Bump in the Ether:
Mobile Phones as Proxies for Sensitive Input

Jonathan M. McCune   Adrian Perrig   Michael K. Reiter

December 8, 2005
CMU-CyLab-05-007

CyLab
Carnegie Mellon University
Pittsburgh, PA 15213

Bump in the Ether: Mobile Phones as Proxies for Sensitive Input

Jonathan M. McCune Adrian Perrig Michael K. Reiter
Carnegie Mellon University
{jonmccune, perrig, reiter}@cmu.edu

Abstract

User-space malware such as keyboard sniffers, spyware, and Trojans represent a significant threat to today's desktop computing environment. Users have little assurance that such malware cannot observe their input to a particular application. In this paper, we present Bump in the Ether (BitE), an approach for preventing malware from accessing sensitive user input and providing the user with additional confidence that her input is being processed as desired.

Rather than preventing malware from running or detecting already-running malware, we facilitate user input that bypasses common avenues of attack. User input traverses a *trusted tunnel* from the input device to the application. This trusted tunnel is implemented using a trusted user device working in tandem with a TCG-compliant host platform. The user device verifies the integrity of the host platform and application, provides a trusted display through which the user selects the application to which her inputs should be directed, and encrypts those inputs so that only the application can decrypt them. We describe the design and implementation of BitE, with emphasis on both usability and security issues.

1 Introduction

Using security-sensitive applications on current computer systems exposes the user to numerous risks. User-level malware such as spyware or keyloggers often monitor and log the user's every keystroke. Through keystrokes, an adversary may learn sensitive information such as passwords, bank account numbers, or credit card numbers. Unfortunately, current computing environments make such keystroke logging trivial; for example, X-windows allows any application to register a callback function for keyboard events destined for any application.[1] Similar vulnerabilities exist in Microsoft Windows.[2]

Besides the ease of eavesdropping on keystrokes, another serious risk to the user is the integrity of screen content. Malicious applications can easily overwrite any screen area with their content.

[1] Giampaolo shows—with his infamous `xkey.c` 100-line C program [19]—that it is easy to capture keyboard input events that the user intended for some other application under X11. We conclude that it is desirable to reduce the involvement of the window manager in sensitive I/O activities as much as possible.

[2] For example, search `http://msdn.microsoft.com/` for `RegisterHotKey` and `SendInput`, and consider the consequences of judicious use of both.

This introduces the threat that a user cannot trust any content displayed on the screen since it may originate from a malicious application. An example of such a vulnerability is that malicious Javascript code embedded in a web page can overwrite security-critical browser elements [43].

In such an environment, it is challenging to design a system that provides the user with guarantees that the correct operating system and the correct application are currently running, and that only the correct application will receive the user's keystrokes. In particular, we would like a computing environment with the following properties:

- The user obtains user-verifiable evidence that the correct OS and the correct application were loaded.
- The user obtains user-verifiable evidence that only the correct application is receiving keystroke events.

Our approach for providing these properties is to establish a user-verifiable trusted tunnel that securely transports keystrokes from the keyboard to the desired application. Figure 1 shows a comparison of the legacy input path versus input through a trusted tunnel. To reduce the user's need to trust the window manager, we use the display on a mobile device as a trusted output mechanism. Given the increasing ubiquity of advanced cell phones or PDAs, we leverage such devices. Mobile devices are continuously increasing in complexity and thus feature software vulnerabilities of their own. We consider the consequences of various compromises in Section 6.

To provide evidence to the user that the correct OS and applications were loaded, we assume the user's computing platform is equipped with a Trusted Platform Module (TPM) as specified by the Trusted Computing Group (TCG), and that the BIOS and OS are TPM-enabled and perform integrity measurements of code loaded for execution [1, 31, 39]. The user's mobile device is used to verify these measurements.

To achieve a trusted tunnel that securely delivers keystrokes to the correct application, we design an OS module that directly passes sensitive keystrokes from the user's mobile device to the correct application, bypassing the vulnerable X-windows system. We designed and prototyped Bump in the Ether (BitE), a system that provides secure user-verifiable trusted tunnels, which we describe in the remainder of this paper.

2 Related Work

We review related work on secure window managers, followed by related work on attestation and integrity measurement.

2.1 Secure Window Managers

A goal of BitE is to ensure that only the correct application is receiving input events, and to provide user-verifiable evidence that this is so. While much prior work has addressed this issue, none of it is readily available for non-expert users on commodity systems today. We now review related work chronologically.

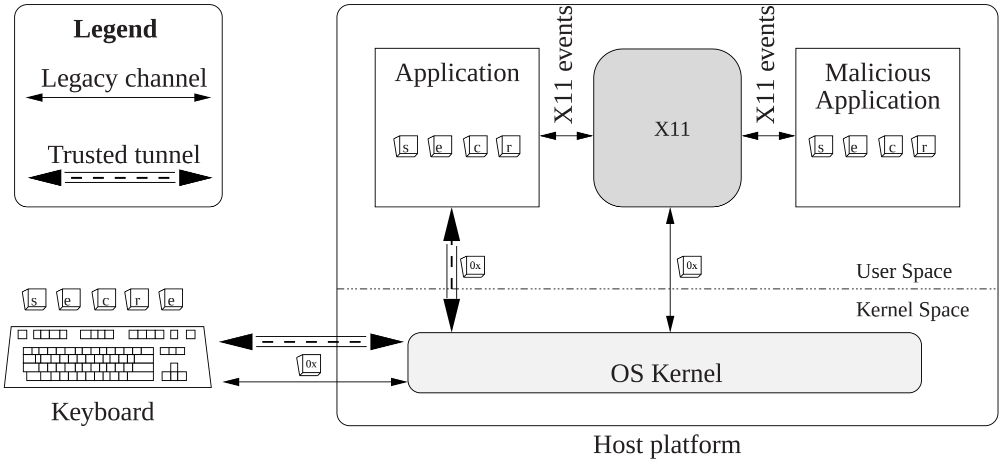


Figure 1: Traditional flow of keystrokes vs. trusted tunnels. On a traditional computer system, keystrokes are first sent to the OS kernel, which passes them to X-windows, which then passes them as X11 events to all applications that register for that class of keyboard event. Unfortunately, malicious applications can register a callback function for keyboard events for other applications. Our trusted tunnels would protect keystrokes and only send them to the desired application.

Several government and military computer windowing systems have been developed with attention to security and the need to carefully isolate different grades of information (e.g., classified, secret, top secret). Early efforts to secure commercial window managers resulted in the development of *Compartmented Mode Workstations* (e.g., [6, 8, 9, 17, 20, 27, 29, 30, 40]), where tasks with different security requirements are strictly isolated from each other. These works consider an operating environment where an employee has various tasks she needs to perform, and some of her tasks have security requirements that necessitate isolation from other tasks. For example, Picciotto et al. consider trusted cut-and-paste in the X window system. Cut-and-paste is strictly confined to allow information flow from low-sensitivity to high-sensitivity applications, so that high-sensitivity information can never make its way into a low-sensitivity application [28]. Epstein et al. performed significant work towards trusted X for military systems in the early 1990s [11, 12, 13, 14, 15, 16]. While these systems are effective for employees trained in security-sensitive tasks, they are unsuitable for use by consumers.

Trostle details some timing attacks against trusted path mechanisms [37]. His attacks greatly reduce the password search space for secrets entered using systems' trusted path mechanisms (e.g., Ctrl+Alt+Del to login to a machine running Microsoft Windows).

Shapiro et al. propose the EROS Trusted Window System [34], which demonstrates that breaking an application into smaller components can greatly increase security while maintaining very powerful windowing functionality. Unfortunately, EROS is incompatible with a significant amount of legacy software, which hampers widespread adoption. In contrast, BitE works in concert with existing window managers.

Common to the majority of these schemes is a mechanism by which some portion of the computer's screen is trusted. That is, an area of the screen is controlled by some component of the trusted computing base (TCB) and is inaccessible to all user applications. Due to the complexity of X, it is difficult to implement this trusted screen area in an assurable way. BitE uses the trusted mobile device's screen as a trusted output device. In Section 6, we discuss other issues with trusted window managers—such as the semantics of full-screen mode. We defer this discussion until after the presentation of BitE.

We emphasize that, despite the large body of work on trusted windowing systems, the majority of users do not employ any kind of trusted windowing system. Thus, we proceed under the assumption that users do not want to change their windowing system. In the remainder of the paper, we show that BitE can increase user input security under these conditions.

2.2 Attestation and Integrity Measurement

BitE depends on the ability of the user's computing platform to provide attestations of the software executing therein. In order to be capable of attestation, the platform must be equipped with an integrity measurement architecture (IMA). In this section we provide some background on attestation and integrity measurement.

The Trusted Computing Group (TCG) is an organization that promotes open standards to strengthen computing platforms against software-based attacks [1, 2]. The TCG specifies a Trusted Platform Module (TPM). A TPM is a dedicated security chip that enables many trusted computing features, including sealed storage, attestation, and integrity measurement. Computers featuring TPMs are readily available today.

TPMs can generate a Storage Root Key (SRK) that will never leave the chip. The SRK enables *sealed storage*, whereby data leaving the TPM chip is encrypted under the SRK for storage on another medium. Several other keys are maintained by the TPM and kept in sealed storage when not in use. One of these is the Attestation Identity Key (AIK), which is an RSA signing keypair used to sign attestations. To the remote party trying to verify the attestation, the AIK represents the identity of attesting platform.

TPMs have one-way platform configuration registers (PCRs) that an IMA can extend with *measurements* (typically cryptographic hashes computed over a complete executable) of software loaded for execution. The IMA extends the appropriate PCR registers with the measurement of each software executable just before it is loaded. Figure 2 shows the architecture of a host making use of the TPM and integrity measurement.

An *attestation* produced by the IMA and TPM consists of two parts: (1) a list of the measurements of all software loaded for execution, typically maintained in the OS; and (2) an AIK-signed list of the values in the PCR registers. A remote party with an authentic copy of the public AIK can compute the expected values for the PCR registers based on the measurement list, and check to see whether the signed values match the computed values. The end result is a *chain* of measurements of all software loaded since the last reboot. The security requirement is that all software is measured before being loaded for execution.

Sailer et al. developed an IMA for Linux [31]. They show that it is difficult to manage the integrity measurement of a complete interactive computer system, since the order in which applications are executed is reflected in the resulting PCR values. During the boot process, however, a well-behaved system always loads in the same order. Hence, integrity measurement of the system from boot through the loading of the kernel, its modules, and deterministic system services will be consistent across multiple boot cycles on a well-behaved host platform.

Sealed storage enables another feature that is useful to BitE: data can be encrypted under a key generated based on the current PCR values. The stored data is thus inaccessible unless the host platform's software state is consistent with the state recorded during the initial creation of the key. When using sealed storage, it is assumed that the user's platform will always load certain software in the same order. If the order changes, the system is considered to be sufficiently different to invalidate existing security relationships.

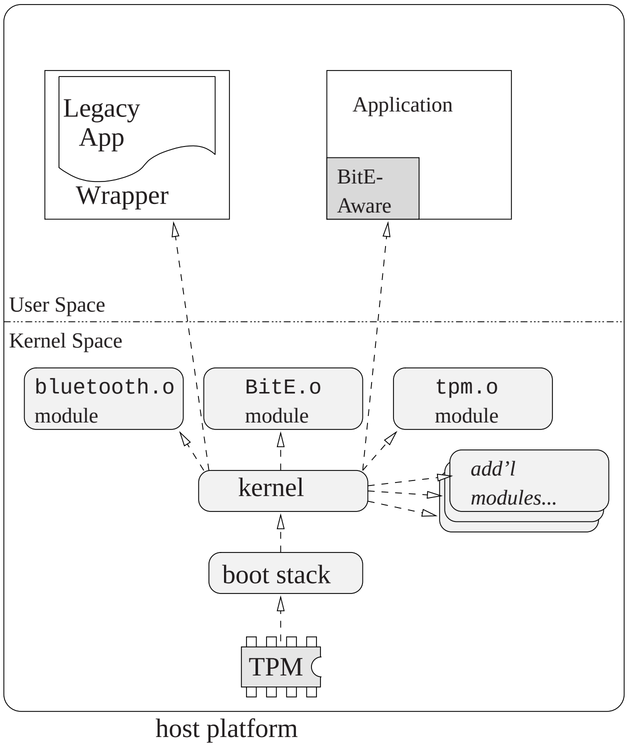


Figure 2: TCG-enabled host platform integrity measurement architecture (IMA). The dashed lines denote *integrity measurement* as described by Sailer et al. [31].

3 Architecture

The essence of BitE is end-to-end encrypted, authenticated tunnels between a trusted mobile device and a particular application on the user's TPM-equipped host platform. Trusted tunnels use per-application cryptographic keys which are established during an application registration phase. The process of establishing a trusted input session over the trusted tunnel is contingent on the mobile device's successfully verifying an attestation from the IMA and TPM on the user's host platform.

The user's trusted mobile device must be capable of establishing a secure connection to the user's input device (i.e., keyboard). This can take the form of a physical connection or authenti-

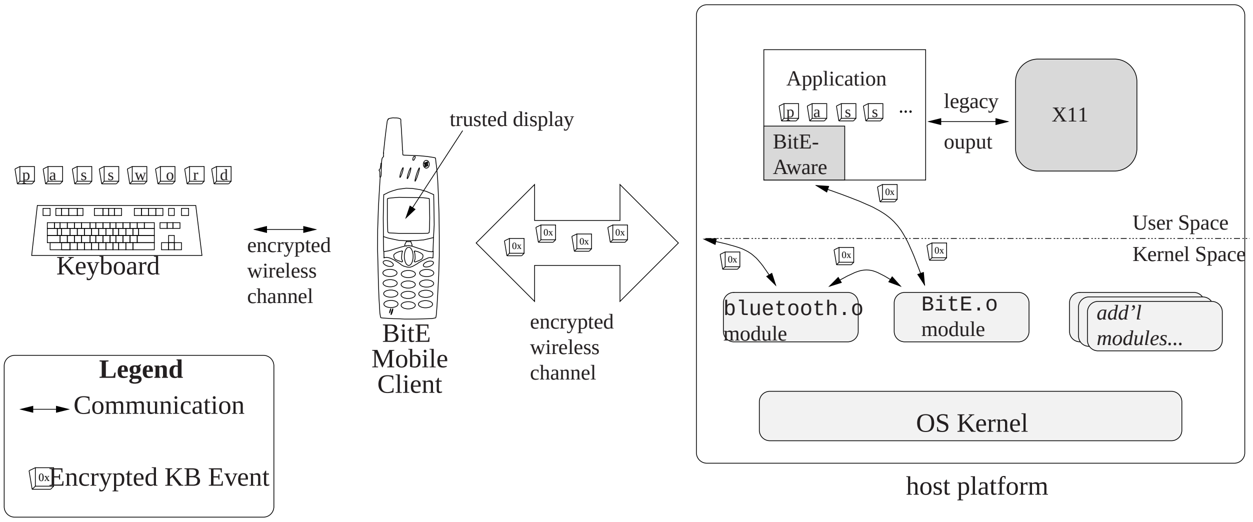


Figure 3: BitE system architecture. The user presses keys (e.g., types a password) on the keyboard. The keypress events are sent over an encrypted channel to the BitE Mobile Client. The BitE Mobile Client re-encrypts the keyboard events with a cryptographic key that is specific to some application. On the host platform, the encrypted keyboard events are passed from the `bluetooth.o` kernel module to the BitE Kernel Module, and then to the application, where they are decrypted.

cated, encrypted communication over a wireless link. The mobile devices we consider have the following properties:

- A display.
- Persistent storage capable of storing cryptographic keys.
- Sufficient computational power to compute asymmetric cryptographic functions (e.g., digital signatures).
- Wireless network interface(s) capable of simultaneously[3] communicating with I/O devices (e.g., Bluetooth keyboard and mouse [7]) and the host platform.

The mobile phone's display serves as a trusted output channel to the user. This enables us to minimize the amount of trust we place in the window manager on the host platform.

To minimize the burden on the user, we envision two wireless connections with the mobile device: one connection to the input device and another connection to the user's host platform. We assume that a secure (authenticated and encrypted) association between the user's wireless keyboard and her mobile phone can be established. For the remainder of the paper, we consider the user's input devices (e.g., wireless keyboard and mouse) as extensions of her mobile phone. Thus, subsequent discussions will focus on interaction between the the user, her mobile phone, and her host platform.

[3] Fast enough so that the user does not notice additional input latency.

For our prototype (which we describe in Section 7), we used a Nokia 6620 as the trusted device and Bluetooth and infrared as the wireless network interfaces. As mobile phones have become ubiquitous, we assume the use of a mobile phone as the trusted device for the remainder of the paper.

Mobile devices are continuously increasing in complexity and thus feature software vulnerabilities of their own. Efforts are ongoing to improve the security of mobile devices, augmented by the experience gained working to secure more traditional platforms [38]. We consider the consequences of various compromises in Section 6.

We must trust the application and kernel on the host platform with which the user wishes to establish a trusted tunnel for input. One reason we must trust the OS kernel is because of its ability to arbitrarily read the memory space of any process executing on the system—we cannot trust an application without also trusting the kernel on which it runs. Despite this trust, it is desirable to minimize the involvement of the kernel in the handling of user input. For example, consider the infrequency of situations in which user input is actually destined for the kernel. In most Linux distributions, Ctrl+Alt+Del is intercepted by the kernel and passed to the `init` process. In the Windows NT family, Ctrl+Alt+Del is received by the GINA[4] DLL[5]. We further investigate the reasons why a trusted kernel is insufficient for secure input in Section 6.

In the next section, we describe BitE in detail. For ease of exposition, we describe our system using Linux and X11 terminology [41, 42]. However, our techniques can be applied in other operating environments, e.g., Microsoft Windows.

4 Bump in the Ether – Detailed Design

Bump in the Ether (BitE) is built around a trusted mobile device that can proxy input, show data on its own display, and perform asymmetric and symmetric cryptographic operations efficiently. This trusted device runs a piece of software called the BitE Mobile Client. The BitE Mobile Client communicates with the BitE Kernel Module, which is loaded on the host platform. The software required to leverage BitE also includes application extensions (for BitE-aware applications) and wrappers (for legacy applications).

Figure 3 shows the main components of BitE. The user types on a wireless keyboard which communicates via an authenticated, encrypted channel with a BitE Mobile Client running on that user's mobile phone. The BitE Mobile Client simultaneously establishes a second authenticated, encrypted channel with the BitE Kernel Module loaded on the host platform.

The BitE Kernel Module manages per-application cryptographic keys which it keeps in TPM-protected sealed storage [39], denoted K_{App_i} for application i. These keys are shared with the BitE Mobile Client (i.e., the keys are simultaneously stored by the BitE Mobile Client on the user's mobile phone). The per-application keys are established during application registration, which we describe in Section 4.2.

The trusted tunnel for input of sensitive information is an encrypted, authenticated tunnel constructed with session keys. BitE-aware applications obtain session keys (e.g., keys for encryption

[4] Graphical Identification aNd Authentication
[5] Dynamic Link Library

and MAC[6]) from the BitE Kernel Module. The session keys are derived from the per-application keys using standard protocols (e.g., [10, 18, 26]). The BitE Mobile Client uses the session keys to encrypt and MAC the actual keyboard events such that they can be authenticated and decrypted by the BitE-aware application in an end-to-end fashion. We now describe the operation of BitE in detail.

4.1 Initial Cryptographic Key Setup

BitE requires that several cryptographic keys be setup correctly to properly protect user input. The BitE Mobile Client and the BitE Kernel Module must be able to mutually authenticate, from which encrypted channels can be bootstrapped using standard protocols (e.g., [10, 18, 26]). The necessary keys for authentication can be setup using location-limited channels [3, 22, 36], or—due to the infrequent need for initial key setup—we can assume that the initial configuration occurs in the absence of malicious activity (as SSH does today). We use the notation $\{K_{module}, K^{-1}_{module}\}$, and $\{K_{phone}, K^{-1}_{phone}\}$ for the asymmetric (e.g., RSA) keypairs for the BitE Kernel Module and the BitE Mobile Client, respectively.

Additionally, the BitE Mobile Client must be equipped with the public *Attestation Identity Key* from the host platform, denoted AIK. AIK is required by the BitE Mobile Client to verify the signature on attestations from the TPM in the host platform. AIK can be sent to the BitE Mobile Client signed by K^{-1}_{module}, since the phone can verify the signature with K_{module}.

4.2 Application Registration

Each application with which the user desires to be able to establish a trusted tunnel must be registered with the BitE Kernel Module and the BitE Mobile Client. The user performs an initial execution of the application to be registered. The IMA automatically measures this application and its library dependencies[7] and stores them in the IMA measurement list (see [31] for details). We assume the system state can be trusted during application registration (i.e., there is no malicious code executing).

The BitE Kernel Module generates a symmetric key K_{App_i} (for application i) to be used in subsequent connections for the derivation of encryption and MAC session keys for establishing the trusted tunnel. The IMA measurement for application i, the newly generated symmetric key K_{App_i}, and the user-friendly name of the registered application (e.g., *Mozilla Firefox*), are sent over a mutually authenticated, encrypted channel (established using K_{module} and K_{phone}) to the BitE Mobile Client, where they are stored for future use. The role of this data in establishing the trusted tunnel is detailed in the next section.

[6] Message Authentication Code, e.g., HMAC [4].

[7] Other dependencies may exist that we wish to measure. For example, configuration files can have a significant impact on application security. Automatic identification of configuration files associated with a particular application is complex, and beyond the scope of this paper.

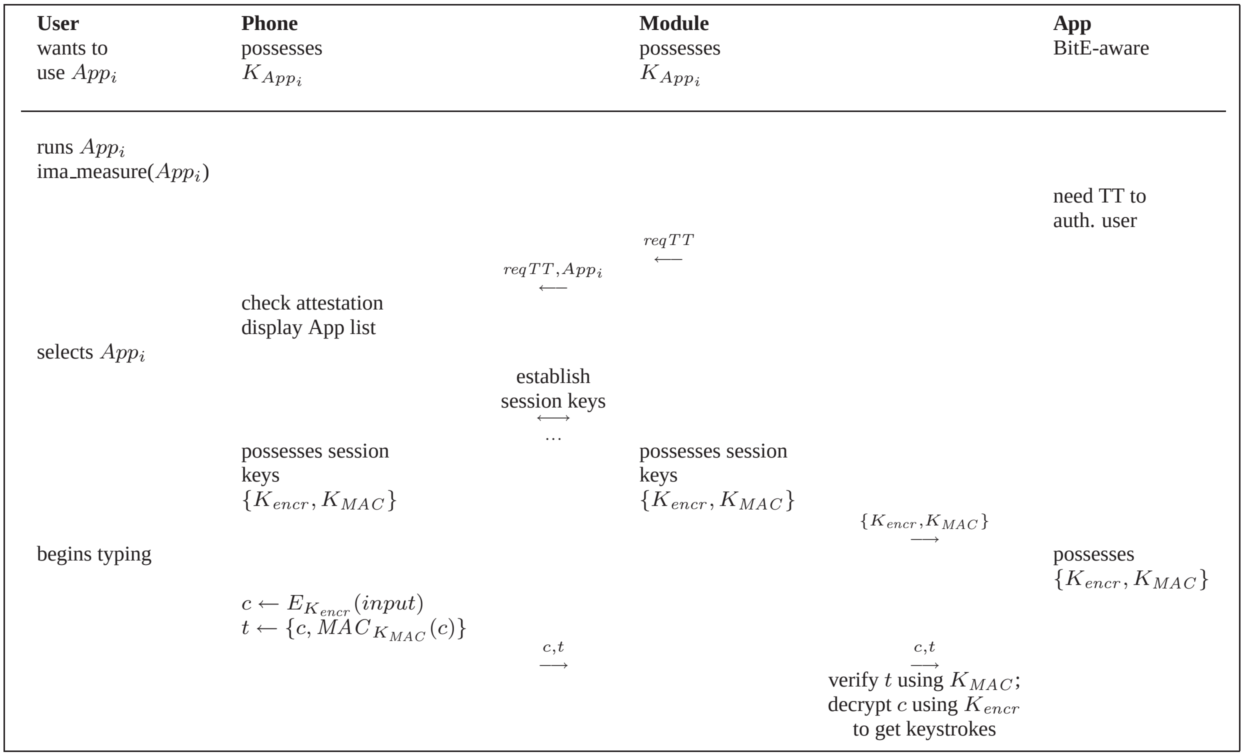


Figure 4: Example application execution and trusted tunnel establishment with registered BitE-aware application so that the user can enter sensitive information to that application. In this figure we assume the wireless keyboard is an extension of the phone, so we do not show the keyboard. We also assume key setup (Section 4.1) and application registration (Section 4.2) have already been successfully completed. TT stands for Trusted Tunnel.

4.3 Trusted Tunnel Setup

This section describes the setup of the BitE trusted tunnel to allow the user to securely send input to registered applications. Establishment of a trusted tunnel is initiated by a BitE-aware application when it requires sensitive input from the user. We discuss extensions to allow the user to manually initiate a trusted tunnel in Section 6. We first describe normal operation when a single registered application requests a trusted tunnel, then detail conflict resolution which must be performed when multiple applications request a trusted tunnel at or near the same time.

When a BitE-aware application requires security-sensitive input (e.g., passwords or credit card numbers), it sends a message to the BitE Kernel Module to register an input-event callback function implemented by the BitE-aware portion of the application. If the BitE Kernel Module has no other outstanding requests, it begins the process of establishing the trusted tunnel. This process has three steps: the host platform must attest the state of its software to the BitE Mobile Client (Section 4.3.1); the user must interact with the BitE Mobile Client (Section 4.3.2); and session keys must be established for creation of the actual tunnel (Section 4.3.3).

4.3.1 Attestation

The BitE Mobile Client must verify an attestation of the currently running software on the user's host platform (recall Section 2.2). The BitE Mobile Client can verify the signature on the attestation with its authentic copy of the public AIK, and it can then verify the measurement list is consistent with the signed PCR values. It can then compare the measurement values with those present during application registration (Section 4.2). The measurements of interest to the BitE Mobile Client are those of the boot stack up through the loading of the kernel, its modules, deterministic system services, and the measurement of the application requesting the trusted tunnel. If the values match, we consider the host platform to have successfully attested its software state to the BitE Mobile Client. The user now has assurance that the boot process up through the loading of the kernel, its modules (notably the BitE Kernel Module), and deterministic system services happened in exactly the same way that it did during application registration, and that the same version of her application is running that was running during application registration.

Note that the software state of a host platform is comprised of *all* software loaded for execution, and that verification of the boot stack up through the loading of the kernel, its modules, and deterministic system services, and the application requesting the trusted tunnel, leaves room for unknown user-level software to execute. It is this unknown user-level software against which the trusted tunnel offers protection. We discuss this further in Section 6.1.

4.3.2 User – Phone Interaction

Upon verifying the attestation from the BitE Kernel Module, the BitE Mobile Client has assurance that the correct application is executing. Before session keys can be established to form the trusted tunnel, it is necessary to involve the user via her mobile phone to ensure that the application with which the user intends to interact and the application asking for her input are the same. This property can be challenging to achieve without annoying the user. A viable solution is one that is easy to use, but not so easy that the user "just hits OK" every time.

Our solution is to display a list of registered applications on the BitE Mobile Client. The user must scroll down (using the arrow keys on her keyboard, or the navigational buttons on the phone itself) and then select (e.g., press enter) the correct application. Note that since all input from the user's keyboard passes through the mobile phone the user does not actually need to press buttons on her phone. The phone will interpret the user input from her keyboard appropriately. Refer to Figure 4 for more information on the interaction between the user and her phone.

We are concerned about users developing habits that might increase their susceptibility to spoofing or phishing attacks. Thus, we randomize the order of the list so that the user cannot develop a habit of pressing, e.g., "down-down-enter," when starting a particular application that requires a trusted tunnel. Instead, the user must actually read the list displayed on her phone and think about selecting the appropriate application. We believe selection from a randomized list achieves a good balance between security and usability, provided that the length of the list is constrained (e.g., it always fits on the phone's screen).

Once the list is displayed, the BitE Mobile Client signals the user—e.g., by beeping. This serves two purposes: (1) to let the user know that a secure input process is beginning; and (2) to let the user know that she must make a selection from choices on the phone's screen. Item (2) is necessary because a user may become confused if her application seems unresponsive when in reality the BitE Mobile Client on her mobile phone is prompting her for a particular action.

Note that a look-alike (e.g., Trojan, phishing attack) application will be unable to get the mobile phone to display an appropriate name, because the look-alike application was never registered with the BitE system (recall Section 4.2). Only applications that were initially registered are options for trusted tunnel endpoints.

If the user is satisfied, she selects the option given by her phone corresponding to the name of the application with which she wants to establish a trusted tunnel. If she suspects anything is wrong, she selects the *Abort* option given by her phone. It is an error if the user selects any application other than the one which is currently requesting a trusted tunnel. That is, the BitE Mobile Client will report an error to the user (the application she selected from the list is not the same application that requested a trusted tunnel). It is a policy decision to decide how to handle this type of error. One approach is to fail secure, and prevent the user from entering sensitive input into her application until a successful retry.

Variations on this user interface that might also be effective in practice are discussed in Section 6.3.

4.3.3 Session Keys

Session keys must be established which will be used to encrypt and authenticate keyboard events from the BitE Mobile Client to the BitE-aware application, denoted K_{encr}, K_{MAC}. This process is similar to that performed by SSH[8]. Depending on the structure of an actual implementation, it may also be necessary to incorporate keystroke timing attack countermeasures, e.g., [35, 37].

Session key establishment depends on the BitE Kernel Module and the BitE Mobile Client establishing mutually authenticated communications over which session key establishment can proceed (see Section 4.1). Session keys are derived from the application key, K_{App_i}—which was generated during application registration—using standard protocols (e.g., [10, 18, 26]).

Figure 4 presents step-by-step details on the process of input via the BitE trusted tunnel. Once the trusted tunnel is established, the user can input her sensitive data. After this data has been input, the application notifies the BitE Kernel Module that it is finished receiving input via the trusted tunnel. At this point, the BitE Kernel Module tears down the encrypted channel from the BitE Mobile Client to the application, and reverts to listening for requests for trusted tunnels from other registered applications.

4.3.4 Handling Concurrent Trusted Tunnels to Prevent User Confusion

While there are no technical difficulties involved in maintaining multiple active trusted tunnel connections from the BitE Mobile Client to applications, there are user-interface issues. We know of no way to disambiguate to the user which application is receiving input without requiring excessive user diligence. For example, a naive solution is to display the name of the application for which user input is currently being tunneled on the mobile phone's screen. This requires the user to look at the screen of her mobile phone and ensure that the name matches that of the application with which she is currently interacting.

[8] Secure SHell, `http://www.openssh.com/`.

To prevent user confusion, we force the user to interact with one application at a time in a trusted way. If we allow users to rapidly switch applications (as today's window managers do), then the binding of user intent with user action is dramatically weakened. The rapid context switching makes it easy for the user to become confused and enter sensitive input into the wrong application. An adversary may be able to exploit this weakness.

We consider two example applications which we assume to be BitE-aware and that require a trusted tunnel for user authentication:

1. Banking software which requires the user to authenticate with an account number and a password.

2. A virtual private network (VPN) client which requires the user to authenticate with a username and a password.

Suppose the user needs to interact with both applications at the same time, for example, to compare payroll information from her company with entries in her personal bank account. In today's systems, there is nothing to cause the user to serialize her authentication to these applications. She may start the banking software, then start the VPN client, then authenticate to the banking software, then authenticate to the VPN client. In the BitE system, assuming the banking software and VPN client are BitE-aware, the BitE Kernel Module considers this behavior to be a concurrent request by two applications for establishment of a trusted tunnel.

It is a policy decision how to handle concurrent trusted tunnel requests. One option is to default-deny both applications, and alert the user to the contention. She can then retry with one of the two applications, and use it first. This forces the user to establish a trusted tunnel to the first application and fully input her sensitive data to that application. Once her data is input, the first application will relinquish the trusted tunnel, and it will be torn down by BitE. The user can then begin the process of entering her sensitive data to the second application, which will entail the establishment of another trusted tunnel. These one-at-a-time semantics may induce some additional latency for the user before she can begin using her applications, but we consider this to be an acceptable tradeoff in view of the gains in security.

4.4 BitE-Unaware (Legacy) Applications

We now describe BitE operation with an application that is *unaware* of the BitE system. That is, this section describes how BitE is backwards-compatible with existing applications. Legacy applications were written without knowledge of BitE, so there is no way for a legacy application to request a trusted tunnel. Hence, all input to a legacy application must go through a trusted tunnel. The basic idea is that we run legacy applications inside a *wrapper* application (the BitE-wrapper) that provides input events to that application (e.g., `stdin` or X keyboard events).

The legacy application gets measured by the IMA and registered with BitE in the same way the BitE-aware applications do. If the application changes after its initial registration, the BitE Kernel Module will not release the session keys necessary to decrypt and authenticate keyboard events. The most challenging part of interacting with a legacy application is that it contains no BitE-aware component that can handle the decryption and authentication of keyboard events. Instead, the BitE-wrapper does the decryption and authentication of keyboard events. It is necessary to prevent the legacy application from receiving keyboard events from the window manager (or other user-level processes), while allowing it to receive input from the wrapper application. This is easy to achieve for console applications (e.g., just redirect `stdin`); however, it is challenging for graphical applications. We now consider the necessary BitE-wrapper functionality for X11 applications.

X11 applications (*clients* in the context of X) register to receive certain types of event notifications from the X server. Common event types include keyboard press and release events. Applications register to receive these events using the `XSelectInput` function. The BitE-wrapper application can intercept this call for dynamically linked applications using the `LD_PRELOAD` environment variable. With `LD_PRELOAD` defined to a custom BitE shared library, the run-time linker will call the BitE `XSelectInput` instead of the X11 `XSelectInput`. Thus, the BitE Kernel Module has hooked into the application's input event loop. The BitE Kernel Module can generate its own input events to send to the application simply by calling the callback function the application registered in its call to `XSelectInput`.

5 Security Analysis

In this section we analyze the security of BitE. During the design of BitE, we tried to make it difficult for the user to make self-destructive mistakes. For example, the BitE Mobile Client will not allow the user's keystrokes to reach the BitE Kernel Module if verification of an attestation fails. The user must respond to messages displayed on her mobile device before she can proceed. Security mechanisms on the critical input path cannot go unnoticed by the user. These mechanisms must provide a tangible benefit with a value commensurate with the difficulty of using them.

We provide some examples of attacks that BitE is able to protect against. We then consider the failure modes of BitE when the assumptions upon which it is constructed do not hold.

5.1 Stopped Attacks

We consider multiple scenarios where the use of BitE protects the user.

Capturing Keystrokes with X Giampaolo shows how easy it is for an attacker to use a malicious application to capture the keystrokes the user intends to go to the active (and assumed benign) application [19]. If the user is using BitE to enter sensitive data; however, this attack does not work (see Figure 1). The user's keystrokes are encrypted and authenticated with session keys (as discussed in Section 4.3) which are unavailable to the malicious application. Hence, the encrypted keystrokes reach the user's desired application unobserved.

Hardware Keyloggers Hardware keyloggers are becoming a significant threat. Sumitomo Bank in London was the victim of a sophisticated fraud scam involving hardware keyloggers [32]. With BitE, the user's keystrokes travel inside encrypted, authenticated tunnels. Even if an adversary can capture the ciphertext, he will be unable to extract the keystrokes (assuming the relevant cryptographic primitives are secure).

Bluetooth Eavesdropping BitE is most convenient for the user when wireless communication mechanisms can be used. As long as the initial exchange of public keys between the BitE Mobile Client and the BitE Kernel Module proceeds securely (using, e.g., [3, 22, 36]), all communication between them can be encrypted and authenticated using standard protocols. If keystroke timing attack countermeasures are incorporated (e.g., [35, 37]), timing side-channels can also be eliminated. Since all communication is strongly authenticated, an adversary will not be able to masquerade as a valid BitE Mobile Client or BitE Kernel Module.

Modification of Registered Applications An attacker may be able to modify (e.g., by exploiting a buffer overflow vulnerability in a different application) the binary of a registered application. Such an attack may modify the application's executable such that it may log user input to a file, or send it via email to a malicious party on the Internet. With BitE, an IMA measurement of the executable was recorded during initial application registration (recall Section 4.2). The modified application binary will be detected during trusted tunnel setup when the BitE Mobile Client tries to verify the attestation from the host platform. The BitE Mobile Client will alert the user that the application has been modified.

Kernel Modification A measurement of the kernel binary is part of the integrity measurement which is verified when a trusted tunnel is established. Modification of the kernel image on disk will be detected after the next reboot. As a disk-only modification of the kernel image will not affect the running system until a reboot, the attack is detected by the BitE Mobile Client before it can affect the operation of BitE.

Kernel Rootkits Kernel rootkits often modify one or more operating system daemons for malicious use. Typically, a vulnerability in one daemon is used to replace the binaries of several daemons. As soon as the replacement daemons are executed, the integrity measurements for their respective binaries will change. The BitE Mobile Client will reject the attestation from the host platform during the next attempt to establish a trusted tunnel.

5.2 Failure Modes

We now describe what happens if the assumptions upon which the security of BitE is based turn out to be invalid. Specifically, we discuss the extent to which the failure of our assumptions permit the attacker to perform one or more of the following:

- To observe keystrokes in one ongoing session.
- To observe keystrokes in current and future sessions.
- To register applications of his own choosing.

Compromise of Active Application If the attacker is able to compromise an application while the user has a trusted tunnel established, he may be able to observe the user's keystrokes. This break is limited to the compromised application, however, as the attacker has no way to access keys established between the BitE Kernel Module and other registered applications. This break is feasible because the adversary is exploiting a time-of-measurement, time-of-use (TOMTOU) limitation of the integrity measurement architecture (e.g., a buffer overflow attack).

Compromised Mobile Phone Since the mobile phone is used as a central point of trust in our system, its compromise will allow an attacker to access all keyboard events. The attacker will have possession of $\{K_{phone}, K_{phone}^{-1}\}$ so he may be able to masquerade as a trusted BitE Mobile Client using an arbitrary device (e.g., one with a very powerful radio transmitter). Further, the attacker will capture all registered applications' unique keys, K_{App_i} for application i, and user-friendly name. This will enable the attacker to establish trusted paths with registered applications, and it will allow the attacker to register new applications.

Note that the TCG is currently working on trusted platform standards for mobile devices [38], which may be able to minimize the severity of a mobile phone compromise. For example, the BitE Mobile Client could store its secrets in sealed storage, rendering them inaccessible to malicious software installed on the phone by the adversary.

Compromise of Active Kernel on Host Platform If the operating system kernel on the host platform is compromised without rewriting a measured binary (e.g., exploiting TOMTOU limitations with a buffer overflow attack), the attacker may be in a position to capture sensitive user input despite the BitE system. If the host platform is not TCG-compliant, there is little protection against an attacker with superuser privileges trying to read the BitE Kernel Module's secrets. This gives the attacker access to $\{K_{module}, K_{module}^{-1}\}$ and to the unique application keys K_{App_i}. The attacker can also capture keystrokes from ongoing sessions by reading the session keys out of the memory space of the BitE Kernel Module or the application.

6 Discussion

In this section we discuss additional issues that arise while using the BitE system. These issues include verification of a subset of integrity measurements (as opposed to all measurements); alternative system architectures (elimination of the trusted mobile device or TPM); and alternative user interface designs for the BitE system. We also perform a comparison of BitE with Microsoft's NGSCB.

6.1 Unknown Software Measurements

In Section 4.3, we discussed how the BitE Mobile Client verifies the integrity measurements of the boot stack up through the loading of the kernel, its modules, deterministic system services, and the application requesting the trusted tunnel. It is important to note that this leaves room for unknown software to execute—the true software state of a host platform is comprised of *all* software loaded for execution. Today's host platforms load and run unknown software frequently, and requiring the user to manage all possible executable content is intractable. However, we can distinguish between two periods of integrity measurement: (1) the boot process up through the loading of the kernel, its modules, and deterministic system services (e.g., `syslogd` and `sshd`), which changes infrequently; and (2) the software loaded interactively by the user, which is constantly changing. Verifying that the boot process up through the loading of the kernel, its modules, and deterministic system services has not been altered gives the user a significantly higher probability of detecting root-level compromises of her host platform. Verifying that the same application is running that was initially registered allows the BitE Kernel Module to release session keys to that application only. In other words, BitE is not a panacea, but it raises the bar for attackers considerably.

6.2 Alternative System Architectures

The BitE system as presented in this paper is designed around a TPM-equipped host platform and a trusted mobile device. We briefly consider alternative design approaches, namely, designs that eliminate the mobile device or the TPM. It is particularly tempting to think that one or both of these requirements may be unnecessary since we include the OS kernel in the TCB for BitE. In addition to its role as resource manager, we must trust the OS kernel because of its ability to arbitrarily read the memory space of any process executing on the system.

6.2.1 Eliminating the Mobile Device

A significant challenge addressed by BitE is for the user to obtain a user-verifiable property of the integrity of the OS and application. It is important to distinguish between two of the many roles the mobile device fulfills:

- Verification of integrity measurements from the host platform.
- Trusted visual output to the user.

Integrity measurement on the host platform puts in place the facility for verification by an external entity, but the host platform cannot "self-verify." In BitE, the mobile device fulfills the role of the verifying entity. The mobile device must have trusted visual output to the user so it can appropriately notify the user of the success or failure of this verification.

6.2.2 Trusting the Window Manager

Without the trusted mobile device to securely display information to the user, the window manager must perform this function (and hence become part of the TCB). We return to the discussion of trusted windowing systems from Section 2.1. Recall that the most common trusted-output mechanism is based on a dedicated area of the screen being reserved for output from "trusted" processes only. Despite several decades of effort, trusted windowing systems are not readily available for commodity platforms. We emphasize that BitE will work today, on some of today's most prevalent platforms.

We believe one reason window manager-reserved trusted screen areas are not common-place is because their existence complicates the semantics of maximizing applications or using full-screen mode. If an application can put the system into full-screen mode, it may be able to spoof the trusted output area. Precisely defining trusted full-screen semantics that a non-expert user can operate securely is, to the best of our knowledge, an unsolved problem.

Considering the value that the user receives from being able to maximize applications, and the role of multi-media applications on today's commodity PCs, we believe the ability to run applications in full-screen mode on the system's primary display is an indispensable feature. BitE enables the user to interact securely with applications even when they run in full-screen mode.

6.2.3 Input Proxying by the Mobile Device

With a trusted OS kernel, there is no technical reason why user input must travel through the mobile device. The BitE Kernel Module already possesses copies of the cryptographic keys shared with the application, and it could encrypt user input from the traditional keyboard driver before it passes through the window manager. However, this design raises an important usability issue. The mobile device must be on the critical input path so that it can ensure that the user cannot proceed unaware of a failed integrity verification. We are concerned that non-expert users will proceed to interact with their application even if a message appears on the mobile phone stating that the system is compromised. With the mobile device on the critical input path, it can stop user input from reaching the application while also providing secure feedback to the user.

6.2.4 TPM Alternatives

Finally, we discuss the extent to which the TPM is an essential requirement for BitE. We could leverage software-based attestation mechanisms to verify the authenticity of the OS [33]. However, in open computing environments, it may be challenging to satisfy the assumptions underlying these techniques, e.g., that the verified device cannot communicate with a more powerful computer during the attestation process.

6.3 Alternative User Interfaces

An important property achieved by the BitE system is that the user selects the registered application with which she would like to establish a trusted tunnel from a list on her trusted mobile phone's screen. We present two alternative operating models that may be more convenient for the user, though they tradeoff the strength of the resulting trusted tunnel.

6.3.1 Active Selection

As described so far, BitE requires the user to select an application from a list presented by the BitE Mobile Client when a registered application requests a trusted tunnel for input. An alternative structure, and one that may be preferable for legacy applications, is one where the user directs the BitE system to establish a trusted tunnel. Recall that the trusted tunnel is established in response to a request from a BitE-aware application (Section 4.3). One possibility for giving the user this ability is to maintain a list of all registered applications on the BitE Mobile Client. When the user wants to send secure input to, e.g., *Mozilla Firefox*, she selects "Mozilla Firefox" from the list on her phone.

This system has advantages for legacy applications since they do not actively request a trusted tunnel for input. BitE as described in Section 4.4 requires a legacy application to receive *all* input through the trusted tunnel, which may be inconvenient for the user if she wishes to interact with a second application with a trusted tunnel while her legacy application is still running (and using the trusted tunnel).

The drawback of this *Active Selection* scheme is that it requires user diligence. We are concerned about users forgetting to manually enable the trusted tunnel when they input sensitive data.

6.3.2 Always-On-Top

Another possible configuration for a trusted tunnel system is one where the window manager is involved. In this scenario, the BitE Mobile Client and the BitE Kernel Module maintain multiple active sessions. The user's typing goes to whatever application the window manager considers to be "on top." This configuration for BitE is problematic since the window manager becomes a part of the TCB. Much of the motivation for BitE is that it is able to remove the window manager from the TCB for trusted tunnel input.

6.4 Comparison with Microsoft NGSCB

We now compare BitE with a well-known host-only solution—the trusted input capabilities of Microsoft's Next-Generation Secure Computing Base (NGSCB) [24]. NGSCB includes a design for encrypted I/O. In NGSCB, special USB keyboards encrypt keystrokes which pass through the regular operating system into the *Nexus*, where they are decrypted. Once in the Nexus, they can be sent to a trusted application running in Nexus-mode, or they can be sent to the legacy OS. Applications running in Nexus-mode have the ability to take control of the system's primary display, which could be useful for establishing a trusted tunnel. However, the establishment of a trusted tunnel depends on the user observing visual cues on the primary display. We are concerned that a clever attacker can easily fool a regular user into believing they are interacting with a trusted application when they are not. As Ye and Smith demonstrate, there is no effective way to establish a trusted tunnel if there is no trusted display [43, 44].

With BitE, the user identifies an application by interacting with the display on her mobile phone, which is physically distinct from the host's primary display. Sensitive keystrokes are encrypted by the mobile phone under a key shared with that particular application. Encrypted keystrokes are sent directly to the registered application, completely avoiding use of the window manager for user input. BitE provides end-to-end encryption from the keyboard through user's mobile phone to the registered application. Further, all the requirements for BitE can be met with commodity devices today (as we show in our evaluation in Section 7).

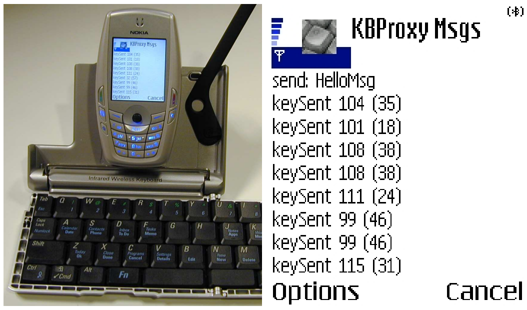
7 Prototype and Evaluation

In this section, we describe our prototype and evaluate some practical considerations necessary for actually building BitE. We have developed a prototype J2ME MIDP 2.0 [25] BitE Mobile Client that runs on the mobile phone. It receives keystrokes via an infrared keyboard and sends them to a prototype BitE Kernel Module loaded on the host platform via Bluetooth. Our prototype is shown in operation in Figure 5. Our prototypes use the BouncyCastle Lightweight Cryptography API[9] for all cryptographic operations.

The BitE Mobile Client consists of less than 1000 lines of Java code, not including source code from libraries. The BitE Kernel Module consists of approximately 500 lines of C and Java code which interacts with the BitE Mobile Client via Bluetooth, the legacy input system via the

[9] `http://www.bouncycastle.org/`

`uinput` kernel module, and the integrity measurement architecture of Sailer et al. [31] via the `/proc` filesystem.



KBProxy Msgs

```
send: HelloMsg
keySent 104 (35)
keySent 101 (18)
keySent 108 (38)
keySent 108 (38)
keySent 111 (24)
keySent 99 (46)
keySent 99 (46)
keySent 115 (31)
```

(a) Phone closeup.

(b) Debugging screenshot captured on the phone.

Figure 5: Debugging screenshot showing the BitE prototype in operation.

7.1 Encrypted Channel Setup Latency Between Phone and Host Platform

We have performed experiments to determine the overhead associated with asymmetric cryptographic operations necessary to establish encrypted, authenticated communication between a mobile phone and host. We used a J2ME MIDP 2.0 application running on a Nokia 6600, and a J2SE application running on the host platform.

Establishing mutually authenticated communication involves performing asymmetric signature and verification operations at both communication endpoints. In our experiments with 1024-bit RSA keys (see Table 1), signing operations take 2546 ms on average. Signature verification averages 82 ms. Thus, mutual authentication on the Nokia 6600 will take on the order of 4 to 5 seconds, which is a noticeable but tolerable delay. This is because these asymmetric operations are only required for communication setup. Once session keys are established, efficient symmetric primitives can be used for communication.

7.2 Keyboard – Phone Communication

We experimented with an infrared keyboard to provide user input to the BitE Mobile Client, and a Bluetooth connection from the BitE Mobile Client to the host platform. This is because our development phones (Nokia's 6600 and 6620) can support only one Bluetooth connection at a time. The use of an infrared keyboard is undesirable because the keystrokes are transmitted in the clear. A better solution is to use a keyboard that physically attaches to the mobile phone. Alternatively, a Bluetooth keyboard capable of authenticated, encrypted communication can be used. We are unaware of any mobile phones available today that support more than one active Bluetooth connection simultaneously, but such devices may become available in the near future.

We performed simple usability experiments to analyze keystroke latencies, to ensure that BitE is not rendered useless by excessive latency while typing. We observe no noticeable latency with debug logging disabled. Our prototype does not yet encrypt the keystrokes it sends, but the use of symmetric cryptographic primitives introduces minimal overhead per keystroke. For example, the use of counter-mode encryption could enable the BitE Mobile Client to precompute enough of the key stream so that the only encrypt / decrypt operation on the critical path for a keystroke is an exclusive-or [23]. This reduces the most significant cryptographic per-keystroke operation to the verification of a MAC (e.g., HMAC with SHA-1, [4, 5, 21]). Our experiments below on verifying integrity measurements indicate that SHA-1 operations can be performed efficiently on the class of mobile phones we consider.

7.3 Verifying Attestations on the Mobile Phone

We have the integrity measurement architecture (IMA) of Sailer et al. [31] running on our development system. We have implemented the operations necessary for the BitE Kernel Module to send the IMA measurement list to the BitE Mobile Client, and for the BitE Mobile Client to compute the PCR aggregate for comparison with a signed PCR quote from the TPM. Table 1 shows some performance results for our prototype on a Nokia 6620. In order to validate an attested set of measurements from the host platform, the measurement list must be hashed for comparison with the signed PCR aggregate. For a typical desktop system, this involves hundreds of hash operations. Our experiments show that the average time necessary to compute a SHA-1 hash of 401 measurements (the number of measurements our development system had performed at the time of the experiment), 171 ms, is dominated by the time necessary to manipulate the data from the measurement list—2087 ms. In this experiment, the Nokia 6620 is bound by memory access and not CPU operations. Our results show that the expected time for a Nokia 6620 to verify an attestation (check the signature on the aggregate values from the PCRs, hash the measurement list, and compare the aggregate hash from the measurement list to the appropriate PCR value) is approximately 3 seconds. We feel that this is reasonable, considering that significantly faster phones are already available (unfortunately, we did not have one to experiment with).

Action	Time (ms)	Variance
RSA PSS (sign)	2546	343
RSA verify	82	251
SHA-1 aggregate	171	110
Data manipulation	2087	703

Table 1: Average time (in milliseconds) to perform an RSA signature and verification with a 1024-bit key; compute an aggregate hash of 401 SHA-1 measurements from a Debian workstation running Linux kernel 2.6.12.5; and manipulate the measurement list data. The RSA experiments were performed on a Nokia 6600; the SHA-1 and data manipulation experiments were performed on a Nokia 6620.

8 Conclusions

This work addresses the design and analysis of Bump in the Ether (BitE), a system that uses a mobile phone as a trusted proxy between a keyboard and a TCG-compliant host platform to establish a trusted tunnel for user input to applications. The resulting tunnel is an end-to-end trusted tunnel all the way from a user's mobile phone to an application running on the host platform. BitE places specific emphasis on these design issues: (1) malware such as keyboard sniffers, spyware, Trojans, and phishing attacks running at user level will be unable to capture the user's input; (2) operation of the system is convenient and intuitive for users; (3) the BitE system is feasible today on commodity hardware; and (4) the BitE system still offers some protection for legacy applications.

References

[1] Trusted Computing Group. `http://www.trustedcomputinggroup.org/`, August 2005.

[2] B. Balacheff, L. Chen, S. Pearson, D. Plaquin, and G. Proudler. *Trusted Computing Platforms – TCPA Technology in Context*. Prentice Hall, 2003.

[3] D. Balfanz, D.K. Smetters, P. Stewart, and H. C. Wong. Talking to strangers: Authentication in ad-hoc wireless networks. In *Proceedings of the Symposium on Network and Distributed Systems Security (NDSS)*, February 2002.

[4] M. Bellare, R. Canetti, and H. Krawczyk. Keyed hash functions and message authentication. In *Proceedings of Crypto*, pages 1–15, 1996.

[5] M. Bellare, T. Kohno, and C. Namprempre. SSH transport layer encryption modes. Internet Draft; Available at: `http://www.ietf.org/internet-drafts/draft-ietf-secsh-newmodes-05.txt`, August 2005.

[6] J. L. Berger, J. Picciotto, J. P. L. Woodward, and P. T. Cummings. Compartmented mode workstation: Prototype highlights. *Software Engineering*, 16(6):608–618, June 1990.

[7] Bluetooth Special Interest Group. Human interface devices working group. `https://www.bluetooth.org/bluetooth/landing/wg_hid.php`, May 2005.

[8] M. Carson and J. Cugini. An X11-based Multilevel Window System architecture. In *Proceedings of the EUUG Technical Conference*, Nice, France, 1990.

[9] M. Carson, et al. Secure window systems for UNIX. In *Proceedings of the USENIX Winter 1989 Conference*, January 1989.

[10] T. Dierks and C. Allen. RFC 2246: The TLS protocol: Version 1.0. `http://www.faqs.org/rfcs/rfc2246.html`, January 1999.

[11] J. Epstein. A prototype for Trusted X labeling policies. In *Proceedings of the Sixth Annual Computer Security Applications Conference*, Tucson, AZ, USA, December 1990. A discussion of visible labeling issues, not specific to X, but applicable to any windowing environment.

[12] J. Epstein and J. Picciotto. Issues in building Trusted X Window Systems. *The X Resource*, 1(1), Fall 1991. A revision of the previous paper [13], aimed at an audience which is X literate, but security ignorant.

[13] J. Epstein and J. Picciotto. Trusting X: Issues in building Trusted X window systems -or- what's not trusted about X? In *Proceedings of the 14th Annual National Computer Security Conference*, Washington, DC, USA, October 1991. A survey of the issues involved in building trusted X systems, especially of the multi-level secure variety.

[14] J. Epstein and M. Shugerman. A Trusted X Window System server for Trusted Mach. In *Proceedings of the USENIX Mach Conference*, Burlington, VT, USA, October 1990. This paper describes the initial architecture of the Trusted X Window System prototype developed at TRW. This paper was superseded by the paper at the Seventh Annual Computer Security Applications Conference [15].

[15] J. Epstein, et al. A prototype B3 Trusted X Window System. In *Proceedings of the Seventh Annual Computer Security Applications Conference*, San Antonio, TX, USA, December 1991. The architecture for TRW's high assurance multi-level secure X prototype.

[16] J. Epstein, et al. Evolution of a Trusted B3 Window System prototype. In *Proceedings of the 1992 IEEE Symposium on Research in Security and Privacy*, Oakland, CA, USA, May 1992. The history of the design and tradeoffs taken in TRW's prototype.

[17] G. Faden. Reconciling CMW requirements with those of X11 applications. In *Proceedings of the 14th Annual National Computer Security Conference*, October 1991.

[18] A. O. Freier, P. Karlton, and P. C. Kocher. The SSL protocol: Version 3.0. `http://wp.netscape.com/eng/ssl3/draft302.txt`, November 1996.

[19] D. Giampaolo. xkey source code. `http://www.google.com/`, May 2005.

[20] R. D. Graubart, J. L. Berger, and J. P. L. Woodward. Compartmented mode, workstation evaluation criteria, version 1. Technical Report MTR 10953 (also published by the Defense Intelligence Agency as document DDS-2600-6243-91), The MITRE Corporation, June 1991.

[21] P. Jones. RFC 3174: US secure hash algorithm 1 (SHA1). `http://www.faqs.org/rfcs/rfc3174.html`, September 2001.

[22] J. M. McCune, A. Perrig, and M. K. Reiter. Seeing-is-believing: Using camera phones for human-verifiable authentication. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2005.

[23] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press Series on Discrete Mathematics and its Applications. CRC Press, 1997.

[24] Microsoft Corporation. Next generation secure computing base. `http://www.microsoft.com/resources/ngscb/`, May 2005.

[25] Sun Microsystems. Mobile information device profile (MIDP) version 2.0. `http://java.sun.com/products/midp/`, May 2005.

[26] OpenSSL Project team. OpenSSL. `http://www.openssl.org/`, May 2005.

[27] J. Picciotto. Trusted X Window System. Technical Report MTP 288, The MITRE Corporation, February 1990.

[28] J. Picciotto. Towards trusted cut and paste in the X Window System. In *Proceedings of the Seventh Annual Computer Security Applications Conference*, December 1991.

[29] J. Picciotto and J. Epstein. A comparison of Trusted X security policies, architectures, and interoperability. In *Proceedings of the Eighth Annual Computer Security Applications Conference*, December 1992.

[30] D. S. H. Rosenthal. LInX—a Less INsecure X server (Sun Microsystems unpublished draft).

[31] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. Design and implementation of a TCG-based integrity measurement architecture. In *Proceedings of the USENIX Security Symposium*, 2004.

[32] Jeremy Scott-Joynt. The enemy within. BBC News, Available at: `http://news.bbc.co.uk/2/hi/business/4510561.stm`, May 2005.

[33] Arvind Seshadri, Mark Luk, Elaine Shi, Adrian Perrig, Leendert van Doorn, and Pradeep Khosla. Pioneer: Verifying integrity and guaranteeing execution of code on legacy platforms. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, October 2005.

[34] J. S. Shapiro, J. Vanderburgh, E. Northup, and D. Chizmadia. Design of the EROS trusted window system. In *Proceedings of the USENIX Security Symposium*, 2004.

[35] D. X. Song, D. Wagner, and X. Tian. Timing analysis of keystrokes and timing attacks on ssh. In *Proceedings of the USENIX Security Symposium*, 2001.

[36] F. Stajano and R. Anderson. The resurrecting duckling: Security issues for ad-hoc wireless networks. In *Proceedings of the Security Protocols Workshop*, 1999.

[37] J. T. Trostle. Timing attacks against trusted path. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 1998.

[38] Trusted Computing Group. Security in mobile phones whitepaper. `https://www.trustedcomputinggroup.org/downloads/whitepapers/TCG-SP-mobi%le-sec_final_10-14-03_V2.pdf`, October 2003.

[39] Trusted Computing Group. Trusted platform module main specification, Part 1: Design principles, Part 2: TPM structures, Part 3: Commands. `http://www.trustedcomputinggroup.org`, October 2003. Version 1.2, Revision 62.

[40] J. P. L. Woodward. Security requirements for system high and compartmented mode workstations. Technical Report MTR 9992, Revision 1 (also published by the Defense Intelligence Agency as document DDS-2600-5502-87), The MITRE Corporation, November 1987.

[41] The XFree86 project, Inc. `http://www.xfree86.org/`, May 2005.

[42] The X.Org foundation. `http://www.x.org/`, May 2005.

[43] E. Ye and S. W. Smith. Trusted paths for browsers. In *Proceedings of the USENIX Security Symposium*, August 2002.

[44] E. Ye, S. W. Smith, and D. Anthony. Trusted paths and web browsers. *ACM Transactions on Information System Security*, 2004.