it to an ad hoc media player. Playback commences only after the user provides her Apple userid and password to the iTunes application on the ad hoc media player [2]. In contrast, Cobalt improves usability as it removes the requirement of entering a userid and password. Additionally, Cobalt improves privacy as the content provider is not informed every time the user accesses content from an ad hoc media player. Finally, Cobalt better protects the interests of content providers as the physical proximity of a Cobalt token such as a cell phone is a better indicator of the presence of the content owner than a password, which can be entered by another person.

Microsoft's Zune can wirelessly discover another Zune in its vicinity and share media. However, Zune places usage restrictions such as disallowing repeated reception of the same content and limiting the number of times shared content is played [18]. In comparison to Cobalt, Zune operates on a different model where content acquisition, not playback, is proximity-based. Unlike Zune whose playback policy is based on the number of accesses, Cobalt permits playback of protected content from ad hoc media players as long as the Cobalt token is in its vicinity.

Many distributed file systems support a form of federation. For instance, the Self-certifying File System [16] supports secure federation through symbolic links that include the public key of the federated server. AFS [10] has long supported a global namespace. Coda [12] has been recently enhanced to enable clients to access data in more than one cell, and the Glamour project [29] has added federation to NFSv4. Other file systems such as NFSv3 [4] and CIFS [13] allow ad hoc clients to connect to arbitrary servers. Cobalt could potentially leverage these federation mechanisms to allow ad hoc media players to access data from other file systems.

8 Future work

In the future, we hope to investigate what other novel applications are enabled by Cobalt. The presence of personal mobile devices such as cell phones provides valuable context about which people are physically present. These devices can inform their environment about the tastes and preferences of their users, allowing customization of pervasive applications. Adaptive playlists are one example of such customization. Alternatively, the Cobalt token could record which content has been recently played in the presence of its user to avoid repeats or help start playback of content such as movies at the place where its user last left off. Context could be used to manage deletion of old content; for instance, a DVR might delay automatic deletion of a TV show until all users who typically watch that show have viewed it.

We also would like to explore prefetching and caching policies for ad hoc media players. Given sufficient storage, a media player may choose to hide network delays by fetching encrypted content before it is played (perhaps using a playlist to anticipate what might be played). The media player may also choose to cache encrypted content in case a visitor who has recently departed returns in the future. While ad hoc media players are currently limited to read-only access by the BlueFS federation mechanism, we are considering adding the ability for a token to grant update permission by signing and passing a capability for the authorized access to a media player.

Cobalt currently supports only one token per file. We plan to support more than one token for a file by storing multiple token identifiers and protectors in the BlueFS metadata. Use of Cobalt does not preclude a user from authorizing additional devices that they own to play content using existing provider mechanisms. For instance, a user might authorize a home computer and download content directly to its hard drive, then also store a copy of the content using Cobalt so that it can be accessed on ad hoc media players.

Finally, Cobalt currently assumes a one-size-fits-all policy for how content is invalidated when the token is inaccessible. A better solution would be to have the per-file policy specify the amount of time that a file would remain valid after the token departs. Potentially, the per-file policy could also specify the minimum frequency for challenge-response messages and the maximum number of consecutive responses that can be missed before the token is assumed to no longer be present.

9 Conclusion

The goals of content protection often conflict with those of a distributed file system: the former is designed to make data less accessible, while the latter is designed to make data more accessible. Cobalt is targeted at reaching a reasonable compromise between these two goals that meets the needs of both users and content providers. Cobalt bases its authorization on the physical presence of a user and leverages a personal mobile device such as a cell phone to determine when a user is located nearby. Our results show that the overhead of Cobalt is quite reasonable. Our case study shows that Cobalt can also add value by enabling context-sensitive applications such as adaptive playlists.

Acknowledgments

We thank Ed Nightingale and Dan Peek for their help with BlueFS. Manish Anand, Evan Cooke, our shepherd, Mike Swift, and the anonymous reviewers provided valuable feedback about these ideas. The work is supported by the National Science Foundation under award CNS-0306251. Jason Flinn is supported by NSF CAREER award CNS-0346686. Intel Corp. and Motorola Corp. have provided additional support. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of NSF, Intel, Motorola, the University of Michigan, or the U.S. government.

References

[1] APPLE. iTunes Music Store Customer Service - Authorizing your computer. http://www.apple.com/support/itunes/musicstore/authorization/.

[2] APPLE. iTunes Tutorial - Sharing your music on your local network. http://www.apple.com/support/itunes/windows/tutorial/segment102094b.htm%l.

[3] BALFANZ, D., AND FELTEN, E. W. Hand-held computers can be better smart cards. In *Proceedings of the 1999 USENIX Security Symposium* (1999).

[4] CALLAGHAN, B., PAWLOWSKI, B., AND STAUBACH, P. NFS Version 3 Protocol Specification. Tech. Rep. RFC 1813, IETF, June 1995.

[5] CORNER, M. D., AND NOBLE, B. D. Zero-interaction authentication. In *Proceedings of the 8th International Conference on Mobile Computing and Networking* (September 2002), pp. 1–11.

[6] DIFFIE, W., VAN OORSCHOT, P. C., AND WIENER, M. J. Authentication and authenticated key exchanges. *Designs, Codes and Cryptography 2*, 2 (1992), 107–125.

[7] DWORKIN, M. NIST Special Publication 800-38A: Recommendation for Block Cipher Modes of Operation. Tech. rep., National Institute of Standards and Technology (NIST), 2001.

[8] freedb. http://www.freedb.org.

[9] GARFINKEL, T., PFAFF, B., CHOW, J., ROSENBLUM, M., AND BONEH, D. Terra: A virtual machine-based platform for trusted computing. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles* (Bolton Landing, NY, October 2003), pp. 193–206.

[10] HOWARD, J. H., KAZAR, M. L., MENEES, S. G., NICHOLS, D. A., SATYANARAYANAN, M., SIDEBOTHAM, R. N., AND WEST, M. J. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems 6*, 1 (February 1988).

[11] HU, Y.-C., PERRIG, A., AND JOHNSON, D. B. Wormhole attacks in wireless networks. *IEEE Journal on Selected Areas in Communications 24*, 2 (February 2006), 370–380.

[12] KISTLER, J. J., AND SATYANARAYANAN, M. Disconnected operation in the Coda file system. *ACM Transactions on Computer Systems 10*, 1 (February 1992).

[13] LEACH, P., AND PERRY, D. CIFS: A Common Internet File System. In *Microsoft Interactive Developer* (November 1996).

[14] LI, J., KROHN, M., MAZIÈRES, D., AND SHASHA, D. Secure untrusted data repository (SUNDR). In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation* (San Francisco, CA, December 2004), pp. 121–136.

[15] LIE, D., THEKKATH, C. A., AND HOROWITZ, M. Implementing an untrusted operating system on trusted hardware. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles* (Bolton Landing, NY, 2003), pp. 178–192.

[16] MAZIÈRES, D., KAMINSKY, M., KAASHOEK, M. F., AND WITCHEL, E. Separating key management from file system security. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles* (Kiawah Island, SC, December 1999), pp. 124–139.

[17] MCCUNE, J. M., PERRIG, A., AND REITER, M. K. Bump in the ether: A framework for securing sensitive user input. In *Proceedings of the USENIX 2006 Annual Technical Conference* (Boston, MA, June 2006).

[18] MICROSOFT. Share Audio Files Zune to Zune. http://www.zune.net/en-us/support/howto/zunetozune/sharesongs.htm.

[19] Mobile Device Security and Trusted Computing - Next Steps. Tech. rep., Trusted Computing Group, 2005. https://www.trustedcomputinggroup.org/groups/mobile.

[20] MOTOROLA. *Motorola E680*, September 2004. http://www.motorola.com/us/products.jsp.

[21] MUMMERT, L., EBLING, M., AND SATYANARAYANAN, M. Exploiting weak connectivity in mobile file access. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles* (Copper Mountain, CO, Dec. 1995).

[22] NIGHTINGALE, E. B., AND FLINN, J. Energy-efficiency and storage flexibility in the Blue File System. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation* (San Francisco, CA, December 2004), pp. 363–378.

[23] PEEK, D., AND FLINN, J. EnsemBlue: Integrating consumer electronics and distributed storage. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation* (Seattle, WA, November 2006), pp. 219–232.

[24] PIERCE, J. S., AND MAHANEY, H. Opportunistic annexing for handheld devices: Opportunities and challenges. In *Proceedings of HCIC* (2004).

[25] SIBERT, O., BERNSTEIN, D., AND WIE, D. V. Digibox: A self-protecting container for information commerce. In *Proceedings of the first USENIX Workshop on Electronic Commerce* (New York, New York, 1995).

[26] STAJANO, F., AND ANDERSON, R. The resurrecting duckling: Security issues for ad-hoc wireless networks. In *Proceedings of the 7th International Workshop on Security Protocols* (1999), pp. 172–194.

[27] TCG PC Specific Implementation Specification v1.1. Tech. rep., Trusted Computing Group, 2006.

[28] TCG TPM Specification Version 1.2 Revision 94. Tech. rep., Trusted Computing Group, March 2006. https://www.trustedcomputinggroup.org/specs/TPM.

[29] TEWARI, R., HASWELL, J. M., NAIK, M. P., AND PARKES, S. M. Glamour: A Wide-Area Filesystem Middleware Using NFSv4. Tech. Rep. RJ10368, IBM, June 2005.

[30] Yahoo! Music Unlimited. http://music.yahoo.com/unlimited/.