Wireless Networks 11, 7–19, 2005

Protecting File Systems with Transient Authentication

MARK D. CORNER and BRIAN D. NOBLE *
Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI, USA

Abstract. Laptops are vulnerable to theft, greatly increasing the likelihood of exposing sensitive files. Unfortunately, storing data in a cryptographic file system does not fully address this problem. Such systems ask the user to imbue them with long-term authority for decryption, but that authority can be used by anyone who physically possesses the machine. Forcing the user to frequently reestablish his identity is intrusive, encouraging him to disable encryption.

This tension between usability and security is eliminated through *Transient Authentication*, in which a small hardware token continuously authenticates the user's presence to the laptop over a short-range, wireless link. Whenever the laptop needs decryption authority, it acquires it from the token; authority is retained only as long as necessary. With careful key management, ZIA imposes an overhead of less than 7% for representative workloads, though some infrequent operations suffer greater overheads. The largest file cache on our hardware can be re-encrypted within five seconds of the user's departure, and restored in just over six seconds after detecting the user's return. This secures the machine before an attacker can gain physical access, but recovers full performance before a returning user resumes work. Key granularity plays an important role in determining performance; assigning encryption keys on a per-directory basis limits the cost of an exposed key while maintaining acceptable overhead.

1. Introduction

Mobile devices are susceptible to loss and theft because they are small, light, and easy to carry. Unfortunately, they often contain sensitive data that their owners would prefer to keep private. The consequences of exposing such data range from the inconvenience of canceling credit cards to the public loss of state secrets. If a user were confident that data on a missing laptop could not be viewed by unprivileged eyes, he could simply replace the laptop and restore from backup. Without that confidence, one must assume the worst.

Cryptographic file systems offer the most common common defense against the exposure of persistent data. On-disk data is encrypted and only a legitimate user is able to supply the decryption key. Usually, decryption keys are supplied during login or when mounting the file system, and are retained for later use. Unfortunately, cryptographic file systems do not adequately protect the data on a laptop's disk. If the laptop is lost or stolen, the decryption key goes with it; as long as the key is retained, anyone holding the laptop has access to the data.

The only way to limit this vulnerability is to force the users resupply decryption keys frequently. Unfortunately, users find such reauthentication burdensome; it encourages them to disable security systems that depend on it. For example, Windows can require users to reauthenticate each time their laptop awakens from suspension. Most people who are aware that this feature can be disabled do so.

Security requires frequent reauthentication, but this limits usability. We resolve this tension with *Transient Authentication*, in which the user wears a small authentication token. In turn, the token continuously authenticates to other devices by means of a short-range, wireless link. In Transient Authentication, the token stores the user's capabilities and decryption keys; proof of these is required to perform any sensitive action.

We have applied this model to a cryptographic file system, called *ZIA*, to ensure that the user – or at least her token – is present for disk operations. Whenever the laptop reads file system data, it first obtains a decryption key from the token. If the token (and hence the user) is not present, the read cannot complete. With ZIA, stolen laptops are protected from malicious use; an attacker cannot reproduce the decryption key. The effectiveness of this scheme depends on a token small enough to be worn unobtrusively, such as an IBM Linux watch [30]. This makes the token much less vulnerable to loss or theft than a device that is carried and often set down. The core idea of ZIA is simple, but it requires careful design and implementation. ZIA must not impose undue usability burdens or noticeably reduce file system performance.

The main contribution of this paper is not the construction of a cryptographic file system. Blaze's CFS [2], Zadok's Cryptfs [41], and Microsoft's EFS [29] all address the architecture, administration, and cryptographic methods for a file system. However, all of these systems rely on infrequent authentication, weakening the protection that cryptography provides. Some systems, such as EFS, require the user to reauthenticate after certain events, such as suspension, hibernation, or long idle periods, in an attempt to bound the window of vulnerability. The user must explicitly produce a password when any of these events occur. This burden, though small, will encourage some users to disable or work around the mechanism, rendering its protection forfeit.

* Corresponding author.
E-mail: bnoble@umich.edu

2. Design

ZIA's goal is to provide effective file encryption without reducing performance or usability. All on-disk files are encrypted for safety, but all cached files are decrypted for performance. With its limited hardware and networking performance, the token is not able to encrypt and decrypt file data without a significant performance penalty. Instead, file keys are stored on the laptop's disk, encrypted by a key-encrypting key. Only an authorized token holds the key-encrypting key, thus the token is required to read files. This process is illustrated in figure 1.

There are two requirements for system security. First, a user's token cannot provide key decryption services to other users' laptops. Second, the token cannot send decrypted file keys over the wireless link in cleartext form. Therefore, the token and laptop use an authenticated, encrypted link. Before the first use of a token, the user must unlock it using a PIN. Then he must *bind* the token and laptop, ensuring that his token only answers key requests from his laptop. Next, ZIA mutually authenticates the identity of the token and laptop over the wireless link and exchanges a session encryption key. After authentication, polling ensures that the token, and thus the user, is still present. When the token is out of range, ZIA encrypts cached objects for safety. The cache retains these encrypted pages to minimize recovery time when the user returns, preserving usability. The overall process is illustrated in figure 2. The remainder of this section presents the detailed design of ZIA, starting with the trust and threat model.

2.1. Trust and threat model

Our focus is to defend against attacks involving *physical possession* of a laptop or *proximity* to it. Possession enables a wide range of exploits. If the user leaves his login session open, attacks are not even necessary; the attacker has all of the legitimate user's rights. Even without a current login session, console access admits a variety of well-known attacks, some resulting in root access. An attacker can also bypass the operating system entirely. For example, one can remove and inspect the disk using another machine. A determined attacker might even probe the physical memory of a running machine.

ZIA must also defend against exploitation of the wireless link between the laptop and token: observation, modification, or insertion of messages. Simple attacks include eavesdropping in the hopes of obtaining decrypted file keys. A more sophisticated attacker might record a session between the token and laptop, and later steal the laptop in the hopes of decrypting prior traffic. ZIA defeats these attacks through the use of well-known, secure mechanisms.

We assume that some collection of users and laptops belong to a single administrative domain, within which data can be shared. The domain includes at least one trusted authority to simplify key management and rights revocation. However, the system must be usable even when the laptop is disconnected from the rest of the network.

ZIA does not defend against a trusted but malicious user, who can easily leak sensitive data and potentially extract key material. ZIA does not provide protection for remote users; they must be physically present. Attackers that jam the spectrum used by the laptop-token channel will effectively deny users access to their files. Our work is orthogonal to the prevention of network-based exploits such as buffer overflow attacks.

ZIA's security depends on the limited range of the radio link between the token and the laptop. Repeaters could be used to extend this range, though time-based techniques to defeat such *wormhole* attacks exist [6,19]. Similarly, an attacker with an arbitrarily powerful and sensitive radio could extend the range, though such attacks are difficult given the attenuation of high frequency radios.

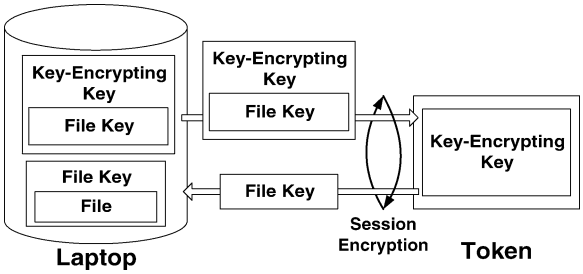


Figure 1. Decrypting file encrypting keys. This figure illustrates the process of file key acquisition. Encrypted file keys are read from disk and shipped to the token. The token decrypts it and returns the file key. Traffic between the laptop and token is encrypted, preventing eavesdroppers from obtaining file keys.

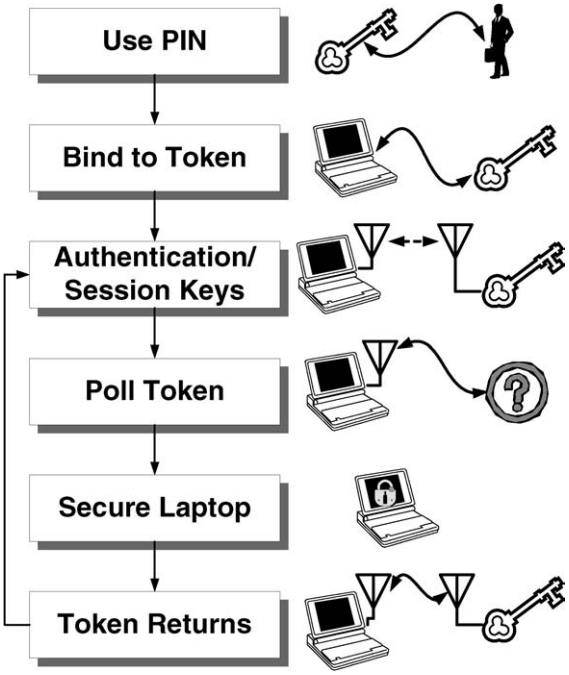


Figure 2. Token authentication system. This figure shows the process for authenticating and interacting with the token. Once an unlocked token is bound to a laptop, ZIA negotiates session keys and can detect the departure of the token.

2.2. Key-encrypting keys

In ZIA, each on-disk object is encrypted by some symmetric key, K_e. The link connecting the laptop and token is slow, and the token is much less powerful than the laptop. Consequently, file decryption must take place on the laptop, not the token. The file system stores each K_e, encrypted by some key-encrypting key, K_k; we write this as $K_k(K_e)$. Only tokens know key-encrypting keys; they are never divulged. A token with the appropriate K_k can decrypt K_e, and hence enable reading any file encrypted by K_e.

In our model, the local administrative authority is responsible for assigning key-encrypting keys. For reliability in the face of lost or destroyed tokens, the administrative authority must also hold these keys in escrow. Otherwise, losing a token is the equivalent of losing all of one's files. Escrowed keys need not be highly available, eliminating the need for oblivious escrow [4] or similar approaches.

Laptops are typically "owned" by a particular user; in many settings, one could provide only a single, unique K_k to each user. However, ZIA must support shared access as well, because most installations within a single administrative domain share notions of identity and privilege. For example, two colleagues in a department may borrow each others' machines, and ZIA cannot preclude such uses. To support sharing, each file key, K_e, can be encrypted by both a *user key*, K_u, and some number of *group keys*, K_g.

The specific semantics of group access and authorization are left to the file system. For example, one can assign key-encrypting keys to approximate standard UNIX file protections. Access to a file in the UNIX model is determined by dividing the universe of users into three disjoint sets: the file's *owner*, members of the file's *group*, and anyone else empowered to log in to that machine. We refer to this last set as the *world*. Each user has a particular identity, and is a member of one or more groups. One could assign a *user key*, K_u, to each user; a *group key*, K_g, to each group; and a *world key*, K_w, to each machine. A user's token holds her specific K_u, each applicable K_g, and one K_w per machine on which she has an account. Each file's encryption key, K_e, is stored on disk, sealed with its owner's key, K_u. If a file was readable, writable, or executable by members of its owning group, $K_g(K_e)$ would also be stored. Finally, $K_w(K_e)$ would be stored for files that are world-accessible. Note that the latter is not equivalent to leaving world-accessible files unencrypted; only those with login authority on a machine would hold the appropriate K_w.

Group keys have important implications for sharing and revocation in ZIA. Members of a group have the implicit ability to share files belonging to that group, since each member has the corresponding K_g. However, if a user leaves a group, that group's K_g must be changed to a new K'_g. Furthermore, the departing user – who is no longer authorized to view these files – may have access to previously-unsealed file keys. As a result, re-keying a group requires that the contents of each file accessible to that group be re-encrypted with a new K'_e, and that new key be re-sealed with the appropriate key-encrypting keys.

Re-keying can be done incrementally. To distribute a new group key, the administrative authority must supply a certified K'_g to the token of each still-authorized user. This must be done in a secure environment to prevent exposure of K'_g. Thereafter, a token encountering a laptop with "old" keys can continue to use it until it is re-keyed. However, this policy must be pursued judiciously, since it increases the amount of data potentially visible to an ejected group member.

2.3. Token vulnerabilities

Tokens provide higher physical security than laptops, since they are worn rather than carried. Unfortunately, it is still possible for a user to lose a token. Token loss is a very serious threat since tokens hold key-encrypting keys. How can we limit the damage of such an occurrence?

The most serious vulnerability surrounding token loss is the extraction of key-encrypting keys. PIN-protected, tamper-resistant hardware [40] makes this more difficult, as does storing all K_k encrypted with some password. In either case, the PIN/password must be known only to the token's rightful owner. At first glance, this seems to merely shift the problem of authentication from the laptop to the token. However, since the token is worn, it is more physically secure than a laptop – it is reasonable to allow long-lived authentication between the token and the user, perhaps on the order of once a day.

Bounding the authentication session between the user and token also prevents an attacker from profitably stealing a token, and then later a laptop. After the authentication period expires, the token will no longer be able to supply any requested K_e. Such schemes can be further improved through the use of server-assisted protocols to prevent offline dictionary attacks [25], with the laptop playing the role of the server to the token's device.

Even tokens that have not been stolen can act as liabilities. Supposed an attacker has a stolen laptop but no token, and is sitting near a legitimate user from the same domain. This *tailgating* attacker can force the stolen laptop to generate key decryption requests that could use one of the legitimate user's key-encrypting keys. If the legitimate token were to respond, the system would be compromised.

To prevent this, we provide a mechanism that establishes *bindings* between tokens and laptops. Before a token will respond to a particular laptop's request, the user must acknowledge that he intends to use *this* token with *that* laptop. There are several ways one can accomplish this. For example, a token with a rudimentary user interface would alert the user when some new laptop first asks it to decrypt a file key. The user then chooses to allow or deny that laptop's current and future requests. As with token authentication, bindings have relatively long but bounded duration; after a binding expires, the token/laptop pair must be rebound. Since a user can use more than one machine, a token may be bound to more than one laptop. Likewise, a laptop may have more than one token bound to it.

User–token authentication and token–laptop binding are necessarily visible to the user, and thus add to the burden of

using the system. However, since they are both long-lived, they require infrequent user action. In practice, they are no more intrusive than having to unlock your office door once daily, without the accompanying threat of forgetting to re-lock it. The right balance between usability and security depends on the physical nature of the token, its user interface capabilities, and the user population.

2.4. Token–laptop interaction

The binding process must accomplish two things: mutual authentication and session key establishment. Mutual authentication can be provided with public-key cryptography [31]. In public-key systems, each principal has a pair of keys, one public and one secret. To be secure, each principal's public key must be certified, so that it is known to belong to that principal. Because laptops and tokens fall under the same administrative domain, that domain is also responsible for certifying public keys.

ZIA uses the Station-to-Station protocol [14], which combines public-key authentication and Diffie–Hellman key exchange. Diffie–Hellman key exchange provides *perfect forward security*; session keys cannot be reconstructed, even if the private keys of both endpoints are known. Once a session key is established, it is used to encrypt all messages between the laptop and token. Each message includes a *nonce*, a number that uniquely identifies a packet within each session to prevent replay attacks [7]. In addition, the session key is used to compute a *message authentication code*, verifying that a received packet was neither sent nor modified by some malicious third party [33].

2.5. Assigning file keys

What is the right granularity at which to assign file encryption keys? A small grain size reduces the data exposed if a file key is revealed, but a larger grain size provides more opportunity for key caching and re-use.

ZIA hides the latency of key acquisition by overlapping it with physical disk I/O. Further, it must amortize acquisition costs by re-using keys when locality suggests that doing so is beneficial. In light of this, we have chosen to assign file keys on a *per-directory* basis.

People tend to put related files together, so files in the same directory tend to be used at the same time. Therefore, many file systems place all files in a directory in the same cylinder group to reduce seek time between them [28]. This makes it difficult to hide key acquisition costs for per-file keys. Instead, since each file in a directory shares the same file key, key acquisition costs are amortized across intra-directory accesses. We explore the impact of various choices of key granularity in section 4.

In our prototype, we store the file key for a directory in a *keyfile* within that directory. The keyfile contains two encrypted copies of the file key; $K_{\mathrm{u}}(K_{\mathrm{e}})$ and $K_{\mathrm{g}}(K_{\mathrm{e}})$, where K_{u} and K_{g} correspond to the directory's owner and group. We have chosen not to implement world keys, but adding them is straightforward. This borrows from the UNIX protection model, though it does not replicate it exactly. AFS, the Andrew File System, makes a similar tradeoff in managing access control lists on a per-directory basis rather than a per-file one [36]. However, AFS is motivated by conceptual simplicity and storage overhead, not efficiency in retrieving access control list entries.

2.6. Handling keys efficiently

Key acquisition time can be a significant expense, so we overlap key acquisition with disk operations whenever possible. Since disk layout policies and other optimizations often reduce the opportunity to hide latency, we cache decrypted keys obtained from the token.

Disk reads provide opportunities for overlap. When a read requiring an uncached key commences, ZIA asks the token to decrypt the key in parallel. Unfortunately, writes do not offer the same opportunity; the key must be in hand to encrypt the data before the write commences. However, it is likely that the decryption key is already in the key cache for writes. To write a file, one must first open it. This open requires a lookup in the enclosing directory. If this lookup is cached, the file key is also likely to be cached. If not, then key acquisition can be overlapped with any disk I/O required for lookup.

Neither overlapping nor caching applies to directory creation, which requires a fresh key. Since this directory is new, it cannot have a cached key already in place. Since this is a write, the key must be acquired before the disk operation initiates. However, ZIA does not need a *particular* key to associate with this directory; any key will do. Therefore, ZIA can prefetch keys from the authentication token, encrypted with the current user's K_{u} and K_{g}, to be used for directories created later. The initial set of fresh keys is prefetched when the user binds a token to a laptop. Thereafter, if the number of fresh keys drops below a threshold, a background daemon obtains more.

Key caching and prefetching greatly reduce the need for laptop/token interactions. However, frequent assurance that the token is present is our only defense against intruders. To provide this assurance, we add a periodic challenge/response between the laptop and the token. The period must be short enough that the time to discover an absence plus the time to secure the machine is less than that required for a physical attack. It also must be long enough to impose only a light load on the system. We currently set the interval to be one second; this is long enough to produce no measurable load, but shorter than the time to protect the laptop in the worst case. Thus, it does not contribute substantially to the window in which an attacker can work.

2.7. Departure and return

When the token does not respond to key requests or challenges, the user is declared absent. All file system state must be protected and all cached file keys flushed. When the user returns, ZIA must re-fetch file keys and restore the file cache

to its pre-departure state. This process should be transparent to the user: it should complete before he resumes work.

There are two reasons why a laptop might not receive a response from the token. The user could truly be away, or the link may have dropped a packet. ZIA must recover from the latter to avoid imposing a performance penalty on a still-present user. To accomplish this, we use the expected round trip time between the laptop and the token. Because this is a single, uncongested network hop, this time is relatively stable. ZIA retries key requests if responses are not received within a tunable amount of time, with a total of three attempts. Retries do not employ exponential backoff, since we expect losses to be due to link noise, not congestion; congestion from nearby users is unlikely because of the short radio range.

If there is still no response, the user is declared absent and the file system must be secured. ZIA first removes all name mappings from the name cache, forcing any new operations to block during lookup. ZIA then walks the list of its cached pages, removing the clear text versions of the pages. There are two ways to accomplish this: writing dirty pages to disk and zeroing the cache, or encrypting all cached pages in place.

Zeroing the cache has the attractive property that little work is required to secure the machine. Most pages will be clean, and do not need to be written to disk. However, when the user returns, ZIA must recover and decrypt pages that were in the cache. They are likely to be scattered across the disk, so this will be expensive.

Instead, ZIA encrypts all of the cached pages in place. Each page belongs to a file on disk, with a matching file key. The page descriptor holds a reference to the cached, decrypted key. Referenced keys may not be evicted – they are *wired* in the cache. Without a corresponding key, there would be no way to encrypt a cached page, and such keys cannot be obtained from the now-departed token.

The expense of encryption is tolerable given our goal of foiling a physical attack. For example, the largest file cache we can observe on our hardware can be encrypted within five seconds. To be successful, an attacker would have to gain possession of the machine and extract information within that time – an unlikely occurrence.

While the user is absent, most disk operations block until the token is once again within range; ZIA then resumes pending operations. This means that background processes cannot continue while the user is away. In a physically secure location, such as an office building, fixed beacons can provide authentication in lieu of the user. Unfortunately, such beacons would not prevent intra-office theft and must be used judiciously. At insecure locations, such as an airport, the user must not leave unencrypted data exposed and background computation should not be enabled. This would defeat the purpose of the system.

2.8. Laptop vulnerabilities

What happens when a laptop is stolen or lost? Since ZIA automatically secures the file system, no data can be extracted from the disk. Likewise, all file keys and session keys have been zeroed in memory. However, the laptop's private key, s_d, must remain on the laptop to allow transparent re-authentication. If the attacker recovers s_d, he can impersonate a valid laptop. To defend against this, the user must remove the binding between the token and the stolen device. This capability can be provided through a simple interface on the token. Use of tamper-resistant hardware in the laptop would make extracting s_d more difficult.

Instead of offline inspection, suppose an attacker modifies the device and returns it to a user. Now the system may contain trojans, nullifying all protections afforded by ZIA. Any device that is stolen, and later recovered, should be regarded as suspect and not used. Secure booting [9,21] can be used to guard against this attack.

3. Implementation

Our implementation of ZIA consists of two parts: an in-kernel encryption module and a user-level authentication system. The kernel portion provides cryptographic I/O, manages file keys, and polls for the token's presence. The authentication system consists of a client on the user's laptop and a server on the token, communicating via a secured channel.

Figure 3 is a block diagram of the ZIA prototype. The kernel module handles all operations intended for our file system and forwards key requests to the authentication system. We used FiST [42], a tool for constructing stackable file systems [17,35], to build our kernel-resident code. This code is integrated with the Linux 2.4.18 kernel.

The authentication system consists of two components. The client, `keyiod`, runs on the laptop, and the server, `keyd`, runs on the token; both are written in C. The client handles session establishment and request retransmission. The server must respond to key decryption and polling requests. The processing requirements of `keyd` are small

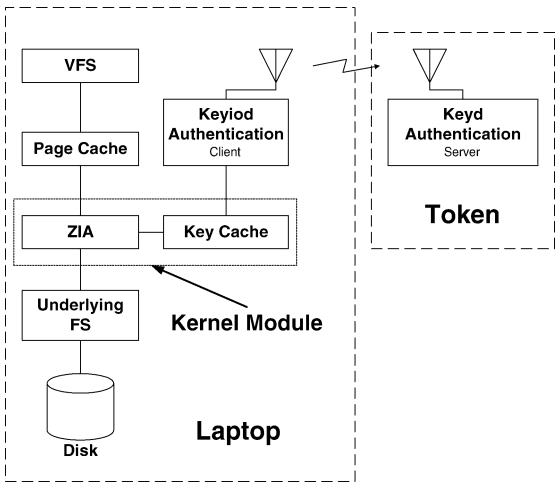


Figure 3. An overall view of ZIA. This figure shows ZIA's design. The kernel module handles cryptographic file I/O. The authentication client and server manage key decryption and detect token proximity. A key cache is included to improve performance.

enough that it can be implemented in a simple, low-power device.

3.1. Kernel module

In Linux, all file system calls pass through the Virtual File System (VFS) layer [23]. VFS provides an abstract view of the file systems supported by the OS. A stackable file system inserts services between the concrete implementations of an *upper* and *lower* file system. FiST implements a general mechanism for manipulating page data and file names; this makes it ideal for constructing cryptographic file services. The FiST distribution also includes a proof-of-concept cryptographic file system, *Cryptfs*.

3.1.1. File and name encryption

The kernel module encrypts both file pages and file names with the Rijndael cipher [13]. We selected Rijndael for two reasons. First, it has been chosen as NIST's Advanced Encryption Standard, AES. Second, it has excellent performance, particularly for key setup – a serious concern in the face of per-directory keys.

ZIA preserves file sizes under encryption. File pages are encrypted in cipher block chaining (CBC) mode with a 16 byte block. We use the inode and page offsets to compute a different initialization vector for each page of a file. Tail portions that are not an even 16 bytes are encrypted in cipher feedback mode (CFB). We chose CFB rather than ciphertext stealing [12], since we are concerned with preventing exposure, not providing integrity.

ZIA does not preserve the size of file names under encryption; they are further encoded in Base-64, ensuring that encrypted filenames use only printable characters. Otherwise, the underlying file system might reject encrypted file names as invalid. In exchange, limits on file and path name sizes are reduced by 25%. Cryptfs made the same decision for the same reasons [41].

The kernel module performs two additional tasks. First, the module prefetches fresh file keys to be used during directory creation. Second, the module manages the storage of encrypted keys. The underlying file system stores keys in a keyfile, but keyfiles are not visible within ZIA. This is done for transparency, not security; on-disk file keys are always encrypted.

3.1.2. Polling, disconnection, and reconnection

ZIA periodically polls the token to ensure that the user is still present. The polling period must be longer than a small multiple of network round-trip time, but shorter than the time required for an adversary to obtain and inspect the laptop. This window is between hundreds of milliseconds and tens of seconds. We chose a period of one second; this generates unnoticeable traffic, but provides tight control. Demonstrated knowledge of the session key is sufficient to prove the token's presence. Therefore, a poll message need only be an exchange of nonces [7]: the device sends a number, n, encrypted with the key and the token returns $n + 1$ encrypted by the same key. The kernel is responsible for polling; it cannot depend on a user-level process to declare the token absent, since it must be fail-stop. Similarly, if the user suspends the laptop, or it suspends itself due to inactivity, the kernel treats this as equivalent to loss of communication.

If the kernel declares the user absent, it secures the file system. Cached data is encrypted, decrypted file keys are flushed, and both are marked invalid. We added a flag to the page structure to distinguish encrypted pages from those that were invalidated through other means. Most I/O in ZIA blocks during the user's absence; non-blocking operations return the appropriate error code.

When `keyiod` reestablishes a secure connection with the token, two things happen. First, decrypted file keys are refetched from the token. Second, file pages are decrypted and made valid. As pages are made valid, any operations blocked on them resume. We considered overlapping key validation with page decryption to improve restore latency. However, the simpler scheme is sufficiently fast.

3.2. Authentication system

The authentication system is implemented in user space for convenience. All laptop–token communication is encrypted and authenticated by session keys plus nonces. Communication between the laptop and the token uses UDP rather than TCP, so that we can provide our own retransmission mechanism. This enables a more aggressive schedule, since congestion is not a concern. We declare the user absent after three dropped messages; this parameter is tunable. The token, in the form of `keyd`, holds all of a user's key-encrypting keys. Since session establishment is the most taxing operation required of `keyd`, and it is infrequent, `keyd` is easily implemented on low-power hardware.

4. Evaluation

In evaluating ZIA, we set out to answer the following questions:

- What is the cost of key acquisition?
- What overhead does ZIA impose? What level of overhead do particular operations incur?
- Can ZIA secure the machine quickly enough to prevent attacks when the user departs? Can ZIA recover system state before a returning user resumes work?
- What are the energy requirements for the token?

To answer these questions, we subjected our prototype to a variety of benchmarks. For these experiments, the client machine was an IBM ThinkPad X24, with 256 MB of physical memory, a 1.1 GHz Pentium III CPU, and an IBM Travelstar IC25N030ATC504-0 30.0 GB IDE disk drive with a 12 ms average seek time. The token was a Compaq iPAQ 3870 with 64 MB of RAM. Both the iPAQ and laptop ran Linux 2.4.18. For each experiment the laptop and token were connected by either an 802.11 wireless network running in ad hoc mode at

Experiment	Time (msec)
Decrypt key w/802.11	9.56 (0.48)
Decrypt key w/Bluetooth	64.63 (7.72)
Decrypt key processing	2.58 (0.22)
10 new keys w/802.11	48.38 (8.65)
10 new keys w/Bluetooth	131.35 (8.18)
10 new keys processing	31.77 (1.38)

Figure 4. Key acquisition cost. This table shows the round trip time required to acquire fresh and decrypted keys from the token. Token processing time is also shown for both requests. Bluetooth's higher latency substantially effects key acquisition time.

1 Mb/s or by Bluetooth running in PAN mode. The X24 has a builtin 802.11 interface, while the iPAQ used a PCMCIA expansion pack with an 802.11 card. The Bluetooth setup consisted of a PCMCIA card in the laptop, the iPAQ's internal Bluetooth interface, and the Bluez Linux protocol stack. The typical round trip time for standard ping messages was 55 ms for Bluetooth and 1 ms for 802.11. All AES file keys and session keys were 128 bits long. The token is somewhat more powerful than current wearable devices. However, the rapid advancements in embedded, low-power devices makes this a realistic token in the near future.

4.1. Key acquisition

Our first task is to compare the cost of key acquisition with typical file access times. To do so, we measured the round trip time for two typical laptop–token interactions: a request for a single key decryption and a request for a batch of 10 fresh keys. We also measured the amount of token processing time for each request. Note that the round trip time includes this processing time. The results are shown in figure 4. In each case, we used 50 data points and we report the mean and standard deviation.

Using 802.11, the cost to decrypt a key is similar to the average seek time of the disk in our laptops, though layout policy and other disk optimizations will tend to reduce seek costs. Our Bluetooth radios had much higher latencies than 802.11, impacting the cost of acquiring fresh and decrypted keys. Computation time could be reduced by caching decrypted keys and pre-generating fresh keys.

4.2. ZIA overhead

Our second goal is to understand the overhead imposed by ZIA on typical system operation. Our benchmark is similar to the Andrew Benchmark [18] in structure. The Andrew Benchmark consists of copying a source tree, traversing the tree and its contents, and compiling it. We use the Apache 2.0.43 source tree. It is 28.2 MB in size; when compiled, the total tree occupies 59.7 MB. We pre-configure the source tree for each trial of the benchmark, since the configuration step does not involve appreciable I/O in the test file system.

While the Andrew Benchmark is well known, it does have several shortcomings; the primary one is a marked dependence on compiler performance. In light of this, we also subject ZIA to three I/O-intensive workloads: directory creation, directory traversal, and tree copying. The first two highlight the cost of key creation and acquisition. The third measures the cost of data encryption and decryption.

4.2.1. Modified Andrew Benchmark

We compare the performance of Linux's ext2fs against six stacking file systems: Base+, Cryptfs, ZIA-FS, ZIA, ZIA-FILE, and ZIA-NPC. Base+ is a null stacked file system. It transfers file pages but provides no name translation. Cryptfs adds file and name encryption; it uses a single, static key for the entire file system. Both Base+ and Cryptfs are samples from the FiST distribution [42]. To provide a fair comparison, we replaced Blowfish [37] with Rijndael in Cryptfs, improving its performance. The ZIA file system is as described in this paper. For comparison purposes ZIA-FS uses one key for the whole file system and ZIA-FILE uses one key per file and one per directory listing. ZIA-NPC uses per-directory keys and obtains a key on every disk access; it provides neither caching nor prefetching of keys.

Each experiment consists of 20 runs. Before each set, we compile the same source in a separate location. This ensures that the test does not include the effects of loading the compiler and linker from a separate file system. Each run uses separate source and destination directories to avoid caching files and name translations. The results for 802.11 and Bluetooth are shown in figure 5.

The results for ext2fs give baseline performance. The result for Base+ quantifies the penalty for using a stacking file system. Cryptfs adds overhead for encrypting and decrypting file pages and names. ZIA encompasses both of these penalties, plus any costs due to key retrieval, token communication and key storage.

For this benchmark, 802.11 ZIA imposes less than a 7% penalty over ext2fs. Its performance is statistically indistinguishable from that of Cryptfs, which uses a single key for all cryptographic operations. Using Bluetooth, ZIA imposes less than a 15% overhead over ext2fs. This is due to increased overhead in the `mkdir` and `cp` phases; these expose Bluetooth's higher cost in retrieving fresh keys.

We also ran the experiment using ZIA-NPC, which uses no key caching or prefetching optimizations. The results are shown in figure 6. Key caching is critical; without it, ZIA-NPC is more than four times slower than ext2fs. Bluetooth's extra latency exaggerates this effect making ZIA-NPC more than 25 times slower than ext2fs.

4.2.2. I/O intensive benchmarks

Although the Modified Andrew Benchmark shows only a small overhead, I/O intensive workloads will incur larger penalties. We conducted three benchmarks to quantify them. The first two stress directory operations, and the third measures the cost of copying data in bulk.

The first experiment measures the time to create 5000 directories, each containing a zero length file. The results are shown in figure 7. Each new directory requires ZIA to write a new keyfile to the disk, adding an extra disk write to each operation; the write-behind policy of ext2fs keeps these over-

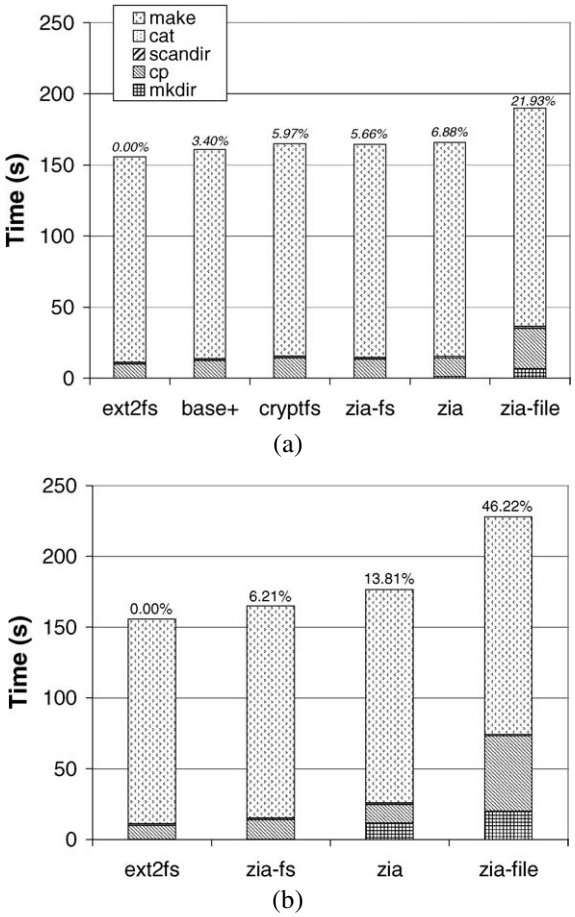


Figure 5. Modified Andrew Benchmark. (a) All file systems and ZIA over 802.11. (b) Ext2fs and ZIA over Bluetooth. This shows the performance of Ext2fs against five stacked file systems using a Modified Andrew Benchmark. (a) All the file systems compared to ZIA running over 802.11; (b) ext2fs compared to ZIA using Bluetooth. When using per-directory or per-mount keys, 802.11 ZIA has an overhead of less than 7% in comparison to an Ext2fs system and performs similarly to a simple single key encryption system, Cryptfs.

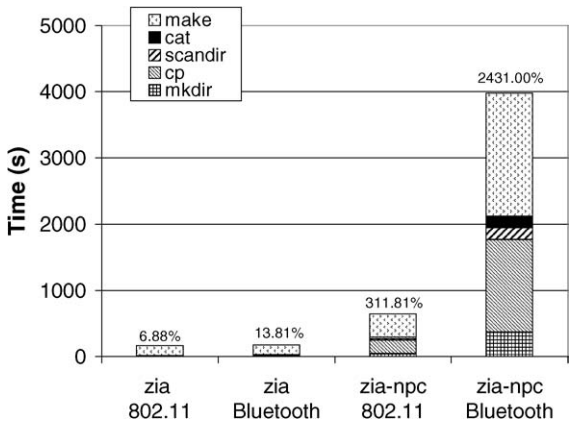


Figure 6. Modified Andrew Benchmark. This figure compares the optimized version of ZIA against ZIA-NPC, which uses no key caching or fresh key prefetching. Both 802.11 and Bluetooth are shown. The percentage overhead is in comparison to the raw file system. Optimizations are crucial to the performance of the system and high Bluetooth latencies exaggerate this effect.

File system	Time (s)	Over Ext2fs
Ext2fs	10.05 (0.01)	–
Base+	10.60 (0.47)	5.5%
Cryptfs	11.06 (0.01)	10.1%
ZIA-FS	11.04 (0.01)	9.9%
ZIA	17.21 (0.36)	71.3%
ZIA-FILE	36.00 (1.47)	258.2%

(a)

File system	Time (s)	Over Ext2fs
Ext2fs	10.05 (0.01)	–
ZIA-FS	12.02 (0.01)	19.6%
ZIA	27.05 (0.01)	169.2%
ZIA-FILE	57.43 (0.60)	471.6%

(b)

Figure 7. Creating directories. (a) 802.11. (b) Bluetooth. This table shows the performance for the creation of 5000 directories, each containing one zero-length file. Standard deviations are shown in parentheses. Although ZIA has a cache of fresh keys for directory creation, it must write those keyfiles to disk.

File system	Time (s)	Over Ext2fs
Ext2fs	2.00 (0.87)	–
Base+	2.89 (0.67)	44.9%
Cryptfs	2.95 (0.53)	47.9%
ZIA-FS	3.10 (0.87)	55.5%
ZIA	51.61 (2.57)	2485.7%
ZIA-FILE	64.91 (3.83)	3151.9%

(a)

File system	Time (s)	Over Ext2fs
Ext2fs	2.00 (0.87)	–
ZIA-FS	3.20 (0.97)	60.1%
ZIA	356.80 (2.73)	17776.3%
ZIA-FILE	369.50 (4.06)	18413.2%

(b)

Figure 8. Scanning directories. (a) 802.11. (b) Bluetooth. This table shows the performance for reading 5000 directories, each containing one zero-length file. Standard deviations are shown in parentheses. In this case, ZIA must synchronously acquire each file key.

heads manageable. In addition, the filenames must be encrypted, accounting for the rest of the overhead. ZIA-FILE additionally writes a key for each newly created file, adding additional overhead.

The next benchmark examines ZIA's overhead for reading 5000 directories and a zero length file in each directory. This stresses keyfile reads and key acquisition. Note that without the empty file ZIA does not need the decrypted key and the token would never be used. We ran a `find` across the 5000 directories and files created during the previous experiment. We flushed the cache between tests to make sure the name cache was not a factor. The results are shown in figure 8.

The results show a large overhead for ZIA. This is not surprising since we have created a file layout with the smallest degree of directory locality possible. ZIA is forced to fetch 5000 keys, one for each directory; there is no locality for key caching to exploit. This inserts a network round trip into reading the contents of each directory, accounting for an extra 10 milliseconds per directory read. An optimization in ZIA-

File system	Time (s)	Over Ext2fs
Ext2fs	21.24 (0.39)	–
Base+	22.61 (0.58)	6.4%
Cryptfs	23.13 (0.21)	8.9%
ZIA-FS	26.99 (0.21)	27.1%
ZIA	28.20 (0.35)	32.8%
ZIA-FILE	53.11 (1.15)	150.0%

(a)

File system	Time (s)	Over Ext2fs
Ext2fs	21.24 (0.39)	–
ZIA-FS	27.54 (0.49)	29.7%
ZIA	31.80 (0.53)	49.7%
ZIA-FILE	149.59 (0.88)	604.3%

(b)

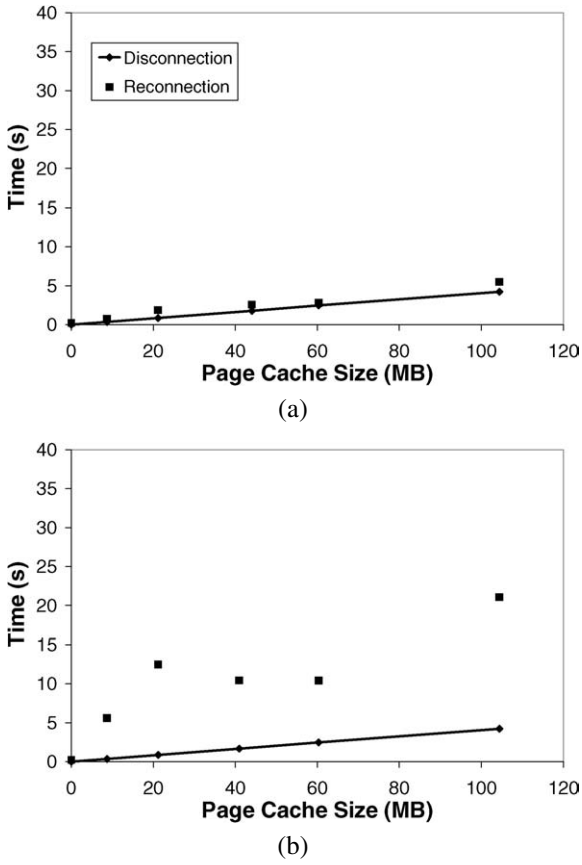
Figure 9. Copying within the file system. (a) 802.11. (b) Bluetooth. This table shows the performance for copying a 66 MB source tree from one directory in the file system to another. Standard deviations are shown in parentheses. Synchronously decrypting and encrypting each file page adds overhead to each page copy. This is true for ZIA as well as Cryptfs.

FILE, reads and prefetches the decrypted key for the zero-length file. This reduces file read time in the common case. However, this benchmark does not need this decrypted key, so it does not need to wait for the response from the token. However, the act of reading this key from disk adds to the overhead shown for ZIA-FILE. ZIA-FS only needs to fetch one key from the token, giving it similar performance to Cryptfs.

Each directory read in ZIA requires a keyfile read and a key acquisition in addition to the work done by the underlying ext2fs. Interestingly, the amount of unmasked acquisition time plus the time to read the keyfile was similar to the measured acquisition costs. To better understand this phenomenon, we instrumented the internals of the directory operations. Surprisingly, the directory read completed in a few tens of microseconds, while the keyfile read was a typical disk access. We believe that this is because, in our benchmark, keyfiles and directory pages are always placed on the same disk track. In this situation, the track buffer will contain the directory page before it is requested.

It is likely that an aged file system would not show such consistent behavior [38]. Nevertheless, we are considering moving keyfiles out of directories and into a separate location in the lower file system. Since keys are small, one could read them in batches, in the hopes of prefetching useful encrypted file keys. When encrypted keys are already in hand, the directory read would no longer be found in the track buffer, and would have to go to disk. However, this time would be overlapped with key acquisition, reducing total overheads.

The final I/O intensive experiment is to copy the Pine 4.21 source tree from one part of the file system to another. The initial files are copied in and then the cache is flushed to avoid caching effects. This measures data intensive operations. The Pine source is 66.1 MB spread across 50 directories. The results are shown in figure 9. In light of the earlier experiments, it is clear why Crypt and ZIA are slow in comparison to Base+ and Ext2fs. Each file page is synchronously decrypted after a `read` and protected before a `write`.



(a)

(b)

Figure 10. 802.11 disconnection and reconnection. (a) ZIA. (b) ZIA-FILE. This plot shows the disconnection encryption time and reconnection decryption time for ZIA and ZIA-FILE over 802.11. The line shows the time required to encrypt all the file pages when the token moves out of range. The blocks show the time required to refetch all the cached keys and decrypt the cached file pages. Per-file keys takes a similar amount of time to secure, but much longer to restore.

4.3. Departure and return

In addition to good performance, ZIA must have two additional properties. For security, all file page data must be encrypted soon after a user departs. To be usable, ZIA should restore the machine to the pre-departure state before the user resumes work. Recall that when the user leaves, the system encrypts the file pages in place. When the user returns, ZIA requests decryption of all keys in the key cache and then decrypts the data in the page cache. To measure both disconnection and reconnection time, we copied several source directories of various sizes into ZIA and ZIA-FILE, removed the token, and then brought it back into range. Figure 10 shows these results for 802.11, and figure 11 shows the results for Bluetooth. The line shows the time required to secure the file system and the points represent the time required to restore it. The right-most points on the graph represent the largest file cache we could produce in our test system; this is roughly half the available memory due to caching of encrypted and decrypted pages in the stacking implementation.

The encryption time depends solely on the amount of data in the page cache. Unsurprisingly, encryption time is linear with page cache size and not dependent on key granularity.

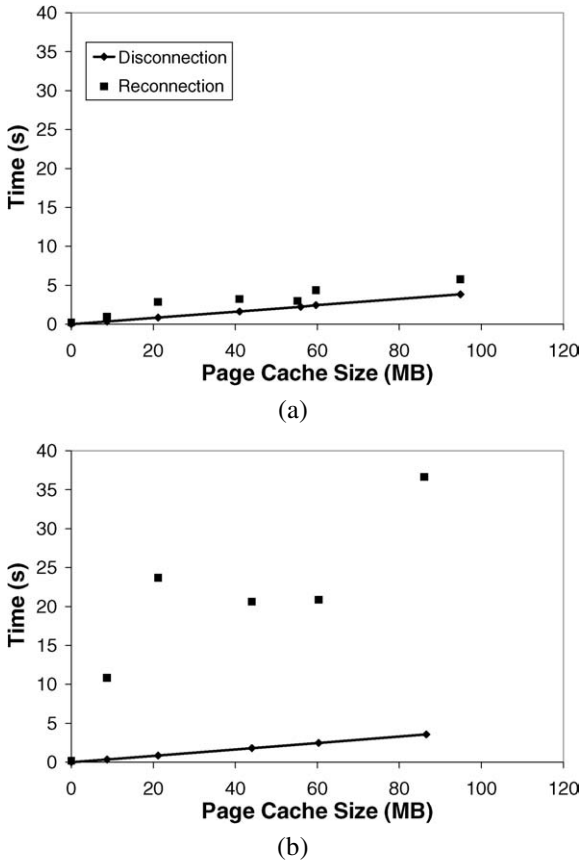


Figure 11. Disconnection and reconnection. (a) ZIA. (b) ZIA-FILE. This plot shows the disconnection encryption time and reconnection decryption time for ZIA and ZIA-FILE over Bluetooth.

Decryption is also linear, though key fetching requires a variable amount of time due to the unknown number of keys in the cache. ZIA-FILE uses a much larger number of keys, slowing its recovery time. We believe that a window of five seconds is too short for a thief to obtain the laptop and examine the contents of the page cache. Furthermore, the user should come back to a system with a warm cache. Once the user is within radio range, he must walk to the laptop, sit down, and resume work; this is likely to be more than six seconds.

4.4. Energy

To gain user acceptance, and minimize users removing the token for charging, the energy budget of the token must be carefully managed. Unfortunately, without building a prototype closer to production size and efficiency, it is difficult to measure the energy requirements of the token. Although the energy density is likely to be similar, the device will be smaller in size than an iPAQ. However, extra functionality on the iPAQ, including a full scale OS and significant flash memory, are extraneous power sinks. A production token may also use a more energy efficient wireless link, as the extra functionality of Bluetooth, and the high power of 802.11, are unnecessary.

In order to obtain an estimate of energy usage, we ran the token unplugged responding to constant polling messages. Both the 802.11 and Bluetooth tests were conducted using the iPAQ external expansion jacket which contains an additional battery. In a single trial, the 802.11 token ran for 3 hours 38 minutes. Using Bluetooth it ran for 11 hours 58 minutes.

5. Related work

To the best of our knowledge, ZIA is the first system to provide encrypted filing services that defend against physical attack while imposing negligible usability and performance burdens on a trusted user. ZIA accomplishes this by separating the *long-term authority* to act on the user's behalf from the entity performing the actions. The actor holds this authority only over the short term, and refreshes it as necessary.

There are a number of file systems that provide transparent encryption for files; the best known is CFS [2]. CFS is built as an indirection layer between applications and an arbitrary underlying file system. This layer is implemented as a "thin" NFS server that composes encryption atop some other, locally-available file system. Keys are assigned on a directory tree basis. These trees are exposed to the user; the secure file system consists of a set of one or more top-level subtrees, each protected by a single key.

When mounting a secure directory tree in CFS, the user must supply the decryption keys via a pass-phrase. These keys remain in force until the user consciously revokes them. This is an explicit design decision, intended to reduce the burden on users of the system. In exchange, the security of the system is weakened by vesting long-term authority with the laptop. CFS also provides for the use of smart cards to provide keys [3], but they too are fetched at mount time rather than periodically. Even if fetched periodically, a user would be tempted to leave the smart card in the machine most of the time.

SC-CFS [22] also employs a smart-card for providing keys, but in a much tighter security framework. It provides a separate key per-file, using the smart-card to generate them. In addition, it changes the key every time the file is written, ensuring confidentiality even when previous keys are exposed. This mechanism creates considerable file system overhead.

Due to their use of an NFS server, both CFS' and SC-CFS' total overhead can be substantial. One way to implement a cryptographic file system more efficiently is to place it in the kernel, avoiding cross-domain copies. TCFS [8] moves the system into the kernel, and addresses some of the ease-of-use issues in CFS. It also provides data integrity checks and group sharing of files. TCFS main advantage over CFS is the placement of encryption and decryption in the client, allowing the thin NFS server to reside in a distributed server.

Many other systems exist for providing secure local and network file storage. Some provide integrity, such as SUNDR [27] and PFS [39] and SFSRO [16], or add confidentiality, such as SFS [26]. Some distributed file systems only protect the communication between clients and servers, such as Echo [1], and not the stored files themselves.

There are also commercial file encryption systems such as Microsoft's Encrypting File System (EFS) [29]. While EFS solves many administrative issues, it is essentially no different from CFS. A single password serves as the key-encrypting key for on-disk, per-file keys. EFS still depends on screen saver or suspension locks to revoke this key-encrypting key, rather than departure of the authorized user. The user may disable the screen saver or suspension locks after finding them intrusive. Anecdotally, we have found that many Windows 2000 laptop users have done exactly that.

The task of kernel integration is simplified by a stackable file system infrastructure [17,35]. Stackable file systems provide the ability to interpose layers below, within, or above existing file systems, enabling incremental construction of services. FiST [42] is a language and associated compiler for constructing portable, stackable file system layers. We use FiST in our own implementation, though our use of the virtual memory and buffer cache mechanisms native to Linux would require effort to port to other operating systems. We have found FiST to be a very useful tool in constructing file system services.

Cryptfs is the most complete prior example of a stacking implementation of encryption. It was first implemented as a custom-built, stacked layer [41], and later built as an example use of FiST. Cryptfs – in both forms – shares many of the goals and shortcomings of CFS. A user supplies his keys only once; thereafter, the file system is empowered to decrypt files on the user's behalf. Cryptfs significantly outperforms CFS, and our benchmarks show Cryptfs in an even better light. This is primarily due to the replacement of Blowfish [37] with Rijndael [13].

Several efforts have used proximity-based hardware tokens to detect the presence of an authorized user. Landwehr [24] proposes disabling hardware access to the keyboard and mouse when the trusted user is away. This system does not fully defend against physical possession attacks, since the contents of disk and possibly memory may be inspected at the attacker's leisure. Similar systems have reached the commercial world. For example, the XyLoc token system [15] emits a unique ID using a wire loop and a battery. Unfortunately the security of this device depends on the difficulty in replicating the device from eavesdropping the ID.

CyberLocator [11] has proposed using location authentication using GPS satellites; their particular interest is in affirming the location of Internet gamblers. Unfortunately, this technique would be unsuitable for hand-held devices and laptops. GPS must have a line-of-sight to geo-synchronous satellites, and laptop users are typically inside. Also the accuracy of such a system is 100 meter in the worst case, far enough for a laptop to be out of sight of the user.

Rather than use passwords or hardware tokens, one could instead use biometrics. Biometric authentication schemes intrude on users in two ways. The first is the false-negative rate: the chance of rejecting of a valid user [34]. For face recognition, this ranges between 10% and 40%, depending on the amount of time between training and using the recognition system. For fingerprints, the false-negative rate can be as high as 44%, depending on the subject. The second intrusion stems from physical constraints. For example, a user must touch a special reader to validate his fingerprint. Such burdens encourage users to disable or work around biometric protection. A notable exception is iris recognition. It can have a low false-negative rate, and can be performed unobtrusively [32]. However, doing so requires three cameras – an expensive and bulky proposition for a laptop.

I/O software [20] is developing a product based on proximity location provided by Bluetooth in conjunction with facial recognition software. Their intention is to use proximity as a cue to indicate the loss of biometric authentication. The combination of proximity sensing with biometrics address the lack of continuous biometric authentication, however the false negative rate will be problematic. The Bluetooth distance measurement system, developed by Bluesoft [5], would require additional mechanisms to provide security.

6. Conclusion

Because laptops are vulnerable to theft, they require additional protection against physical attacks. Without such protection, anyone in possession of a laptop is also in possession of all of its data. Current cryptographic file systems do not offer this protection, because the user grants the file system long-term authority to decrypt on his behalf. Closing this vulnerability with available mechanisms – passwords, secure hardware, or biometrics – would place unpleasant burdens on the user, encouraging him to forfeit security entirely.

This paper presents our solution to this problem: *Transient Authentication*. In it, a user wears an authentication token that retains the long-term authority to act on his behalf. A laptop, connected to the token by a short-range wireless link, obtains this authority only when it is needed. Applying this scheme to a cryptographic file, called ZIA, system imposes an overhead of less than 7% above the local file system for representative workloads; this is indistinguishable from the costs of simple encryption.

If the user leaves, the laptop encrypts any cached file system data. For the largest buffer cache on our hardware, this process takes less than five seconds – less time than would be required for a nearby thief to examine data. Once the user is back in range, the file system is restored to pre-departure state within six seconds. The user never notices a performance loss on return. ZIA thus prevents physical possession attacks without imposing any performance or usability burden.

We are currently extending ZIA's model to system services and applications [10]. By protecting application state and access to sensitive services, ZIA can protect the entire machine – not just the file system – from attack.

Acknowledgements

The authors wish to thank Peter Chen, who suggested the recovery time metric, and Peter Honeyman, for many valuable

conversations about this work. Mary Baker, Landon Cox, Jason Flinn, Minkyong Kim, Sam King, and James Mickens provided helpful feedback on earlier drafts.

This work is supported in part by the Intel Corporation; Novell, Inc.; the National Science Foundation under grant CCR-0208740; and the Defense Advanced Projects Agency (DARPA) and Air Force Materiel Command, USAF, under agreement number F30602-00-2-0508. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Intel Corporation; Novell, Inc.; the National Science Foundation; the Defense Advanced Research Projects Agency (DARPA); the Air Force Research Laboratory; or the U.S. Government.

References

[1] A. Birrell, A. Hisgen, C. Jerian, T. Mann and G. Swart, The echo distributed file system, Technical Report No. 111, Digital Equipment Corp. Systems Research Center (September 1993).

[2] M. Blaze, A cryptographic file system for UNIX, in: *Proceedings of the 1st ACM Conference on Computer and Communications Security*, Fairfax, VA (November 1993) pp. 9–16.

[3] M. Blaze, Key management in an encrypting file system, in: *Proceedings of the Summer 1994 USENIX Conference*, Boston, MA (June 1994) pp. 27–35.

[4] M. Blaze, J. Feigenbaum and J. Lacy, Decentralized trust management, in: *IEEE Symposium on Security and Privacy*, Oakland, CA (1996) pp. 164–173.

[5] Bluesoft Inc., San Mateo, CA.

[6] S. Brands and D. Chaum, Distance-bounding protocols, in: *Proceedings of EUROCRYPT'93*, Lecture Notes in Computer Science, Vol. 765 (Springer, 1993) pp. 344–359.

[7] M. Burrows, M. Abadi and R. Needham, A logic of authentication, ACM Transactions on Computer Systems 8(1) (1990) 18–36.

[8] G. Cattaneo, L. Catuogno, A.D. Sorbo and P. Persiano, The design and implementation of a transparent cryptographic file system for Unix, in: *Proceedings of the Freenix Track: 2001 USENIX Annual Technical Conference*, Boston, MA (June 2001) pp. 199–212.

[9] P.C. Clark and L.J. Hoffman, BITS: A Smartcard protected operating system, Communications of the ACM 37(11) (1994) 66–70.

[10] M. Corner and B.D. Noble, Protecting applications with transient authentication, in: *Proceedings of the 1st International Conference on Mobile Systems, Applications, and Services (MobiSys'03)*, San Francisco, CA (May 2003), to appear.

[11] CyberLocator Inc., Boulder, CO.

[12] J. Daemen, Cipher and hash function design: strategies based on linear and differential cryptanalysis, Ph.D. Thesis, Katholieke Universiteit Leuven (March 1995).

[13] J. Daemen and V. Rijmen, AES proposal: Rijndael, Advanced Encryption Standard Submission, 2nd version (March 1999).

[14] W. Diffie, P. van Oorschot and M. Wiener, *Design Codes and Cryptograhpy* (Kluwer Academic, 1992).

[15] Ensure Technologies, Ann Arbor, MI.

[16] K. Fu, M.F. Kaashoek and D. Mazieres, Fast and secure distributed read-only file system, in: *Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2000)*, San Diego, CA (October 2000) pp. 181–196.

[17] J.S. Heidmann and G.J. Popek, File-system development with stackable layers, ACM Transactions on Computer Systems 12(1) (1994) 58–89.

[18] J.H. Howard, M.L. Kazar, S.G. Menees, D.A. Nichols, M. Satyanarayanan, R.N. Sidebotham and M.J. West, Scale and performance in a distributed file system, ACM Transactions on Computer Systems 6(1) (1988) 51–81.

[19] Y. Hu, A. Perrig and D.B. Johnson, Wormhole detection in wireless ad hoc networks, Technical Report, Rice University Department of Computer Science (June 2002).

[20] I/O Software Inc., Riverside, CA.

[21] N. Itoi, W.A. Arbaugh, S.J. Pollack and D.M. Reeves, Personal secure booting, in: *Proceedings of ACISP 2001*, Syndney, Australia (July 2001).

[22] N. Itoi, P. Honeyman and J. Rees, SCFS: A Unix filesystem for smartcards, in: *Proceedings of USENIX Workshop on Smartcard Technology*, Chicago, IL (May 1990).

[23] S.R. Kleiman, Vnodes: An architecture for multiple file system types in Sun UNIX, in: *Proceedings of USENIX Association Summer Conference*, Atlanta, GA (June 1986) pp. 238–247.

[24] C.E. Landwehr, Protecting unattended computers without software, in: *Proceedings of the 13th Annual Computer Security Applications Conference*, San Diego, CA (December 1997) pp. 274–283.

[25] P. MacKenzie and M.K. Reiter, Networked cryptographic devices resilient to capture, in: *Proceedings of 2001 IEEE Symposium on Security and Privacy*, Oakland, CA (May 2001) pp. 12–25.

[26] D. Mazieres, M. Kaminsky, M.F. Kaashoek and E. Witchel, Separating key management from file system security, in: *Proceedings of Symposium on Operating Systems Principles* (1999) pp. 124–139.

[27] D. Mazières and D. Shasha, Don't trust your file server, in: *Proceedings of the 8th Workshop on Hot Topics in Operating Systems* (May 2001) pp. 113–118.

[28] M.K. McKusick, W.N. Joy, S.J. Leffler and R.S. Fabry, A fast file system for Unix, Computer Systems 2(3) (1984) 181–197.

[29] Microsoft Corporation, Encrypting file system for Windows 2000, Unpublished Whitepaper, Redmond, WA.

[30] C. Narayanaswami and M.T. Raghunath, Application design for a smart watch with a high resolution display, in: *Proceedings of the 4th International Symposium on Wearable Computers*, Atlanta, GA (October 2000) pp. 7–14.

[31] R.M. Needham and M.D. Schroeder, Using encryption for authentication in large networks of computers, Communications of the ACM 21(12) (1978) 993–999.

[32] M. Negin, T.A. Chemielewski Jr., M. Salganicoff, T.A. Camus, U.M. Cahn von Seelen, P.L. Venetianer and G.G. Zhang, An iris biometric system for public and personal use, IEEE Computer 33(2) (2000) 70–75.

[33] National Institute of Standards and Technology, Computer data authentication, FIPS Publication No. 113 (May 1985).

[34] P.J. Phillips, A. Martin, C.L. Wilson and M. Przybocki, An introduction to evaluating biometric systems, IEEE Computer 33(2) (2000) 56–63.

[35] D. Rosenthal, Evolving the vnode interface, in: *Proceedings of USENIX Association Conference* (June 1990) pp. 107–118.

[36] M. Satyanarayanan, Integrating security in a large distributed system, ACM Transactions on Computer Systems 7(3) (1989) 247–80.

[37] B. Schneier and R. Anderson, Description of a new variable-length key, 64-bit block cipher (Blowfish), in: *Proceedings of Fast Software Encryption, Cambridge Security Workshop*, Cambridge, UK (December 1993) pp. 191–204.

[38] K.A. Smith and M.I. Seltzer, File system aging – increasing the relevance of file system benchmarks, in: *Proceedings of ACM International Conference on Measurement and Modeling of Computer Systems*, Seattle, WA (June 1997) pp. 203–213.

[39] C.A. Stein, J.H. Howard and M.I. Seltzer, Unifying file system protection, in: *Proceedings of the 2001 USENIX Annual Technical Conference* (USENIX Association, 2001).

[40] B. Yee and J.D. Tygar, Secure coprocessors in electronic commerce applications, in: *Proceedings of the 1st USENIX Workship of Electronic Commerce*, New York (July 1995) pp. 155–170.

[41] E. Zadok, I. Badulescu and A. Shender, Cryptfs: A stackable vnode level encryption file system, Technical Report CUCS-021-98, Computer Science Department, Columbia University (1998).

[42] E. Zadok and J. Nieh, FiST: a language for stackable file systems, in: *Proceedings of the 2000 USENIX Annual Technical Conference*, San Diego, CA (June 2000) pp. 55–70.

Mark D. Corner is currently finishing his Ph.D. in electrical engineering systems at the University of Michigan. He received the B.S. and M.S. degrees in electrical engineering from the University of Virginia. His interests include mobile computing, operating systems, wireless networking, file systems, and security. Mr. Corner is a member of Eta Kappa Nu and Tau Beta Pi.

Brian Noble is the Morris Wellman Faculty Development Assistant Professor in the Electrical Engineering and Computer Science Department at the University of Michigan. His research centers on software supporting mobile devices and their users. He completed the Ph.D. in computer science at Carnegie Mellon University in 1998, and is a recipient of the NSF CAREER award.
E-mail: bnable@umich.edu