Mobile Device Security Using Transient Authentication

Anthony J. Nicholson, Mark D. Corner, and Brian D. Noble

Abstract—Mobile devices are vulnerable to theft and loss due to their small size and the characteristics of their common usage environment. Since they allow users to work while away from their desk, they are most useful in public locations and while traveling. Unfortunately, this is also where they are most at risk. Existing schemes for securing data either do not protect the device after it is stolen or require bothersome reauthentication. *Transient Authentication* lifts the burden of authentication from the user by use of a wearable token that constantly attests to the user's presence. When the user departs, the token and device lose contact and the device secures itself. We show how to leverage this authentication framework to secure all the memory and storage locations on a device into which secrets may creep. Our evaluation shows this is done without inconveniencing the user, while imposing a minimal performance overhead.

Index Terms—Transient authentication, human factors, cryptographic controls, security, mobile computing, privacy.

1 Introduction

Powerful and affordable mobile devices have brought users an unprecedented level of convenience and flexibility. Laptops and PDAs let users work anywhere, anytime. Unfortunately, physical security is a major problem for these devices. To be portable, they must be lightweight and small-sized. Since they are designed for mobile use, they are often exposed in public places such as airports, coffee houses, and taxis, where they are vulnerable to theft or loss.

Along with the value of lost hardware, users must worry about the exposure of sensitive information. People store vast amounts of personal data on their mobile devices and the loss of a device may lead to the exposure of credit card numbers, passwords, client data, and military secrets [3], [32].

Mobile devices often protect sensitive data using encryption, but the challenge in device security is not encrypting data but *authenticating* the current user [30]. A device must obtain proof of the user's identity and authority before granting access to data. This proof could take the form of a password, a smart card inserted into a reader, or biometric data from a fingerprint or iris scanner [8]. Unfortunately, these forms of authentication are *infrequent* and *persistent*. Should a device subsequently fall into the wrong hands, an attacker could act as the real user, subverting encryption for the duration that this authentication holds.

- *A.J. Nicholson and B.D. Noble are with the Department of Electrical Engineering and Computer Science, University of Michigan, 2260 Hayward, Ann Arbor, MI 48109-2121. E-mail: {tonynich, bnoble}@eecs.umich.edu.*
- *M.D. Corner is with the Department of Computer Science, University of Massachusetts, Amherst, MA 01003-4610. E-mail: mcorner@cs.umass.edu.*

Manuscript received 4 Mar. 2005; revised 4 Oct. 2005; accepted 1 Dec. 2005; published online 15 Sept. 2006.
For information on obtaining reprints of this article, please send e-mail to: tmc@computer.org, and reference IEEECS Log Number TMC-0050-0305.

But, how often must a user authenticate herself? One might require users to reauthenticate each time the device performed any operation on sensitive data. This would quickly render the system unusable and many users would disable the authentication system out of annoyance [1], [2], [14]. Another mechanism would require the user to "unlock" the device once at boot. This would enhance the user experience but leave data vulnerable if the device were lost or stolen. These two models highlight an inherent tension between security and usability. While data should only be accessible when its authorized user is present, it is obtrusive to continually ask for proof.

Our new model, *Transient Authentication*, resolves this tension. Users wear a small token (e.g., IBM Linux wristwatch [22]) that has a short-range wireless link and modest computational resources. It constantly authenticates to devices on behalf of the user. The limited radio range (several meters) serves as a proximity cue, letting a device take steps to protect its data when the user leaves the physical area. Since users wear the token, it is far less likely to be misplaced or stolen than is a laptop or PDA. A wearable token is also physically bound to a specific user.

Armed with this authentication data from the token, devices protect data when the user departs by encrypting, overwriting, and/or flushing it. Cryptographic file systems secure data in persistent storage, but the unique characteristics of mobile devices make protecting data in other memory locations critical as well. Batteries and wireless network links allow devices to continue running while traveling and in public places. This is precisely where they are most vulnerable to loss or theft. While a device is running, secrets may be present in RAM, the swap partition, CPU registers, and the various buffers and caches of peripheral devices.

We show how to leverage this authentication data to protect all the memory locations where secrets may lie. Data in permanent storage is encrypted in-place by our cryptographic file system, ZIA [10]. When the user departs, the

execution of all running processes is frozen and their resident memory pages are encrypted in-place [11]. We secure other memory and storage, like the swap partition, CPU registers, and various buffers and caches of peripheral devices, by flushing, overwriting, or encrypting data as appropriate. The user's token retains all encryption keys so that the device is unusable while the user is absent.

Some processes can safely continue while the user is absent, either because they do not handle sensitive data or because they secure their secrets themselves. Such applications use our *application-aware protection* API to access authentication data and token services. We modified the Mozilla Web browser to utilize this API, allowing it to protect secrets such as user passwords, SSL keys [31], [33], and cookies, without noticeable degradation of runtime performance.

2 Transient Authentication

2.1 Principles

Transient Authentication is founded on the following four design principles:

1. **Tie Capabilities to Users**. The mobile device should only perform sensitive operations when the user is present. Thus, all encryption keys must reside solely on the user's wearable token, which is in her possession at all times and is therefore far less likely to be stolen or misplaced. For performance, credentials may be temporarily cached at the mobile device, but must be discarded whenever the token is not present.
2. **Do No Harm**. Users quickly disable inconvenient security mechanisms [1], [2]. But, anecdotal evidence shows that users are willing to infrequently reenter passwords. Transient Authentication requires user participation that is no more burdensome. Users will also quickly disable our system if they notice poor performance. To ensure adoption, the additional overhead of key authentication, communication, and data encryption must not be excessive.
3. **Secure and Restore on People Time**. When the user departs, the device must secure itself before an attacker would have the chance to physically extract any information. This time window is on the order of seconds, not milli or microseconds. Conversely, when a user walks back to use the device, the token will regain wireless contact while she is still meters away. This gives the system several seconds to restore the device's state.
4. **Ensure Explicit Consent**. The device must not take any sensitive action without the user's consent. Transient Authentication must ensure that both 1) the user's device is indeed talking to her token and 2) her token is not communicating with any other devices without her knowledge. To do so, users explicitly *bind* tokens to devices through an exchange of public keys that establishes a pairwise trust relationship. To limit the consequences of token loss, users authenticate themselves to their token daily.

2.2 Trust and Threat Model

Our threat model focuses on defending against attacks that require physical proximity or possession—for instance, exploiting cached passwords or credentials present on the device. Even with a cryptographic file system, an attacker can freely examine the contents of the file system if the system cached the master password since the user last entered it. Physical access also permits console-based attacks, which result in root access to the machine.

Our model considers attacks made possible by long-term physical-possession, such as inspecting the memory of a running system via hardware probing or operating system interfaces. We are also concerned with attacks involving the communication medium between the device and the wireless token. An adversary might eavesdrop on this channel and attempt to later impersonate the token. An even more sophisticated attacker would record token-device communications and subsequently steal the laptop with the hope of finding keys to decrypt the recorded traffic. It is possible to defeat these attacks by leveraging the large body of existing work [16], [29].

The security of our design is predicated, however, on the limited, deterministic range of the token's radio. An attacker could use repeaters to fool the token and device into thinking they are still in proximity (a *wormhole attack*). Timing information [6], [19] and use of directional antennae [18] have been suggested as ways to determine token distance from the device, but no techniques currently exist that provably defeat wormhole attacks in real-world situations.

There are several security threats that Transient Authentication does not address. We do not handle trusted users who maliciously leak sensitive data or network-based exploits, such as buffer-overflow attacks or other network security weaknesses. We also cannot deal with denial-of-service attacks that jam the spectrum used for token-device communication or forcible theft of the token and device by an attacker.

2.3 Authentication Architecture

The link between the user's mobile device and her wearable token is the heart of our authentication architecture. Fig. 1 gives a high-level overview of this relationship. In addition to vouching for the user's presence, this link allows the token to help the device with key management. Applications also use the token to encrypt and decrypt small amounts of data directly.

2.3.1 Token System

The token runs an authentication server process and the device runs an authentication client. These two processes communicate via a short-range wireless link (e.g., Bluetooth), as illustrated in Fig. 1. This communication is protected by a session key established via the Station-to-Station protocol, which provides perfect forward security [16].

The token exchanges nonces with the device once per second to indicate that the user is still present; the nonces prevent replay attacks [7]. When the authentication client on the device does not receive an expected nonce, it declares the user absent and secures the various memory spaces on the device. When the user returns, the authentication client

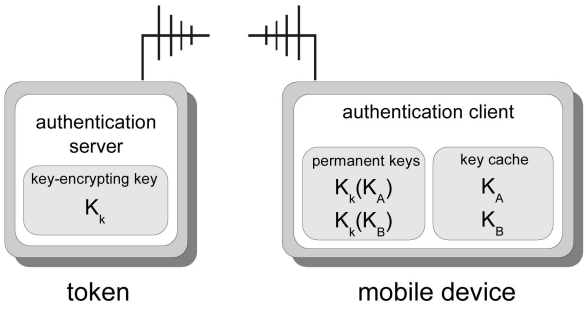

Fig. 1. **Token-Device Communication Architecture**. The wearable token and mobile device communicate via a wireless link. An authentication server runs on the token, polling the device once per second. The authentication server also provides access to the key-encrypting-key, K_k. The device's authentication client listens for the token's poll. All encryption keys used by the mobile device are themselves secured by the key-encrypting-key, but may be cached in the clear for performance. The key cache is flushed when contact with the token is lost.

detects the heartbeat signal again, and restores the system to its original state. Section 4 describes this process.

Since the security of our architecture requires that the device and token lose contact when the user is far enough away from her device that it is vulnerable, the radio on the token must have a narrowly defined range. If this range were too short, our security mechanisms would be invoked far too often (e.g., when the user turned around in her desk chair). Conversely, if the range is too long, a user could be across the room but still in contact, leaving the device open to attack.

The connection between the token and the device must be broken once the user is more than a few meters away. This imposes an upper-bound on how quickly the system must secure and restore state. When the user departs, the device has several seconds in which to secure data before an attacker could be expected to seize control. Conversely, when the user returns, contact will be reestablished when she is a few meters away. It will take several seconds for her to be at the device ready to resume work. If the system can restore the device to its original state in that time, the entire process will be seamless from the user's perspective.

2.3.2 Key Management

Our authentication framework tells a mobile device when its user departs so it may secure data via encryption. To recover data when the user returns, the device must have access to the appropriate decryption keys. However, these keys cannot exist in the clear on the device while the user is gone, or else an attacker could decrypt any data he wishes. Likewise, the device cannot retrieve each possible key, else recovery would take too long.

Therefore, the token secures encryption keys on the mobile device's behalf. The user creates a key-encrypting-key (denoted by K_k) that resides solely on the token. All other encryption keys are stored on the device encrypted with K_k. The mobile device ships encrypted keys to the token on demand for decryption.

Synchronously decrypting keys on the token before every use would add the token-device communication latency to every secure operation, degrading performance unacceptably. Furthermore, constant communication between device and token is wasteful of the wireless link, reducing battery lifetime. Therefore, the device caches decrypted keys but discards them when the token departs.

2.3.3 Token Authentication and Binding

It may seem that Transient Authentication merely shifts the authentication problem from mobile devices to the token. We argue that wearable tokens have better physical security than laptops or PDAs, precisely because they are worn on one's person. Unfortunately, a user may still lose their token. We must handle this contingency since the user's token holds her key-encrypting-key, which can decrypt all local encryption keys on her device.

The key-encrypting-key must at least be protected by a password or PIN, if not by tamper-resistant hardware. But, how often must the user enter the PIN/password? We argue that requiring the user authenticate themselves to the token once per day (by entering the PIN/password) is no more intrusive to the user than forcing them to unlock their office door every morning.

Thus, the user's laptop is only useful to an attacker for a limited time. Once the reauthentication period expires, the token will refuse to process any requests. At this point, all memory and storage spaces on the user's mobile device are encrypted with their various keys. Those keys are only on the device in encrypted form (secured by K_k). The key-encrypting-key K_k is on the token, which the attacker possesses, but is itself unreadable without the password or PIN.

But, how does a device know which token it is using, and vice versa? Consider the scenario where an attacker has a stolen laptop, but no token. He could identify another user whose token can access the stolen laptop (the victim's coworker, for instance). The attacker could then sit nearby and communicate with the coworker's token in order to decrypt the contents of the stolen laptop. This sort of *tailgating attack* would happen without the legitimate user ever knowing.

Requiring *bindings* between tokens and devices prevents this sort of attack. Once a user authenticates to her token, she also *binds* her token to any devices she wants to use. A user may access several devices through her one token and several tokens may be bound to one laptop. Fig. 2 illustrates the authentication and binding process.

3 Design of Operating System Mechanisms

When the device loses contact with a user's token, it must secure the user's data. This data appears in a multitude of locations, such as disks and memories. With varying levels of difficulty and reward, it is possible to extract information from these spaces, gaining access to the user's personal data. Carefully controlling information that flows into these spaces, encrypting data, or simply flushing and/or overwriting memory locations thwarts the attacker.

3.1 Techniques for Securing State

There are many possible ways to secure data, each of which is appropriate in certain circumstances. Transient Authentication employs three such techniques.

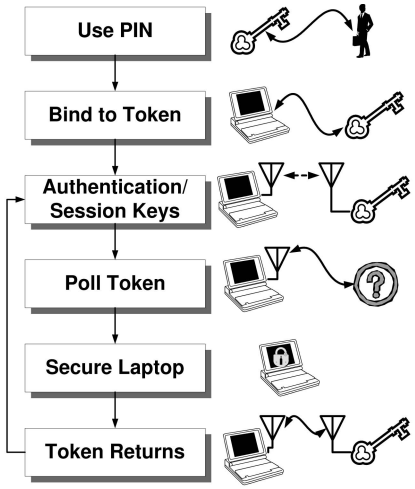

Fig. 2. **Token-Device Authentication and Binding**. Users periodically authenticate themselves to tokens through use of a PIN. Tokens bind to devices by negotiating a session key via public-key encryption techniques. Once a token is bound to a device, it performs key decryption on its behalf.

Some types of data are inherently ephemeral. For example, the contents of registers and the TLB are quickly overwritten after context switches by the values used by the new process. Thus, in certain situations, we rely on the operating system to overwrite and later restore data for free.

Operating systems rely on cache hierarchies in order to maintain performance when addressing large amounts of data. For instance, the kernel uses memory as a cache of disk blocks. Each piece of data need only be secured once. It is often preferable to flush cached data to its backing memory, which the system secures separately. The resulting economy of mechanism simplifies the construction and verification of the system.

Transient Authentication also secures some data in-place. This provides complete transparency to applications, but at the cost of cryptographic overhead to both encrypt and later decrypt the data.

There are pros and cons to each of these techniques. Overwriting would seem fastest, but is not an option when data contents must be preserved. Flushing may in fact be slower than encrypting it in-place if the backing store is much slower (e.g., disks). The overhead of encryption is unnecessary if the data can be safely overwritten. Using encryption also necessitates use of cryptographic keys with all their attendant management complexity and overhead.

3.2 Securing State

The first step in securing a device's data is identifying the different locations where information may reside. This is no easy task, as a user's data tends to permeate the entire hardware/software architecture. While some of it is stored ephemerally, much of it persists for long periods of time.

Consider the receipt of an e-mail on a laptop. A network interface card receives the e-mail text and stores it temporarily in a receive buffer. The NIC transfers the e-mail to RAM using DMA and signals an interrupt. The CPU handles the interrupt by trapping to the OS, which moves the data through the TCP stack via several kernel buffers. During this process, the CPU's cache contains e-mail data, as may several chipset components on the motherboard. The e-mail is eventually passed up to the application layer, where it is interpreted by a mail client, rendered by the X server, and passed to the video card, which may do some additional buffering for performance.

After each of these transfers, unencrypted pieces of the e-mail may be left in memory buffers and caches. It is crucial that all of this data be protected when the user is away from the device. Delineating the abstraction boundaries between all of these data locations often is difficult. The remainder of this section gives an overview of the different memory and storage spaces one must consider when securing the data on a mobile device.

3.2.1 Persistent Storage

Users rely on hard disks, flash memory, and other persistent media to ensure the durability of their data. This permanence also extends to any secrets that have found their way onto disk. Merely deleting data is not enough since well-known techniques exist for retrieving deleted data from magnetic media [17]. Worse, there is nowhere to flush it to since permanent storage is often the base of the caching hierarchy.

In this case, encrypting data in-place is best. Device access latency is on the order of ms, while cryptographic operations typically take orders of magnitude less time. Device access latencies will therefore mask the overhead required to encrypt or decrypt on the fly during disk I/O.

3.2.2 Virtual Memory

A device's virtual memory consists of several distinct address spaces. Each address space contains memory pages, some of which reside in main memory (in RAM), some in the swap partition, and some memory mapped to files in persistent storage. Securing the device's virtual memory means ensuring secrets are not exposed via any of these three spaces.

Without explicit hardware support, encrypting the contents of physical memory in-place is infeasible, requiring a trap to the kernel on every access. One could flush physical memory to disk when the token departs and then overwrite the RAM. This would add many seconds to the securing process since hundreds or thousands of megabytes would need to be written to disk. Instead, the system encrypts contents of RAM in-place when the token departs. This is faster than writing to disk since access latencies to RAM are orders of magnitude smaller than those to disk.

The system also secures memory pages backed by disk files. Since our system secures the disk with a cryptographic file system, securing file-backed pages is trivial. The only unsecured data are the dirty pages and a simple page flush to disk suffices. The key assumption is that dirty pages constitute a small fraction of total physical memory pages and, so, flushing their contents to disk will not significantly increase the time to secure the mobile device.

The remaining unsecured pages reside in the swap space. Transient Authentication cannot simply rely on file system encryption to secure these pages because most operating systems bypass the overhead of file systems and perform swap I/O directly on a raw partition. Thus, the system encrypts pages during swap-out and decrypts them on demand as they are paged back in. As shown in our evaluation, cryptographic overhead is masked by the disk access time.

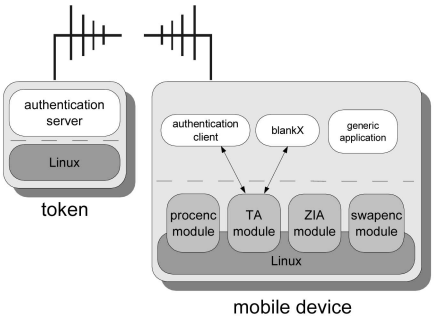

Fig. 3. **Implementation**. The authentication server on the token and the authentication client and video blanking process are user-level applications. Kernel modules interface with the token (TA module), secure processes RAM (procenc module), encrypt persistent data (ZIA module), and secure swap space (swapenc module).

3.2.3 CPU and Chipset Registers and Caches

Any operation that reads or writes main memory will leave traces of that data in the registers and caches of the CPU and chipset. It is difficult to detect what information is left there or to access that state directly; however, these memory locations are commonly erased on every context switch. When the authentication state changes, the OS scheduler implicitly flushes and restores registers and the TLB as part of the context switch.

3.2.4 Peripherals

Many peripheral devices, such as network or video cards, and USB devices use unsecured buffers for incoming and outgoing data. As this internal state is not designed to be modified externally, it may be difficult to access, precluding us from overwriting or encrypting it in-place. Often, end-to-end encryption is sufficient to ensure no secrets are ever exposed in these buffers. For example, use of end-to-end encryption keeps secrets out of the NIC's buffers.

3.2.5 Displays

The display is one of the most vulnerable data space on a mobile device as an attacker can view its contents with no effort. Cleaning a video card's buffers requires overwriting the data, thus blanking the screen. When the user returns, the display's window manager "redraws" the screen. As the data needed to perform the redraw comes from secured areas of virtual memory, the only additional performance penalty is rerendering the contents of the screen, a trivial operation.

4 Securing Machine State

This section describes how our token-device authentication architecture is applied to various memory and storage spaces in a mobile device. These spaces include the local file system, virtual memory, registers and caches, peripherals, and the display. Fig. 3 gives a high-level view of our implementation.

4.1 Securing File Systems

Cryptographic file systems are not new and existing systems [4], [5], [9], [34] illustrate the feasibility and value of encrypting the user's file system. What is lacking is a synergy of authentication and encryption. The security of all these systems is ultimately bounded by the frequency with which the user enters a password. Once the password has been successfully entered, the file system is unlocked until the next authentication timeout. Some systems bound this vulnerability by forcing periodic reauthentication, but forcing the user to repeatedly produce a password on demand leads to annoyance and eventual disabling of security measures [1], [2], [14].

In response to these authentication problems, we designed ZIAfs (Zero-interaction File System) [10], a cryptographic file system. ZIA provides effective file encryption by leveraging our Transient Authentication framework, without adversely impacting performance or usability.

Apart from how ZIAfs handles file keys, its design is similar to that of most cryptographic file systems. Each on-disk object is encrypted by a symmetric key. Keys are stored in the same directory as the files they secure, but are themselves stored encrypted by the key-encrypting-key K_k. For example, if a file was secured by a key K_e, then the keyfile stored on disk with the file contains $K_k(K_e)$. When ZIAfs needs to access the file, it ships the keyfile $K_k(K_e)$ to the token, which performs decryption and returns K_e to the token. This precludes physical-possession attacks since the file keys are not present anywhere on the device except in the form encrypted by K_k.

The roundtrip key transfer from device to token and back necessarily imposes significant latency. Requiring the file system to synchronously contact the token on every file access would result in unacceptable performance. This constant use of the wireless link would also drain the precious battery life of both the token and mobile device. Instead, ZIAfs maintains a local cache of decrypted file keys on the device. When the device loses contact with the token, this cache is flushed and must be repopulated on demand when the user returns. This improves performance while still guaranteeing file keys are never exposed when the user is absent.

The granularity of file keys also impacts performance and usability. Hard drive storage capacities have been increasing into the hundreds or thousands of gigabytes. Assigning each file or chunk on disk its own symmetric key leads to a massive key explosion. The larger the key granularity, the greater opportunities for key caching and reuse. But, as each key covers more files the consequences of key exposure compound. In the extreme, tens or hundreds of gigabytes of data would need to be rekeyed every time a key was rotated or compromised.

In light of this trade-off, ZIAfs uses per-directory keys. Based on the principle of locality-of-reference, this provides opportunities for caching and reuse. The first-time cost to acquire the file key is amortized across all subsequent accesses to files in the same directory. Per-file keys would have led to far fewer hits in the key cache.

We made one further refinement to this design to handle file permissions in a fashion analogous to the UNIX {user, group, world} model. Each directory contains a keyfile containing two encrypted copies of the directory's file key K_e. These are $K_u(K_e)$ and $K_g(K_e)$, corresponding to the directory's owner and group, respectively. World keys

would be implemented similarly in a straightforward fashion, but their implementation was deferred. This modifies the traditional UNIX file access model in a similar fashion as does AFS (the Andrew File System) in that access control is maintained on a per-directory basis. Anecdotal evidence shows files in the same directory are overwhelmingly all accessed by the same person or same group of people supporting this decision.

4.2 Physical Memory

We must also secure main memory when the user is absent. Unlike data on disk, data in main memory cannot be stored encrypted. To do so would require the OS to interpose on each access, adding cryptographic overhead and degrading usability. One option would be to simply flush the contents of memory to the encrypted disk and overwrite the memory (much like hibernating a laptop). This would be secure, but at the cost of disk I/O both to secure and restore the system. Given common disk bandwidth and latency, swapping several hundred megabytes of RAM out to disk would add many seconds to the securing operation. This increases the window of opportunity for any attacker. Conversely, the user would likely have to wait for the machine to "unhibernate" when she returns, violating our principles of "do no harm" and "secure and restore on people time." Instead, Transient Authentication encrypts the contents of memory in-place. When the system detects loss of contact with the token, it freezes the execution of all running processes and encrypts main memory in-place by one global key, K_{mem}. Much like file encryption keys, the system permanently stores this key encrypted by the key-encrypting-key K_k and it caches the key in the clear while the token is present. The cached copy is then discarded [26], [27] when the token departs.

When the user returns, the device will detect the return of the token's heartbeat. At this point, it needs to unencrypt the main memory, so the frozen processes can resume. To do so, it ships the encrypted key $K_k(K_{mem})$ to the token, which uses the key-encrypting-key to return K_{mem} to the device. After unencrypting RAM and restarting the frozen processes, execution resumes normally.

It is not possible to actually freeze all the processes on the mobile device when contact with the token is lost. A bare minimum must keep running to allow the system to later recognize the token's return and recover. Namely, the kernel and the authentication client must remain running while the user is absent, which precludes encrypting their physical memory. Note that the kernel's free page pool and dirty kernel buffers are cleared or flushed to disk at this time. This subset of running processes must still take care to secure any secrets resident in their memory pages since their RAM will remain in the clear while the user is absent.

4.3 Swap Space

While the machine is running, it keeps the contents of main memory in the clear. This preserves application performance since it does not trigger faults on each memory access. While most data on disk is secured by the ZIAfs cryptographic file system, operating systems typically use a raw disk partition for the swap file to maximize I/O performance. In order to prevent unencrypted data from finding its way onto the disk, there are two options—either use an encrypted file to store swap pages or interpose on swap I/O to perform whole-page encryption.

The first option is simpler to implement—we can reuse ZIAfs. This would require the operating system to use a regular file as the swap space, rather than a raw partition, which would impose a noticeable performance penalty. Instead, we added a kernel component that encrypts all writes to the swap file or partition and decrypts on swap ins before unlocking the memory page for access.

In the same way the existing TA implementation uses one global key for securing the whole of main memory, the kernel uses one global key for the entire swap area. This key is not generated by the mobile device itself, but rather fetched from the token upon system initialization. The key is cached during normal operation for performance reasons. When the user (and token) departs, the swap encryption logic need only discard the cached encryption key.

Following the lead of related work [25], the kernel performs whole-page encryption just before a page is written to swap and decrypts the entire page just after it has been paged back in. By locking the pages appropriately, no encrypted data is ever accessed by a user process.

The kernel and the authentication client must keep running while the token is absent. Since the device has discarded the encryption key, swap page encryption/decryption is impossible. Those critical processes still running may have pages in the swap, however. There are three options for handling this situation: 1) Always pin all pages belonging to critical processes in memory, 2) block on any swap requests generated while the token is absent, or 3) never encrypt the swap pages of these critical processes.

The third option is preferable for several reasons. It fits with the design of Transient Authentication, and since it never encrypts the physical memory pages of critical processes, there is no reason to encrypt swap pages. It also ensures correctness of the system. On a system with a small amount of RAM, one of the critical processes may need to access pages in the swap file while the token is absent. Simply blocking on this request will cause deadlock because the daemon that needs to be listening for the token's return is blocked waiting for a key that the token provides. The first option (pinning all pages belonging to critical processes in memory) would prevent any information leakage to the swap file and allow continued execution of critical tasks while the token was absent. This may impose unreasonable constraints on the virtual memory system, especially for systems with limited RAM (precisely our target audience of small, mobile devices).

The kernel therefore tracks the subset of processes that are never frozen and allows their data to pass in and out of the swap file unencrypted. The process is illustrated in Fig. 4. Based on the page's owner process, its contents are either encrypted before being written or allowed to pass into the swapfile unmodified. The kernel also uses a flag in the structure that tracks that state of all pages residing in swap to indicate the encrypted status of each page.

Only critical processes execute while the token is absent. The kernel can therefore service page faults without the encryption key since the only running processes that may

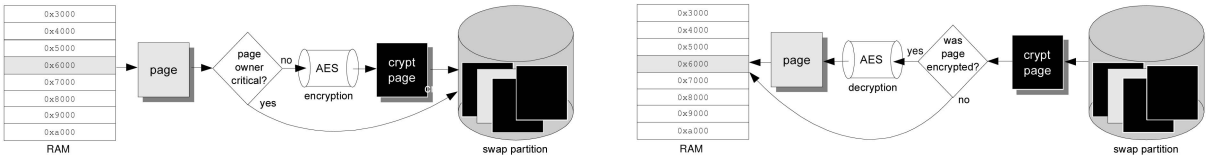

Fig. 4. **Swapfile I/O in Transient Authentication**. "Critical" processes remain running while the token is absent. (a) Swap-out. (b) Swap-in.

need to swap in will not require it. The kernel may try to swap out a page which belongs to a frozen process. Since this swap-out is done by the swapper (to make room) and not because of any action of the process who owns the page, we just reject this swap-out attempt. The system must ensure the encryption keys are pinned in memory, so an attacker cannot find them in the swapfile.

4.4 Video

Besides RAM, disk, and swap, data can creep into other parts of the machine. Examples include video memory and device buffers in disk controllers, network interfaces, and other peripherals. We examined one example: video memory. A determined attacker, thwarted in his attempts to extract information from the disk, swap, and RAM, might probe the video framebuffer, which may contain useful data, like the contents of a spreadsheet window.

At a minimum, Transient Authentication should lock the keyboard and mouse once the token has departed. This precludes any console-based attacks and forces an adversary to resort to hardware probing. The hard question was how best to clear video memory of useful information.

One option would be to directly write to and clear all memory on the video card. This provides a strong guarantee that all useful data was erased. Unfortunately, not all video cards are alike—one would have to handcraft a solution for each model of card.

Display managers interact with diverse video cards by writing to a generic framebuffer interface. This corresponds to the video RAM of the card, `mmap`'ed into a region of RAM. Clearing this region of RAM erases all video information, apart from the video card's buffers.

We chose to blank the framebuffer memory via the display manager, rather than write a video card-specific solution. To avoid modifying the display manager, we use a small helper application that normally runs invisibly in the background and blanks the display when the token departs, overwriting the contents of the framebuffer. Finally, it locks the keyboard and mouse interfaces. Once the token returns, the kernel signals it to reverse the process. It does so simply by restoring the keyboard and mouse events and hiding its window. This "hide" forces the display server to repaint the desktop underneath, restoring the original system state.

The display manager may perform extra buffering of its own. In that case, blanking the display erases the data in the framebuffer but not that cached buffer. Note, however, that the display manager is just another application that will be encrypted in-place when the token is absent. Thus, even if it maintains a "double buffer," it will be encrypted in RAM along with everything else.

5 Application-Aware Mechanisms

Transparently securing all data and all processes on a mobile device is an effective technique. There are situations, however, where this blanket protection scheme is overkill. Processes that rarely or never handle sensitive data must unnecessarily be stopped. Processes that do handle sensitive information may handle only a small amount and may be able to identify and secure that data themselves. Open network connections may not survive the hibernation process and may need to be restarted each time the user returns. And, even if a process is marked as "nonsensitive," if it communicates with any other process that will be stopped, then both must be stopped.

In response, we developed an interface that allows applications to protect their own sensitive information. To the kernel, an application's address space has no semantic meaning. On the other hand, the application knows what data is stored at which addresses. This allows varying degrees of security in how the system responds to loss of authentication. Those applications that deal with no sensitive information continue running unmodified while the user is absent. The system halts high-security applications or those that cannot be modified and encrypts their address space entirely, as described above. Applications that fall between these two extremes, those that can identify the small amounts of sensitive information they possess, should utilize this interface.

To use Transient Authentication services, applications link with our library. Applications may need to be restructured to depend on capabilities, such as keys, which are stored on the token. Some applications already manage authentication and access to sensitive resources, but most revoke access either through explicit user logout or a timeout expiration. Utilizing our system to provide authentication information results in improved security for such applications.

The token can also encrypt and decrypt small pieces of data on behalf of applications. If the token holds an application's key, the application ships a small buffer to the token, which encrypts the buffer and ships it back to the device. This prevents the cleartext key from ever being stored on the mobile device. Note that the channel between the token and device has already been secured via a pairwise key exchange.

Our design is shown in Fig. 5. Unmodified applications are secured as described above (transparently frozen and encrypted in their entirety) when loss of token contact is detected. Modified applications link with a Transient Authentication library and communicate with the kernel-level Transient Authentication components via a user-space

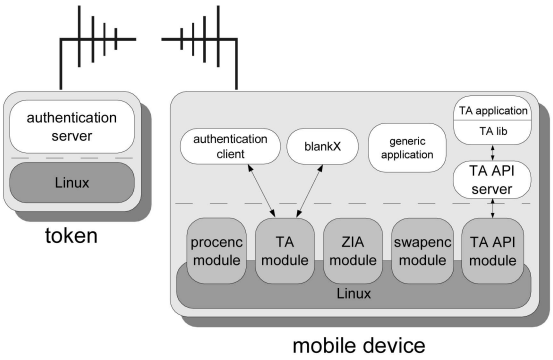

Fig. 5. **Design with application-aware protection mechanisms**. Unmodified applications are secured directly by the ZIAfs and VM Crypt (RAM + swap) modules. Applications modified to use our Transient Authentication API are linked with a TA library and interface with the kernel components via the TA API user-space server.

```
typedef enum ta_change{ TA_LOSS, TA_GAIN } ta_change_t;
typedef int (*ta_auth_hldr_t ) ( IN ta_change_t change, IN int flags );

/* Register an application with the library */
int ta_application_reg( IN char* app_name, IN char* username );

/* Register a handler for change in authentication */
int ta_auth_change_reg( IN int appid, IN ta_auth_hdlr_t hdlr );

/* Decrypt a buffer on the token with a key */
int ta_decr_buf( IN int appid, IN char* keyid, IN char* inbuf, IN size_t inlen,
                 OUT char** outbuf, OUT size_t* outlen );

/* Encrypt a buffer on the token with a key */
int ta_encr_buf( IN int appid, IN char* keyid, IN char* inbuf, IN size_t inlen,
                 OUT char** outbuf, OUT size_t* outlen );
```

Fig. 6. **Application-Aware API**. Functions allow registration with the user-space server, registration of authentication callback functions, and requests for buffer decryption by the token, using a previously registered key.

daemon (called the TA API server). This user-level daemon allows modified applications to communicate with the token to retrieve keys, send decryption requests, and receive decrypted data results.

5.1 Idetifying Secret Data

The most difficult part of protecting an application is deciding which of its data are secrets. There are no general rules-of-thumb, as the situation is different depending on the specifics of the application in question. Once an application's secret data has been identified, the application designer secures this data using either of two mechanisms provided by the Transient Authentication API.

The first method requires an "application key" from the token. The token permanently stores this key and the application caches it when in proximity to the token. When the token departs, the TA API server notifies the application. The application then encrypts its secrets with this key and erases its local copy of the key. When the token later returns, the application refetches the key from the token, decrypts these secrets, and resumes normal execution.

The second method is to always store secret application data encrypted while in RAM and have the token decrypt it. When secret data needs to be accessed, the application ships it to the token. The token holds the encryption key (which exists nowhere on the mobile device) and then honors encrypt or decrypt requests and returns the processed data.

The second method provides slightly better security since the key never exists on the mobile device. However, this puts device-token communication latency into the critical path of application execution. The first method is preferable for applications that access their secret data often.

5.2 API

To use a modified application, the user first installs an application *master key* on her token. Master keys are 128-bit AES keys [13], same as the keys used for transparent protection of disk and virtual memory. An administrative authority installs these keys, which must never be exposed outside the token, much as administrators install initial accounts today. As discussed below, these keys are typically used as *key-encrypting-keys* but can occasionally be used to process small amounts of data directly, on behalf of the mobile device.

Once the master key is installed, the Transient Authentication API is available to the application. Fig. 6 provides an overview of the API. When a modified application starts, it must do two things. First, it registers with Transient Authentication via the API call `ta_application_reg()`. This registration contains the application name and username of the user, which allows Transient Authentication to verify a master key has been installed on the token for this user/application pair. Second, the application must install a handler. This is a callback function, invoked when the token comes in and out of range.

The application can have the token directly encrypt and decrypt small amounts of data on its behalf by calling `ta_encr_buf()` and `ta_decr_buf()`. This makes sense when the data items are small (e.g., passwords, credit card numbers), making it feasible to ship encrypted copies to the token. When this interface is used, the encryption key is the master application key itself, which exists only on the token. To improve performance, the application caches decrypted copies of this data, but must discard them when notified of token departure.

Some data, however, cannot efficiently be secured this way. Passing large amounts of data back and forth over the wireless link would be wasteful of battery power. Also, the token is presumed to be a small, low-power device with considerably less computational power than the mobile device. Encrypting and decrypting large amounts of data takes a prohibitive amount of time.

Instead, to protect larger data elements the application first creates a *submaster key*. A submaster key covers a large piece of secret application data (e.g., the text of an e-mail message). Submaster keys are encrypted by the application master key for permanent storage. The application stores them unencrypted only while the token is present. To ensure submaster keys are never generated without the involvement of token, they are created as follows. First, the application chooses a pseudorandom number. It then asks the token to "decrypt" this number with the master key. The result is the encrypted submaster key, which is permanently stored on the mobile device. The unencrypted

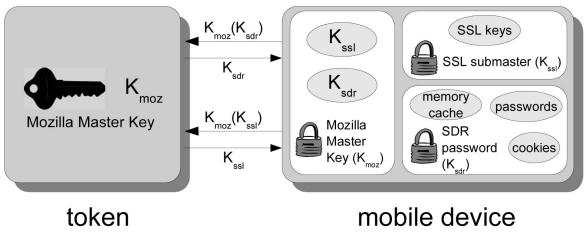

Fig. 7. **Mozilla Modifications**. Cookies, cached user passwords, and the browser cache are secured by submaster key K_{sdr}, via Mozilla's Secret Decoder Ring (SDR) module. SSL session keys are cached in memory for performance, secured by submaster key K_{ssl}, and flushed when the token departs. A global, master key, K_{moz}, protects the two submaster keys, K_{ssl} and K_{sdr}.

version is cached during normal operation. When the application is informed of token departure, it erases this unencrypted key. Upon return of the token, the application ships the encrypted submaster key to the token, using the API function for decryption requests, `ta_decr_buf()`, thus retrieving the unencrypted key.

5.3 Example Application (Mozilla)

As a proof-of-concept, we modified the Mozilla Web browser to secure its secrets via this API. The details of modifications to Mozilla and other applications (such as PGP and OpenSSH) can be found in our previous work [11]. Modifying a Web browser was the most challenging of the three, given the large code-base and the many different types of secrets a browser commonly holds.

For instance, Web browsers may cache user passwords to Web forms, saving the user from having to type their Webmail password each time they check their e-mail. Many Web sites cache credentials by installing a cookie on the user's hard drive, allowing the user to reauthenticate automatically in the future. A browser uses Secure Socket Layer (SSL) encryption to establish a secure session with Web servers and often caches Web pages and images locally for performance. We focused on securing four main categories of secrets within Mozilla: cached user passwords, cookies, SSL session keys, and the browser cache.

Mozilla already features a module, the Secret Decoder Ring (SDR), which can be used to encrypt or decrypt arbitrary data. Its API requires an explicitly provided key. The SDR was therefore the ideal location to add code utilizing our API. The components of our modified Mozilla are illustrated in Fig. 7. Mozilla's SDR module secures all of the user's cookies, cached user passwords, and the browser cache, and we added a new facility to protect SSL secrets. We created two submaster keys: K_{sdr} and K_{ssl}, both of which are secured by the application master-key K_{moz}. Since cookies, passwords, and the contents of cache are all used relatively infrequently, they are stored on a disk encrypted by K_{sdr} and decrypted on demand.

SSL session keys, however, are used frequently. Therefore, the modified Mozilla keeps them cached in the clear during normal operation. When the token departs, the session keys are encrypted in-place by K_{ssl}. When the token returns, Mozilla must send $K_{moz}(K_{ssl})$ to the token, which decrypts the submaster key and returns K_{ssl} to the application. Mozilla then decrypts all the SSL session keys and resumes normal operation.

It is true that this porting effort was nontrivial; it took some time to identify Mozilla's sensitive state and protect it properly. This process may not be feasible for many legacy applications and they can rely on whole-process protection if need be.

6 Evaluation

We currently have a working prototype of all software components comprising Transient Authentication. In fact, one of the authors used the system to secure his personal machine for several months with no ill effects. We have ported the Transient Authentication software to protect both x86-based devices (such as laptops) and iPAQ handhelds with ARM processors. The token software has been successfully ported and tested on both hardware platforms as well.

To evaluate the system, we installed Transient Authentication on an IBM ThinkPad X24 notebook with a 1.113 GHz Intel Pentium III CPU, 256 MB of system RAM and a 30.0 GB IDE disk drive with a 12 ms average seek time. The laptop ran Linux kernel 2.4.20. The token was a Compaq iPAQ 3870 handheld with a 206 MHz StrongARM processor, 64MB of SDRAM, and 32MB of Flash ROM, running Familiar Linux. Granted, this prototype token is certainly larger and somewhat more powerful than current wearable devices, but rapid technological advances make it reasonable to assume that wearable devices with similar capabilities will soon be available.[1] At the time this experiment was performed, the Bluetooth interface was not reliable. To come as close as possible to its performance properties, the token and laptop communicated via 802.11b wireless LAN cards in ad hoc mode at 1 Mbps.

6.1 File System

The most important metric in the file system evaluation was the effect on user-visible performance. We compare the performance of Linux's ext2fs against two other file systems: Cryptfs and ZIA. Cryptfs provides file and name encryption, but uses a single, static key for the entire file system. Cryptfs is drawn from the FiST distribution [35], but uses Blowfish [28] rather than Rijndael [13], the cryptosystem used in ZIA. To provide a fair comparison, we replaced Blowfish with Rijndael in Cryptfs, improving its performance. All keys used in the file system were 128 bits long.

We first need to understand the overhead imposed by ZIA on typical system operation. Our benchmark is similar to the Andrew Benchmark [20] 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 to minimize the benefit of caching as compared to the original benchmark. It is 28.2 MB in size; when compiled, the total tree occupies 59.7 MB. We preconfigure the source tree for each trial of the benchmark since the configuration step does not

1. The IBM WatchPad wristwatch, for example, currently features a 74 MHz processor, a Bluetooth radio, and runs Linux 2.4.

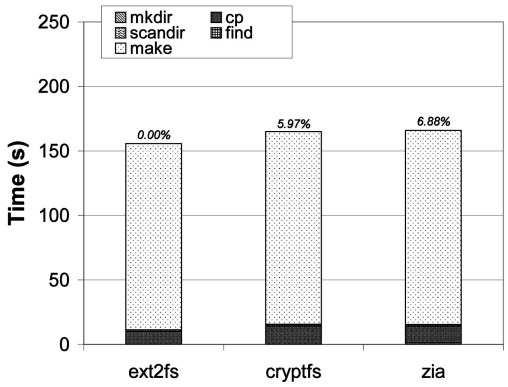

Fig. 8. **File System: Andrew Benchmark results**. ZIA's performance is less than 7 percent worse than ext2fs and is indistinguishable from that of Cryptfs.

involve appreciable I/O in the test file system. For this benchmark, ZIA imposes less than a 7 percent penalty over ext2fs. Its performance is statistically indistinguishable from that of Cryptfs, which uses a single key for all cryptographic operations. The results are summarized in Fig. 8.

6.2 Physical Memory

When the user departs, Transient Authentication secures the data found in physical memory via a three-step process: 1) freeze execution of all running processes (except the kernel and the token authentication client), 2) encrypt in-place the physical memory pages of the frozen processes, and 3) overwrite freed pages and other shared kernel buffers. When the user subsequently returns, the process must be reversed, by decrypting RAM pages and restarting all frozen processes.

To evaluate the overhead of these subtasks, a test program allocated 200 MB of memory, and filled it with pseudorandom data (from the Linux device `/dev/random`). This occupied all of physical memory, and in fact pushed some data into swap. Reading from `/dev/random` forced the entire 200 MB to be allocated, rather than just mapping to a single, zeroed, copy-on-write page.

We then repeatedly brought the token in and out of range and recorded the time spent performing each of the above tasks. Table 1 shows the results averaged over 10 runs. In total, 56 processes were frozen and thawed on each run.

Unsurprisingly, encrypting and decrypting memory dominates the total execution time. On average, 46,740 pages, equaling 182.58 MB of data, were encrypted and decrypted on each run of our test. It took the same amount of time to secure when the user departs as to restore when she returns (approximately 7.2 seconds). When the user returns, the token will enter range of the mobile device when she is still several meters away. It will take approximately this long for her to reach the device, sit down, and be ready to work. Thus, Transient Authentication secures a user's data without any user-visible disruption in the common case.

We also compared encrypting RAM in-place to merely suspending to disk and overwriting RAM when the user departs. If the suspended image were encrypted on disk, this would provide comparable security to that of our solution. We examined three scenarios—overhead just to

TABLE 1
Overhead of Securing Running Processes

	time	σ	error
freeze processes	5.819 ms	2.868 ms	$\pm$1.814 ms
encrypt RAM	7.157 s	0.068 s	$\pm$0.030 s
overwrite free pages	7.970 ms	4.675 ms	$\pm$2.957 ms
decrypt RAM	7.209 s	0.158 s	$\pm$0.099 s
thaw processes	5.993 ms	1.244 ms	$\pm$0.787 ms
total encrypt	7.171 s	0.071 s	$\pm$0.045 s
total decrypt	7.215 s	0.156 s	$\pm$0.098 s

Data was collected on 10 runs. On average, 46,740 pages (182.58 MB) were secured and 1,144 pages (4.47 MB) overwritten. Fifty six processes were frozen and then thawed.

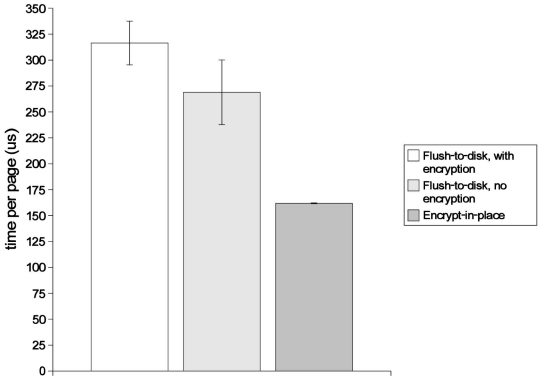

Fig. 9. **Physical Memory: Encryption in-place versus suspending to disk**. Overhead (in μs) required to secure a 4 KB page by one of three methods: 1) flush to disk in encrypted form, 2) flush to disk unencrypted, and 3) encrypt in-place.

write the data to disk (no encryption), encryption before flush-to-disk, and encryption in-place (our solution). Fig. 9 shows time to secure one page in milliseconds. Encrypting RAM in-place was nearly twice as fast as suspending to disk. This is true even if the data is not encrypted, since RAM write throughput is orders of magnitude faster than that of disk.

6.3 Swap Space

In evaluating our implementation of secured swap space, we sought to answer the following questions:

- What is the user-visible effect on system performance?
- What is the source of the overhead imposed by this implementation?
- Is the implementation correct?

We ran two experiments—one that highlights the overhead seen by the user and another that further explains the source of this overhead. In each case, we compare performance of our implementation to that of an unmodified system and an alternate swap encryption system—specifically, loopback device encryption.

The loopback device is commonly used to interpose on or filter file I/O. Using the `losetup` utility, the user initializes a mapping between a loop device and an actual file or disk partition. Applications then read/write to/from the loop device as if it were a file, and data passes through the loop driver before being passed on to the mapped file or partition. When file I/O data passes through the loop

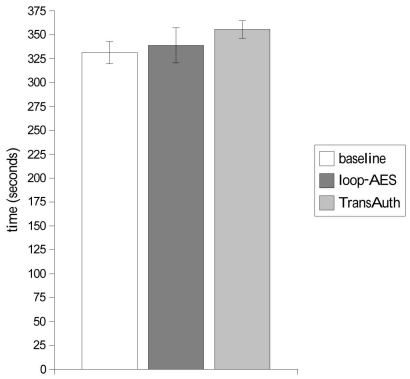

Fig. 10. **Swap: User-Visible Performance**. Mean time, in seconds, over five trials, to perform a Linux 2.4.24 kernel compile (make bzImage). "Baseline" refers to no swap encryption whatsoever, "TransAuth" to our implementation, and "loop-AES" to encryption via the loopback device. Error bars represent the 95 percent confidence bound on the population mean, as determined by the standard $\mu_{pop} = \mu \pm \frac{2\sigma}{\sqrt{n}}$ formula.

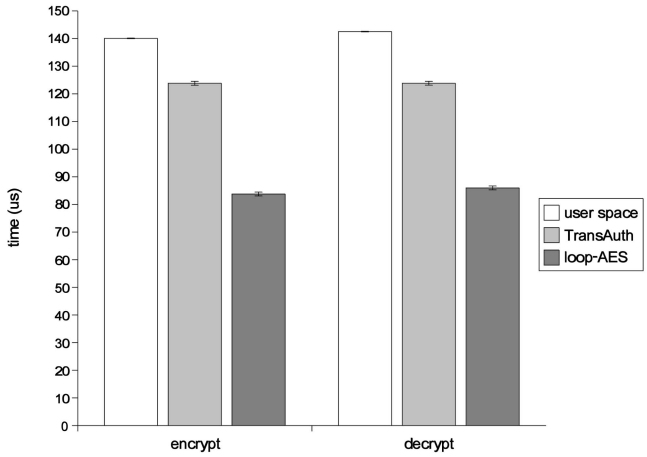

Fig. 11. **Swap: Single Page Encryption Benchmark**. Mean time, in microseconds, to both encrypt and decrypt a 4 KB memory page, using AES encryption in CBC mode, with 128-bit keys. Error bars represent the 95 percent confidence bound on the population mean, as determined by the standard $\mu_{pop} = \mu \pm \frac{2\sigma}{\sqrt{n}}$ formula.

device, the driver performs any operations on it that it wishes. In our case, we initialize a mapping between a loopback device and the swap partition and configure it to perform 128-bit AES encryption (in CBC mode) on writes, and equivalent decryption on reads.[2]

6.3.1 User-Visible Performance

To evaluate user-visible performance, we used a Linux kernel source compile as the benchmark. Each run started from a preconfigured, clean source tree, and an empty swap partition. We compared the kernel compile time for an unmodified Linux 2.4 kernel, our swap encryption implementation, and a loopback encryption implementation.

The initial test runs resulted in little or no activity in the swap partition. This was because the working set of data manipulated by our chosen benchmark fit within the 256 MB of RAM on our test system. We therefore reconfigured the kernel at boot to use only 128 MB of RAM so that the compile operation would actually cause a great deal of swapfile activity.

For each of the three test configurations, five trials were run. The mean (with error) is shown in Fig. 10. The results show the mean execution time for our implementation and loopback encryption to be statistically indistinguishable. Our implementation imposes only 7 percent time overhead, as compared to the baseline case of no swap encryption.

6.3.2 Microbenchmark

To explain the source of this overhead, we examined the total time to encrypt and decrypt one 4 KB page of memory. We compared the same three systems used in the previous experiment.

To collect data on our implementation, we instrumented the kernel module to measure the execution time for swap-outs and swap-ins. Specifically, we timed from the point where our modifications started until execution passed back into unmodified kernel code. In this way, we measured not just the overhead to perform AES encryption/decryption on a 4,096 byte page, but also the overhead of calling the functions in our kernel module, and the overhead introduced by these functions as well. The Linux `get_cycles()` macro allowed for precise timing measurements. This macro reduces (on the x86 architecture) to a read of the RDTSC register that returns a 32-bit cycle count.

We also instrumented the loop driver in a similar fashion, to count the number of cycles executed inside the loopback device. For a user-space comparison, we copied the encryption/decryption code used by the swap encryption module and used it in the same fashion, but as a user-space application. A 4 KB buffer of memory was populated with random data and then repeatedly encrypted and decrypted in place.

The `get_cycles()` macro returned cycle counts at the start and end of the "noteworthy" sections of code in all three examples. The mean values are shown in Fig. 11. In all cases, n was on the order of 10,000 runs.

The results show loopback encryption takes approximately 80 μs per swap-out or swap-in, while our implementation takes approximately 120 μs. The overhead of loopback encryption is extremely low because it blindly encrypts and decrypts everything with one global key. On the other hand, Transient Authentication performs additional work to track which pages belong to which processes since we do not encrypt the pages of critical processes.

This overhead will be masked by the disk I/O overhead required to read or write the page to disk. For example, the drive in our test system featured an Ultra EIDE (ATA/100) interface, with a seek time of 12ms at 5,400 RPM, and maximum external transfer rate of 100 Mbits/s. The time to seek to the correct location on disk is itself several orders of magnitude longer than the encryption overhead. Even ignoring seek time, the theoretical lower bound to transfer 4,096 bytes at 100 Mbits/s is approximately 39 μs, which is by itself about one-third of our measured encryption overhead. Furthermore, the previous experiment shows the 50 percent performance premium over loopback encryption for single-page encryption is "hidden

2. For additional project details, please see http://loop-aes.sourceforge.net/.

TABLE 2
Video: Time to Secure Screen, Keyboard, and Mouse

	time (μs)	error (μs)
blank screen	13.215	$\pm$ 0.043
unblank screen	11.739	$\pm$ 0.044

All times in microseconds. "Error" is the 95 percent confidence bound on the population mean.

in the noise" of normal system operations since the performance of Transient Authentication and loopback encryption are statistically indistinguishable.

The results for user-space encryption are interesting in that both kernel-located solutions performed better. The reason is the the user-level process was often preempted to allow the scheduler to run. Both our implementation and the loopback implementation, on the other hand, execute inside the kernel, and therefore would not be as vulnerable to delays due to process scheduling.

6.3.3 Correctness

While it is easy to find evidence of failure (e.g., finding bits of plaintext in swap), it is difficult or impossible to empirically prove the correctness of a design. Nonetheless, we devised an experiment to examine our effectiveness in keeping unencrypted data out of the swap file. A helper application allocated a specified amount of memory and filled it entirely with a given string.

We used this tool to repeatedly allocate 512 MB of memory (twice the actual physical memory), forcing most of the existing RAM contents into swap. We repeatedly ran two different runs: with swap encryption active and with no encryption. In both cases, a file of size 64 MB was used as the backing store (as opposed to the customary raw swap partition), to facilitate easy post-test inspection. After each run, we piped the swapfile through the UNIX `strings` command, to generate a text file listing of all ASCII strings found in the file. This strings list was then matched against a dictionary of approximately 104,000 English words.[3] Only strings of four or more characters were matched to eliminate spurious matches that arise randomly. We collected data on 10 runs, and in none of the encrypted runs did the post-examination ever reveal an English word in the swapfile. For the unencrypted runs, on the other hand, the scripts found thousands of words.

6.4 Video

We also measured the overhead needed to secure the video and console of the device. The test repeatedly blanked the screen and locked the keyboard and mouse, then reversed the process. Table 2 shows the measured overhead. The time to secure the video and console is on the order of tens of microseconds, an interval that is imperceptible to a human being. As we need only secure and restore the system on human time, rather than on computer time, this overhead is acceptable.

3. This list (english.words) was maintained at Digital Equipment Corporation (DEC) by Jorge Stolfi and is the product of lists compiled by Andy Tanenbaum, Barry Brachman, Geoff Kuenning, Henk Smit, and Walt Buehring.

TABLE 3
Overhead to Secure Mozilla Using the Application-Aware Framework

	Cookie Overhead (s)	Page Load Time (s)
`cnn.com`	0.030 (0.004)	3.083 (0.500)
`ebay.com`	0.086 (0.032)	2.758 (0.625)
`espn.com`	0.528 (0.060)	7.390 (0.715)

TABLE 4
Mozilla Protection and Recovery Overhead

	Protect (s)	Restore (s)
Memory Cache	0.209 (0.002)	0.211 (0.003)
SSL Keys	0.004 (0.001)	0.068 (0.004)
SDR (16 byte key)	N/A	0.059 (0.009)

6.5 Application-Aware Mechanisms

Our changes to the Mozilla Web browser affected the way it manages stored password data and cookies. Mozilla already secures cached passwords and our modifications only change how the encryption keys are managed, without adding any additional encryption or decryption overhead.

Our evaluation of our application-aware API focused, therefore, on the overhead introduced by encrypting and decrypting cookies at each access. We loaded three popular Web pages and measured the overhead of each cookie encryption and decryption operation. We also measured the total time required to load the page each run. The cookie cached was flushed between each trial. Each page was loaded 10 times.

The mean and standard deviation for both values, across all ten runs, are shown in Table 3. "Cookie Overhead" is the total time in seconds required to perform all cookie encryption and decryption operations caused by one page load. One can see the overhead imposed to secure site cookies is dwarfed by the page load times and, therefore, should be unnoticeable to users.

We also measured the time required to protect and restore Mozilla when the user leaves and returns. To measure this, we connected to our department's secure Web server, establishing a connection with an SSL key. For each run, we moved the token out of range and measured the time needed to secure the SSL key and the Mozilla memory cache. We then reconnected the token and measured the time required to restore the memory cache, SSL key, and the Secret Decoder Ring. Since Mozilla simply flushes the SDR secrets when the token departs, there is no data to collect. The results are shown in Table 4.

7 Related Work

Proximity-based hardware tokens are a commonly proposed way to detect the presence of authorized users. For instance, Landwehr [21] proposes disabling access to the keyboard and mouse of a system when the user is away, via a hardware mechanism. His solution is vulnerable to physical-possession attacks, however, since running memory state is not encrypted while the user is absent.

Rather than use hardware tokens, the user could authenticate via biometric information [15]. Use of biometrics intrudes on users in two ways. First, the false negative rate (rejecting a valid user) is often high [24]. For face recognition, the rate ranges from 10 to 40 percent, depending on the duration of system training time. Second, biometric authentication systems necessarily constrain users physically. For example, to use fingerprint identification, the user must keep their finger placed on the reader while using the system. This burden may encourage users to find workarounds. An exception is iris scanning technology. The false negative rate is low, and authentication is performed unobtrusively, allowing users to work normally [23]. These systems require three cameras, however, which is infeasible for use with small, mobile computers.

Our system puts much stock in the fact that cleartext keys are *never* written to anywhere in persistent store. Thus, since they exist solely in RAM, these keys are "forgotten" simply by overwriting their addresses. It is difficult, however, to completely erase previously stored values from either memory or disk [17]. Crescenzo et al. [12], propose *erasable memory*, a system whereby a small block of provably erasable memory can be leveraged to simulate a much larger block (consisting of the machine's entire main memory). This scheme requires special hardware to succeed, however, and our threat model is not sufficiently paranoid as to require this. It would certainly be possible to apply their techniques to Transient Authentication if the situation required this additional guarantee of key confidentiality.

8 Conclusions

The small, lightweight nature of mobile devices, combined with their common usage environments (in public places, amid many untrusted people) makes them an easy target for theft. More important than loss of hardware may be the cost of information exposure. Current system either require burdensome reauthentication, or leave users' data exposed for an unacceptably large time window.

Transient Authentication relieves this tension between security and usability through use of a wireless, wearable hardware token. The token and device maintain constant contact while the user is present. When the user departs, the device detects this and takes steps to secure the user's data. Thus, the user reaps the benefits of constant reauthentication without the burden of having to explicitly do the work herself.

We showed how this authentication framework can be leveraged to secure the data present in the myriad memory and storage locations of mobile devices. Our results showed a device can be secured and restored in a matter of seconds, maintaining usability for the user. The overhead imposed to secure various data spaces was analyzed and found to be minimal. Additionally, we presented a user-level library and API that allows applications to be modified to use the authentication framework directly, to protect application-specific secrets.

Acknowledgments

The authors would like to thank Adrian Perrig, the anonymous reviewers, Minkyong Kim, Landon Cox, James Mickens, and Sam Shah for their useful comments and feedback that greatly improved the quality of the manuscript. This paper is based upon work supported by the National Security Agency and the US National Science Foundation under Grant Nos. CCR-0208740, CNS-0447877, and DUE-0416863. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the National Security Agency or the US National Science Foundation.

References

[1] A. Adams and M.A. Sasse, "Users Are Not the Enemy: Why Users Compromise Security Mechanisms and How to Take Remedial Measures," *Comm. ACM,* vol. 42, no. 12, pp. 40-46, Dec. 1999.

[2] R. Anderson, "Why Cryptosystems Fail," *Comm. ACM,* vol. 37, no. 11, pp. 32-40, Nov. 1994.

[3] B.V. Bigelow, "Qualcomm Secrets Vanish with Laptop," *San Diego Union-Tribune,* 2000.

[4] M. Blaze, "A Cryptographic File System for UNIX," *Proc. First ACM Conf. Computer and Comm. Security,* pp. 9-16, Nov. 1993.

[5] M. Blaze, "Key Management in an Encrypting File System," *Proc. Summer 1994 USENIX Conf.,* pp. 27-35, June 1994.

[6] S. Brands and D. Chaum, "Distance-Bounding Protocols," *Proc. EUROCRYPT Conf.,* pp. 344-359, 1993.

[7] M. Burrows, M. Abadi, and R. Needham, "A Logic of Authentication," *ACM Trans. Computer Systems,* vol. 8, no. 1, pp. 18-36, Feb. 1990.

[8] S. Carlton, J. Taylor, and J. Wyszynski, "Alternate Authentication Mechanisms," *Proc. 11th Nat'l Computer Security Conf.,* 1988.

[9] G. Cattaneo, L. Catuogno, A.D. Sorbo, and P. Persiano, "The Design and Implementation of a Transparent Crytographic File System for UNIX," *Proc. Freenix Track: 2001 USENIX Ann. Technical Conf.,* pp. 199-212, June 2001.

[10] M. Corner and B. Noble, "Zero-Interaction Authentication," *Proc. Eighth Int'l Conf. Mobile Computing and Networking (ACM MobiCom '02),* Sept. 2002.

[11] M.D. Corner and B.D. Noble, "Protecting Applications with Transient Authentication," *Proc. First Int'l Conf. Mobile Systems, Applications, and Services (MobiSys '03),* May 2003.

[12] G. Di Crescenzo, N. Ferguson, R. Impagliazzo, M. Jakobsson, C. Meinel, and S. Tison, "How to Forget a Secret," *Proc. 16th Ann. Symp. Theoretical Aspects in Computer Science (STACS '99),* pp. 500-509, Mar. 1999.

[13] J. Daemen and V. Rijmen, *AES Proposal: Rijindael. Advanced Encryption Standard Submission,* second ed. Mar. 1999.

[14] D. Davis, "Compliance Defects in Public-Key Cryptography," *Proc. Sixth USENIX Security Symp.,* pp. 171-178, 1996.

[15] F. Deane, K. Barrelle, R. Henderson, and D. Mahar, "Perceived Acceptability of Biometric Security Systems," *Computers and Security,* vol. 14, no. 3, pp. 225-231, 1994.

[16] W. Diffie, P. van Oorschot, and M. Wiener, *Design Codes and Cryptography.* Kluwer Academic, 1992.

[17] P. Gutmann, "Secure Deletion of Data from Magnetic and Solid-State Memory," *Proc. Sixth USENIX Security Symp.,* pp. 77-89, July 1996.

[18] L. Hu and D. Evans, "Using Directional Antennas to Prevent Wormhole Attacks," *Proc. 11th Network and Distributed System Security Symp. (NDSS '04),* Feb. 2004.

[19] Y.-C. Hu, A. Perrig, and D.B. Johnson, "Packet Leashes: A Defense Against Wormhole Attacks in Wirless Ad Hoc Networks," *Proc. 22nd Ann. Joint Conf. IEEE Computer and Comm. Soc. (INFOCOM),* pp. 1976-1986, Apr. 2003.

[20] J. Howard et al., "Scale and Performance in a Distributed File System," *ACM Trans. Computer Systems,* vol. 6, no. 1, pp. 51-81, Feb. 1988.

[21] C. Landwehr, "Protecting Unattended Computers without Software," *Proc. 13th Ann. Computer Security and Applications Conf. (ACSAC),* pp. 274-283, 1997.

[22] C. Narayanaswami and M.T. Raghunath, "Application Design for a Smart Watch with a High Resolution Display," *Proc. Fourth Int'l Symp. Wearable Computers,* pp. 7-14, Oct. 2000.
[23] M. Negin, Jr., T.A. Chemielewski, M. Salganicoff, T.A. Camus, U.M. Cahnvon Seelen, P.L. Venetianer, and G.G. Zhang, "An Iris Biometric System for Public and Personal Use," *Computer,* vol. 33, no. 2, pp. 70-75, Feb. 2000.
[24] P.J. Phillips, A. Martin, C.L. Wilson, and M. Przybocki, "An Introduction to Evaluating Biometric Systems," *Computer,* vol. 33, no. 2, pp. 56-63, Feb. 2000.
[25] N. Provos, "Encrypting Virtual Memory," *Proc. Ninth USENIX Security Symp.,* Aug. 2000.
[26] A.D. Rubin and P. Honeyman, "Long Running Jobs in an Authenticated Environment," *Proc. Fourth USENIX Security Symp.,* pp. 19-28, Oct. 1993.
[27] A.D. Rubin and P. Honeyman, "Nonmonotonic Cryptographic Protocols," *Proc. Computer Security Foundations Workshop,* pp. 100-116, June 1994.
[28] B. Schneier, "Description of a New Variable-Length Key, 64-Bit Block Cipher (Blowfish)," *Proc. Cambridge Security Workshop,* pp. 191-204, Dec. 1993.
[29] B. Schneier, *Applied Cryptography.* John Wiley and Sons, 1996.
[30] R.E. Smith, *Authentication: From Passwords to Public Keys.* Addison-Wesley, 2002.
[31] T. Ylonen et al., "SSH Protocol Architecture," Internet Draft, Jan. 2001.
[32] D. Verton, "State Department to Punish Six over Missing Laptop," *Computerworld,* Dec. 2000.
[33] T. Ylonen, "SSH-Security Login Connections over the Internet," *Proc. Sixth USENIX Security Symp.,* pp. 37-42, July 1996.
[34] E. Zadok, I. Badulescu, and A. Shender, "CryptFS: A Stackable Vnode Level Encryption File System," Technical Report CUCS-021-98, Computer Science Dept., Columbia Univ., 1998.
[35] E. Zadok and J. Nieh, "FIST: A Language for Stackable File Systems," *Proc. Ann. USENIX Technical Conf.,* pp. 55-70, June 2000.

Anthony J. Nicholson received the BS degree in computer engineering in 1999 from the University of Kansas and the MS degree in computer science in 2005 from the University of Michigan. He is a PhD candidate in the Department of Electrical Engineering and Computer Science at the University of Michigan. His research interests lie in the areas of mobile and pervasive computing, security, operating systems, and networking.

Mark D. Corner has been an assistant professor in the Computer Science Department at the University of Massachusetts-Amherst since 2003 after receiving the PhD degree in electrical engineering from the University of Michigan. His primary interests lie in the areas of mobile and pervasive computing, networking, file systems, and security. He received a US National Science Foundation (NSF) CAREER award in 2005, a Best Paper at ACM Multimedia 2005, as well as the Best Student Paper Award at Mobicom 2002. His work is supported by the NSF, DARPA, and the NSA.

Brian D. Noble received the PhD degree in computer science from Carnegie Mellon University in 1998, and is a recipient of the US National Science Foundation CAREER award. He is an associate professor in the Electrical Engineering and Computer Science Department at the University of Michigan. His research centers on software supporting mobile devices and distributed systems.