Safe Passage for Passwords and Other Sensitive Data*

Jonathan M. McCune Adrian Perrig
CyLab / Carnegie Mellon University
{jonmccune,perrig}@cmu.edu

Michael K. Reiter
University of North Carolina at Chapel Hill
reiter@unc.edu

Abstract

The prevalence of malware such as keyloggers and screen scrapers has made the prospect of providing sensitive information via web pages disconcerting for security-conscious users. We present Bumpy, a system to exclude the legacy operating system and applications from the trusted computing base for sensitive input, without requiring a hypervisor or VMM. Bumpy allows the user to specify strings of input as sensitive when she enters them, and ensures that these inputs reach the desired endpoint in a protected state. The inputs are processed in an isolated code module on the user's system, where they can be encrypted or otherwise processed for a remote webserver. We present a prototype implementation of Bumpy.

1 Introduction

Today, a security-conscious user who wants to verify that her input is not observed by malicious code during a sensitive online financial transaction faces an impasse. *Keyloggers* can capture a user's typed input and *screen scrapers* can process the content displayed to the user to obtain sensitive information such as credit card numbers.

These malware exploit the vulnerabilities that are endemic to the huge computing base that is trusted to secure our private information. Today's popular operating systems employ monolithic kernels, meaning that a vulnerability in any part of the OS renders users' sensitive data insecure regardless of what application they may be running. On top of this untrustworthy OS sits a complex and monolithic web browser, which faces protection and assurance challenges similar to those of the OS. It is not surprising that trusting this software stack for the protection of private data in web transactions often leads to data compromise.

We present Bumpy, a system for protecting a user's sensitive input intended for a webserver from a compromised client OS or compromised web browser. We consider a user who desires to provide strings of information (e.g., a credit card number or mailing address) to a remote webserver (e.g., her bank) by entering it via her web browser. We focus on user input to web pages, although our techniques can also be applied to local applications. Bumpy is able to protect this sensitive user input by reducing the requisite trusted computing base to exclude the legacy OS and applications without requiring a hypervisor or VMM.

Bumpy employs two primary mechanisms. First, the initial handling of all keystrokes is performed in a special-purpose code module that is isolated from the legacy OS using the Flicker [18] system. Second, we establish the convention that sensitive input begin with the *secure attention sequence* `@@`, so that a user can indicate to this module that the data she is about to type is sensitive. These sensitive inputs are released to the legacy platform only after being encrypted for the end webserver or otherwise processed to protect user privacy [10, 11, 25].

Bumpy allows the remote webserver to configure the nature of the processing performed on user input before it is transmitted to the webserver, and automatically isolates the configurations and data-handling for mutually distrusting webservers. The webserver for which the user's current input will be processed can receive a TCG-style attestation that the desired input protections are in-place, potentially allowing the webserver to offer additional services to users with improved input security.

In order for the user to determine the website for which her input will be encrypted, she requires some trusted display to which the input-handling module can send this information. Since the client computer display cannot be trusted in our threat model, we explore the use of a separate user device, or Trusted Monitor, that receives such indicators from the input-handling module, authenticates them (using digital signatures) and displays them to the user.

Our prototype implementation of Bumpy demonstrates both the practicality of our approach and the fact that com-

*This research was supported in part by CyLab at Carnegie Mellon under grant DAAD19-02-1-0389 from the Army Research Office, and grants CNS-0509004 and CT-0756998 from the National Science Foundation, by the iCAST project, National Science Council, Taiwan under the Grants No. (NSC95-main) and No. (NSC95-org), and by gifts from AMD and Intel. The views and conclusions contained here are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either express or implied, of AMD, ARO, CMU, Intel, NSF, or the U.S. Government or any of its agencies.

modity hardware already offers nearly the full set of functionality needed to achieve these protections. In fact, the only compromise we make in our implementation is using an embedded Linux system as an encrypting USB Interposer, as we have been unable to locate keyboards or mice offering programmable encryption. We also leverage a smartphone as a Trusted Monitor for the user. However, we emphasize that the emergence of encrypting keyboards and far simpler devices to serve as a Trusted Monitor would suffice to remove any bloat from Bumpy's TCB. Bumpy is achievable without any client-side trusted software of complexity even close to that of a general-purpose OS, VMM, or hypervisor.

2 Related Work

We discuss prior work on trusted devices for sensitive user actions, split application architectures, trusted window managers, password processing, and TCB minimization.

The most closely related work is our prior work called Bump[1] in the Ether (BitE) [21]. BitE circumvents the legacy input path by leveraging encryption by user input devices (e.g., an encrypting keyboard), just as Bumpy does. However, BitE retains the legacy OS and Window Manager in its TCB, is tailored to local applications, and performs attestations to its correct functioning based on a static root of trust. In contrast, Bumpy dramatically reduces the TCB for input by leveraging a dynamic root of trust for each input event, works for sensitive input to websites, and supports secure post-processing of sensitive input (e.g., password hashing).

Borders and Prakash propose a Trusted Input Proxy (TIP) as a module in a virtual machine architecture where users can indicate data as sensitive using a keyboard escape sequence [5]. Users are presented with a special dialog box where they can enter their sensitive data, after which it is injected into the SSL session by the TIP. Again, however, the TCB of TIP includes a VMM and OS, whereas Bumpy's TCB includes neither.

The Zone Trusted Information Channel (ZTIC [12]) is a recently-announced device with a dedicated display and the ability to perform cryptographic operations. Its purpose is to confirm online banking transactions in isolation from malware on the user's computer. This device is appropriate for use as a Trusted Monitor in Bumpy.

Bumpy separates the process of accepting user input into trusted and untrusted parts, and thus can be viewed as implementing a type of privilege separation [28]. Several variations of this theme have been explored in the literature. Balfanz and Felten [2] describe the need for "splitting trust" and argue that hand-held computers can make effective smart cards since they have a distinct user interface that is not subject to malware on the user's host computer. Sharp et al. explore an architecture where applications run on a trusted platform and export both a trusted and an untrusted display [31]. They also consider split web applications where all sensitive operations are confirmed on a mobile device [30], and where the mobile device serves as the trusted portion of a physically separate, but logically composed browsing experience [29]. Bumpy optionally uses the separate Trusted Monitor as a verifier and indicator for the input framework, rather than as a platform for execution of portions of a split application or as an input device. But perhaps more importantly, the TCB of Bumpy is far smaller than in these other works, and in fact Bumpy can be viewed as extreme in this respect.

Trusted window managers have also been proposed as a solution to sensitive input and screen content. A compelling recent example is Nitpicker [9], but it currently requires changing operating systems and porting existing legacy applications. Bumpy remains compatible with existing legacy operating systems, to the extent that they meet the requirements for Flicker [18] (i.e., it may be necessary to install a kernel module or driver).

Ross et al. developed PwdHash, an extension for the Firefox web browser that hashes users' typed passwords in combination with the domain serving the page to produce a unique password for every domain [25]. The PwdHash algorithm adapts earlier work by Gabber et al. on protecting users' privacy while browsing the web [10, 11]. Chiasson et al. identify usability problems with PwdHash, specifically, that it provides insufficient feedback to the user regarding the status of protections [7]. We extend this work in two ways. First, we implement the PwdHash algorithm as one possible transformation of sensitive data in Bumpy, with a much smaller TCB than the web browser and OS that must be trusted with PwdHash. Second, we leverage a Trusted Monitor to provide feedback to the user regarding the status of her input. Validating the efficacy of our feedback mechanisms with a user study remains the subject of future work; this paper presents the design and implementation.

Bumpy builds on Flicker, an architecture that leverages the Trusted Computing concept of *Dynamic Root of Trust* to enable execution of a special-purpose code module (called a *Piece of Application Logic*, or PAL) while including only a few hundred lines of additional code in its TCB [18]. Remote attestation technology based on the Trusted Platform Module (TPM [34]) can be used to convince a remote party that precisely this code module and nothing else executed during a Flicker *session*. Flicker supports protocols for establishing authentic communication between a PAL and a remote entity, and it is architected such that the code that generates attestations need not be trusted. Additional background information on the underlying Trusted Computing technologies can be found in Appendix A.

[1] We derive the name Bumpy from Bump in the Ether.

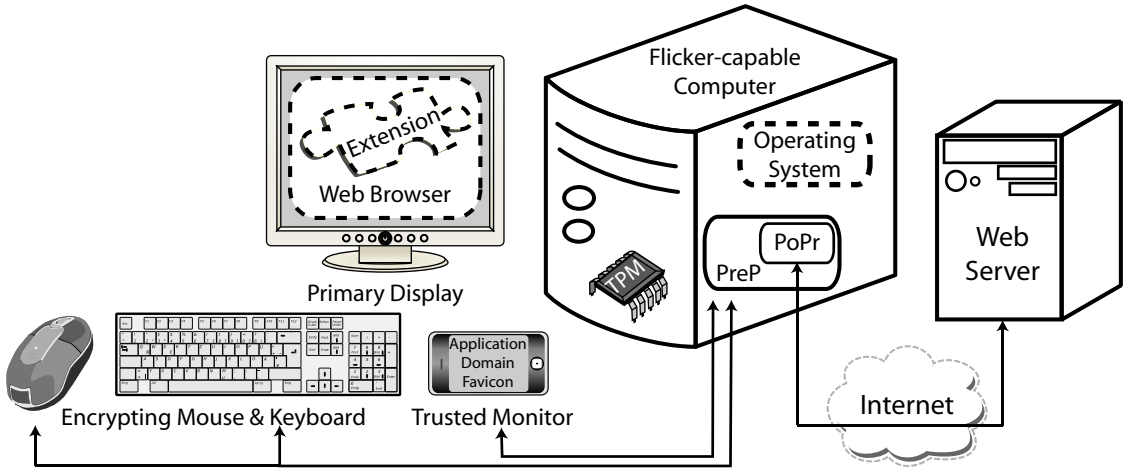


Figure 1. Logical flow through the major components of the Bumpy system. The OS, web browser, and browser extension are untrusted.

3 Overview

We detail our goals and assumptions, introduce the user experience, and provide an overview of our design and the major system components of Bumpy (Figure 1).

3.1 Goals and Assumptions

Goals. Our goals are to protect keystrokes from a potentially malicious legacy input system while retaining a seamless user experience, and to offer assurance to both the remote webserver and the user herself that input is protected. To the remote webserver, we provide an attestation that the user's input was protected with Bumpy, including the presence of encryption-capable input devices. To the user, we provide an indicator of whether it is safe to enter sensitive input. Bumpy achieves this without breaking compatibility with existing operating systems and without requiring a hypervisor or VMM.

Assumptions and Threat Model. We consider the user's OS and applications (including the web browser and its extensions) to be malicious. We assume the user has a trustworthy mobile device to serve as a Trusted Monitor and input devices (keyboard and mouse) capable of encryption. We also assume the remote webserver to which the user wishes to direct her input is uncompromised, and that the certificate authority (CA) that issues the webserver's SSL certificate is similarly uncompromised.

We leverage the Flicker system to protect sensitive code executing on the user's computer [18]. As such, the user's computer must meet the hardware requirements for Flicker: a version 1.2 TPM, and a CPU and chipset capable of establishing a *Dynamic Root of Trust*, also known as *late launch*. Appendix A provides additional background on the relevant technologies, which are widely available today.

3.2 User Experience

We are striving to make Bumpy usable by non-experts to protect sensitive input. Our mechanism employs a convention for entering sensitive information, and a trustworthy indication of the destination for that information. This indication is conveyed via an external display, called the Trusted Monitor (Figure 1). It is our intention that the Trusted Monitor will help to alleviate some of the usability problems (e.g., a lack of feedback) identified for password managers such as PwdHash [7], although we leave a formal usability study as future work.

In the common case, the user experience with Bumpy follows this sequence:

1. The user signals that she is about to enter sensitive information by pressing `@@`. Note that this can be thought of as a convention, e.g., "my passwords should always start with `@@`."

2. The Trusted Monitor beeps to acknowledge the reception of `@@` in the PreP, and updates its display to show the destination of the user's upcoming sensitive input.

3. The user types her sensitive data. Bumpy does not change this step from the user's perspective.

4. The user performs an action that signals the end of sensitive input (e.g., presses Tab or Enter, or clicks the mouse). Bumpy does not change this step from the user's perspective.

While users are accustomed to typing their passwords without seeing the actual characters on-screen (e.g., the characters appear as asterisks), most other sensitive data is displayed following entry. Given our desire to remove the legacy OS from the input TCB and the threat of malicious screen scrapers, this echoing to the main display must be prevented by Bumpy. The usability of entering relatively short sequences of characters (e.g., credit card numbers) under these conditions may remain acceptable to concerned users, but it is not ideal. We perceive this as the price one must pay for secure input with an untrusted OS.

For those users employing a Trusted Monitor of sufficient capability, sensitive keystrokes can be echoed there for validation by the user. While this partially eliminates the challenge of entering input "blind," a minimal Trusted Monitor would still make it impractical to compose lengthy messages.

3.3 Technical Overview

We now summarize the main components of Bumpy. In Figure 1, solid arrows represent logical communication through encrypted tunnels. Bumpy is built around encryption-capable input devices sending input events directly into a Pre-Processor (PreP) protected by the Flicker system on the user's computer. Bumpy allows the remote webserver to control (within certain limits) how users' sensitive input is processed after it is entered with Bumpy. We term this Post-Processing, and enable it by allowing the webserver to provide a post-processor (PoPr) along with web content. Bumpy tracks and isolates PoPrs from different webservers, as well as supports standardized PoPrs that may be used across many websites. Leveraging the Flicker system [18], the PreP and PoPrs execute in isolation from each other and from the legacy OS.

Encryption and password-hashing are two desirable forms of post-processing of user input. Site-specific hashing of passwords (as in PwdHash [25]) can provide password diversity across multiple websites, and prevent the webserver from ever having to handle the user's true password. Dedicated post-processing with server-supplied code can resolve issues with the PwdHash [25] algorithm producing unacceptable passwords (e.g., passwords without any punctuation characters that violate the site's password requirements) or passwords from a reduced namespace, since the webserver itself provides the algorithm. Encrypting input directly within the Bumpy environment to the remote webserver dramatically reduces the client-side TCB for sensitive user input.

4 Identifying and Isolating Sensitive Input

In this section, we focus on acquiring input from the user in the PreP, and storing sensitive input such that it is protected from the legacy OS. Section 5 treats the post-processing and delivery of this input to approved remote servers. We identify three requirements for protecting user input against a potentially malicious legacy OS:

- R1 All input must be captured and isolated.
- R2 Sensitive input must be distinguishable from non-sensitive input.
- R3 The final destination for sensitive input must be identifiable.

Requirement R1 for protecting user input is to acquire the input without exposing it to the legacy OS. The challenge here is that we wish to avoid dependence on a VMM or hypervisor and retain the OS in charge of device I/O. We propose to use encryption-capable input devices to send opaque input events through the untrusted OS to a special-purpose Piece of Application Logic (PAL) that is protected by the Flicker [18] system (Steps 1–4 in Figure 2). This PAL is architected in two components. The first is specifically designed to Pre-Process encrypted input events from the input devices, and we call it the PreP. The PreP achieves requirement R2 by monitoring the user's input stream for the secure attention sequence "`@@`" introduced in Section 3.2, and then taking appropriate action (which affects what input event is released in Step 5 of Figure 2). The PreP serves as the source of input events for post-processing by a destination-specific Post-Processor (PoPr). The process of authenticating a PoPr serves to identify the final destination for sensitive input (requirement R3). The PoPr encrypts or otherwise processes the received input for the remote server (Steps 6–8 in Figure 2).

These components are separated so that the PreP's sensitive state information can be kept isolated from the PoPr, as Bumpy supports multiple, mutually distrusting PoPrs that accept input events from the same PreP. The PreP's state information includes the cryptographic state associated with the encrypting input devices, the currently active PoPr, and a queue of buffered input events. The PreP's state is protected by encrypting it under a master key that is maintained on the user's TPM chip. The properties of Flicker [18] guarantee that no code other than the exact PreP can access it. For the following sections we encourage readers not intimately familiar with trusted computing technology to read Appendix A before proceeding.

We defer discussion of the one-time setup of the cryptographic state associated with the encrypting input device(s) until Section 4.2. We proceed assuming that the setup has already been completed.

4.1 Steady-State User Input Protection

We describe the actions taken by the PreP in response to user input events and events from the web browser. The state machine in Figure 3 summarizes these actions.

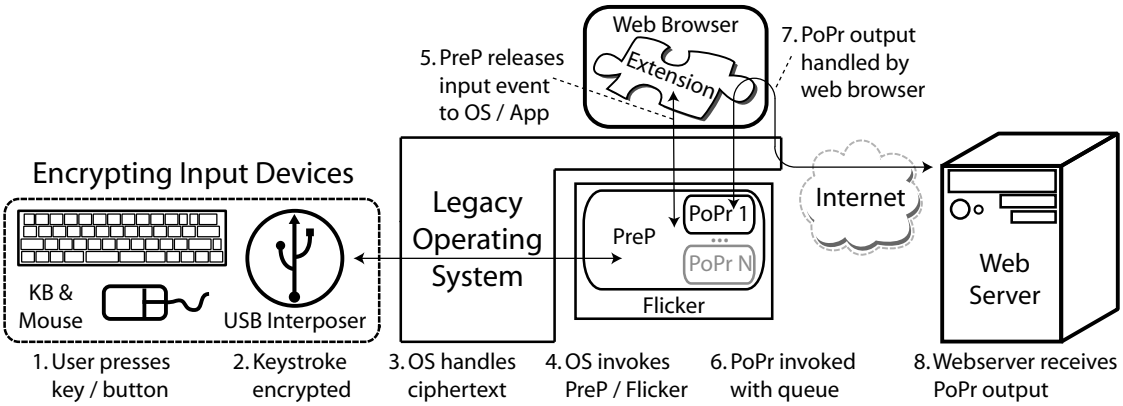


Figure 2. Acquiring user input with Bumpy. Steps 1–5 (described in Section 4) occur for every keystroke or mouse click performed by the user. Steps 6–8 (described in Section 5) occur only in response to a keystroke or mouse click that the PreP detects will cause a *blur* event (in the web browser GUI sense) while the user is entering sensitive data. We revisit this figure in Section 8.3 while describing the life of a keystroke within our implementation.

Every event e is processed in a distinct Flicker session, i.e., the PreP only accepts a single event as an input parameter. We design Bumpy this way out of necessity, due to two conflicting desires. The first is to avoid trusting the OS, and the second is to remain responsive to the user as she provides input to her system. One consequence of this design is that every Flicker session (i.e., PreP invocation) begins and ends with the decryption and encryption of the PreP's sensitive state information, respectively.

The legacy OS provides arguments for each invocation of the PreP: the event e to be processed, the SSL certificate for the active website, the PoPr associated with the active website, and the PreP's encrypted state. Each event e can be an encrypted keystroke or mouse click, or it can be a *focus* event[2] from the browser. All other event types from the browser are ignored. The PreP maintains in its state the necessary cryptographic information to decrypt and integrity-check input events from the input device(s). The master keys used to protect the secrecy and integrity of the PreP's state are TPM-protected based on the identity of the PreP. We describe these protocols in greater detail as part of our implementation in Section 8.

During each run of the PreP (i.e., during each Flicker session in Step 4 of Figure 2), the state machine (Figure 3) begins in PreP Initialization and transitions to the state where the previous PreP invocation ended (maintained as State.Prev in Figure 3), where the current event then causes a single transition. Actions listed in a state are performed when an event causes arrival into that state (as opposed to returning to a state because of the value of State.Prev). If there is no action for a particular event in a particular state, then that event is ignored. For example, browser *focus* events are ignored in the Second @, Enqueue Input, and Invoke PoPr states.

PreP Initialization. Regardless of the previous state of the PreP, it always performs an initialization step. The PreP first decrypts and integrity-checks its own long-term state, verifies that the provided SSL certificate is valid using its own list of trusted certificate authorities (which we define as being part of the PreP itself), and verifies that the provided PoPr is signed by the provided SSL certificate. (If any of these verification steps fail, the current event is dropped.) Next, the incoming event e is processed. If it is an encrypted input event from the input device(s), then it is decrypted, integrity-checked, and verified to be in-sequence (using cryptographic keys and a sequence number maintained in the PreP's state). If any of the steps involving synchronization with the input device(s) fail, then input events can no longer be received. We discuss options for recovery in Section 8.2.3.

The PreP then transitions to State.Prev where e will cause one additional state transition. During the very first invocation of a PreP, it transitions to Pass Input Unmodified. The following paragraphs describe the actions taken upon entry to a state caused by an event, not by State.Prev. At the end of each of these states, the PreP's sensitive long-term state is *sealed*[3] using the TPM-protected master key,

[2] A *focus* event is an event in the web browser's graphical user interface where a new component such as an HTML text input field becomes active. This generally follows a *blur* event caused by the previously focused component becoming inactive. These events fire in response to user actions, such as clicking the mouse.

[3] *Sealed* means that the state is encrypted and integrity-protected (by computing a MAC) for subsequent decryption and integrity-verification. This use of *sealed* is consistent with the TPM's *sealed storage* facility,

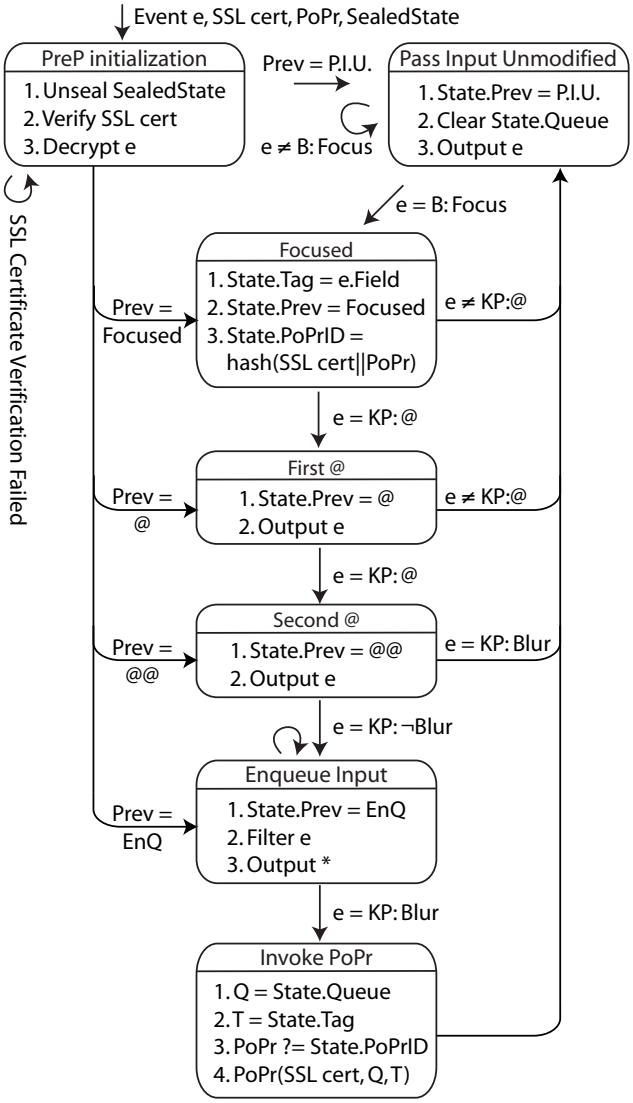


Figure 3. States of the PreP. KP = keypress or mouse click. B:Focus = browser GUI *focus* event. Blur indicates action taken in response to events on the encrypted input channel that cause a GUI *blur* event (e.g., Tab, Shift+Tab, Enter, or mouse click).

and then cleared (set to zero) before the PreP terminates and produces output. Untrusted code running on the OS maintains the ciphertext that makes up the PreP's sealed state and provides it as input to the PreP's next invocation.

Pass Input Unmodified. The common case for user input is that it is non-sensitive, and does not require special protection by the PreP. Any existing queue of sensitive events is discarded upon entry to this state. Unless e is a browser *focus* event, State.Prev is set to remain in the Pass Input Unmodified state. The current input event is not considered sensitive, and it is provided as an output when the PreP exits. The legacy OS then interprets this input event just as it does today.

Focused. A browser *focus* event contains the name of the field that has just received focus (e.Field). The PreP saves the cryptographic hash of the current PoPr and its SSL certificate (which was validated during PreP Initialization) as the *PoPrID* . It is necessary to track the PoPrID to ensure that the PoPr is not maliciously exchanged while the user is typing sensitive input. When invoked with a keystroke, a PreP in the Focused state checks whether the keystroke is the `@` character. If so, the PreP transitions to the First `@` state. Otherwise, the PreP transitions back to the Pass Input Unmodified state. The keystroke is output to the legacy OS for processing. Note that the `@` keystroke is not secret; it serves only to signify that the user may be about to enter something she considers sensitive.

First `@`. We have defined the secure attention sequence for Bumpy to be `@@` in the input stream immediately following a browser *focus* event. This state serves to keep track of the `@` characters that the user enters. It is possible that the user is in the process of initiating sensitive input. When invoked with a keystroke that is the `@` character, the system transitions to the Second `@` state. Otherwise, the system transitions back to the Pass Input Unmodified state. The keystroke is output to the legacy OS.

Second `@`. When in the Second `@` state, the user has successfully indicated that she is about to provide some sensitive input using Bumpy. From this point forward, the only way for the user to terminate the process of entering sensitive input is to perform an action that will cause a *blur* event in the current input field. A *blur* event is an event in the graphical user interface that indicates that a particular component is becoming inactive, generally because the focus is now elsewhere. Relevant actions include clicking the mouse or pressing Tab, Shift+Tab, Alt+Tab, or Enter. Note that we explicitly *do not* listen for *blur* events from the web browser, as a malicious browser would be able to terminate secure input prematurely. If the user's input does not represent a *blur* event, then the system transitions to the Enqueue Input state.

Enqueue Input. For each PreP invocation in the Enqueue Input state, the decrypted keystroke is filtered before being appended to State.Queue. The filter identifies and drops illegal password characters that would not cause a *blur* event (e.g., meta-characters used for editing, such as arrows, Backspace, and Delete). We discuss editable sensitive

which we describe in Appendix A.

input in Section 10. This process continues until the user causes a *blur* event, e.g., presses Tab, Shift+Tab, Alt+Tab, Enter, or clicks the mouse. A decoy input event is released to the legacy OS: an asterisk. From the perspective of the legacy OS, when the user types her sensitive input, she appears to be typing asterisks. This has the convenient property of mimicking the usual functionality of input to password fields even if the current field is a normal text field: all keystrokes appear as asterisks. Note that these asterisks will eventually be discarded, as the PoPr provides the remote webserver with the true input for the protected fields. When the user causes a *blur* event, the system transitions to the Invoke PoPr state.

Invoke PoPr. When the PreP state machine transitions to the Invoke PoPr state, it means that the user successfully directed the web browser's focus to a particular field, entered `@@` to signal the start of sensitive input, provided some sensitive input, and then caused a *blur* event that signals the end of sensitive input. This input will now be handed to the PoPr for destination-specific processing (Section 5). First, however, it is necessary to check that the PoPr provided as an input to the PreP during the current Flicker session is the same PoPr that was provided during the *focus* event that initiated this sensitive input. If the PoPr is changed, malicious code may be trying to fool the user. In this case, the system transitions directly back to Pass Input Unmodified where the queue of sensitive input events is discarded. If the PoPr is confirmed to be the same, it is invoked. The PoPr also runs with Flicker's protections, but it is less trusted than the PreP. Thus, PreP state is sealed and cleared *before* invoking the PoPr with the queue of input events. This is essential, as the PreP state includes the cryptographic secrets involved in communication with the user's physical input device(s). If a PoPr could compromise this information, it might be able to collude with a malicious OS and capture all subsequent input on the user's platform. Once the PoPr has executed, its output is handed to the web browser via the legacy OS for submission to the webserver from which the PoPr originated. If the PoPr considers these input events to be secret, then they should be encrypted such that the legacy OS will be unable to learn their values.

4.2 Associating the PreP and Input Device(s)

Bumpy depends on input devices capable of encrypting input events generated by the user, so that the legacy OS can pass the opaque events to the PreP without learning their value. We now describe the process of establishing the necessary cryptographic keys when input device(s) and a PreP are first associated.

The PreP is invoked with the command to establish a new association with an input device. During this process, the PreP:

1. Generates symmetric encryption and MAC keys for the protection of its own long-term state, K_{lt_enc}, K_{lt_mac}, if they do not already exist.

2. Generates an asymmetric keypair, $K_{input}, K_{input}^{-1}$, for bootstrapping communication with the encrypting input device, and adds the private key to its long-term state. Note that no other software (not even a PoPr) will ever be allowed to access K_{input}^{-1}.

The public key K_{input} is then conveyed to the input device. Given the complexity of equipping input devices with root CA keys to verify certificates, we use a trust-on-first-use model where the input device simply accepts K_{input} and then prevents it from changing unexpectedly. Since encryption-capable input devices like we require do not currently exist, we specify how the input device enters a state where it is willing to accept a public encryption key. The greatest challenge is to prevent key re-establishment from being initiated in the presence of malicious code, whether through a design or implementation failure or a social engineering attack. One promising design may be a physical switch that must be placed into the "Establish Keys" position before key establishment can commence. This way, most users will establish keys once and forget about it. Another alternative is to use a location-limited channel [3, 20, 33]. We describe input device key establishment for our USB Interposer implementation in Section 8.

In the future, more sophisticated input devices may verify an attestation that the public key came from a known-good PreP. Another promising alternative is that input device manufacturers imprint their devices with a certificate establishing them as approved encrypting input devices, though it can be challenging to establish that a certificate corresponds to a particular physical device [22]. The PreP can then verify the origin of input events as trustworthy, provided that the PreP's list of trusted CAs covers manufacturers' signing keys.

Irrespective of the method of public-key exchange, symmetric keys for encryption and integrity protection should be established to maximize performance. Additionally, we require the use of sequence numbers so that the PreP can detect if any keystrokes are dropped or reordered by the legacy OS.

4.3 PreP State Freshness

The PreP must have the ability to protect its own state when the legacy OS has control (i.e., across Flicker sessions). The secrecy and integrity of the PreP's state is ensured by encrypting it under a symmetric master key kept in PCR-protected non-volatile RAM on the TPM chip. It is PCR-protected under the measurement of the PreP itself, so that no other code can ever unseal the master key. However, protecting the freshness of the PCR-protected state is more

challenging. The risk is a state roll-back or replay attack where, e.g., an attacker may try to keep the PreP in the Pass Input Unmodified (Figure 3) state by perpetually providing the same ciphertext of the PreP's state as an input to the PreP. The standard solution to this problem is to employ a secure counter or other versioning scheme that can track the latest version of the PreP's state. While the TPM does implement a monotonic counter [34], its specification dictates that the counter need only support incrementing every five seconds. This is clearly insufficient to keep up with per-keystroke events. Our solution is to leverage the sequence numbers associated with input events coming from the input devices (Section 8.2.3).

5 Input Post-Processing and Attestation

We now detail the actions taken by the PoPr to Post-Process sensitive input events enqueued by the PreP (Step 6 in Figure 2). Then, we describe the attestation process employed to convince the remote webserver (which is the PoPr provider) that it is receiving inputs protected by Bumpy.

5.1 Post-Processing Sensitive Input

The final destination for sensitive input protected by Bumpy is the webserver from which the current web page originated. In Section 4.1, we describe how the PoPr is invoked by the PreP when the user has completed entering sensitive input for a particular field. The PoPr is provided by the webserver hosting the current web page in the user's browser. The sensitive input is tagged with the name of the field that had focus when the input was entered, and this $\{inputString,\ tag\}$ pair is what the PoPr receives. The PoPr can perform arbitrary, site-specific transformations on the sensitive input before handing the (potentially opaque) result to the web browser for transmission to the remote webserver when the page is submitted.

5.1.1 Example Forms of Post-Processing

We consider two example forms of post-processing that we believe to be widely useful on the web today, though there may be many others. The first is encryption of user input such that only the webserver can process the raw input, and the second is destination-specific hashing (with the PwdHash algorithm [25]) so that, e.g., passwords cannot be reused at multiple websites.

End-to-End Encryption. Encrypting sensitive inputs for the webserver completely removes the *web browser and OS* from the TCB of the input path for the field accepting the sensitive input, though it requires the *webserver* to be Bumpy-aware. This capability is achieved on the user's system by embedding a webserver-generated public encryption key in the PoPr. The PreP will automatically check that the PoPr (and hence its encryption key) is certified by the webserver from which it originated.

Note that it may seem tempting to have the input device encrypt the user's input all the way to the remote webserver, removing the need for the PreP or PoPr. We prefer the flexibility afforded by the Flicker architecture to process input in a PoPr on the user's system in whatever way is appropriate for a given application or remote system. It is not the duty of the input device manufacturer to foresee all of these possible applications. Indeed, incorporating too much programmability into the input device itself is sure to make it a promising target for attack. Rather, the input device is tasked solely with getting keystrokes securely to the PreP, and websites concerned about the size of the input TCB can supply their own minimized PoPr.

Destination-Specific Hashing with PwdHash. A second form of post-processing – contained entirely within the user's system – is to perform a site-specific transformation of certain fields before they are released to the web browser for transmission to the webserver. For example, usernames or passwords can be hashed along with the webserver's domain name, thereby providing the user with additional protection when she employs the same username or password at multiple websites. The domain name is obtained from the webserver's SSL certificate, which is verified by the PreP before the PoPr begins executing. In Section 8, we describe our implementation of the PwdHash [10, 11, 25] algorithm within Bumpy.

5.1.2 Activating a PoPr

It is possible that malicious browser or OS code will intentionally load the wrong PoPr for the current site. If the PoPr is well-behaved (i.e., provided by a reputable webserver), then it is unlikely to expose the user's sensitive input. However, attackers may intentionally use a PoPr from a compromised server with a valid SSL certificate. In this case, our defense is the Trusted Monitor, as it will display the domain name and favicon[4] of the website that has certified the current PoPr. It is the user's responsibility to see that the information on the Trusted Monitor corresponds to the web page that she is browsing. We explain the detailed operation of the Trusted Monitor, including the users' responsibilities, and how secure communication between the PreP and the Trusted Monitor is established, in Section 6.

[4] A *favicon* is an image associated with a particular website or web page, installed by the web designer. It is commonly displayed alongside the address bar and alongside a tab's title for browsers that support tabs.

5.2 Attestation and Verifying Input Protections

Flicker enables the computer using Bumpy to attest to the PreP and PoPr that have run most recently. This attestation can be verified by a remote entity to ascertain whether the user's input received the intended protection. Though there are no technical limitations governing which device (or devices) perform the verification, we proceed from the perspective of the remote webserver as the verifier. Institutions such as banks employ professional administrators who are better suited than the average consumer to make trust decisions in response to which PreP and PoPr are in operation on the user's computer. For certain types of transactions (e.g., online banking), the webserver may be willing to expose more services to a user whose computer can provide this assurance that the user's input is being protected. However, the user still must behave responsibly.

If verification by the remote webserver succeeds, then the requested web page can be served normally. However, if verification fails, then the software state of the user's system cannot be trusted, and the webserver should prevent access to sensitive services. One option is to serve a web page with an explanation of the error, though there is no guarantee that the malicious (or unknown) software will display the error. We discuss an extension to create a trusted path between the Trusted Monitor and webserver for conveyance of such error notifications in Section 10.

5.2.1 Establishing Platform Identity

TPM-based remote attestation is used to convince the webserver that the user's input is protected with Bumpy. However, the remote webserver must first have a notion of the identity of the user's computer system. We use an Attestation Identity Key (AIK) generated by the TPM in the user's computer. Appendix A discusses known techniques for certifying an AIK, any of which can be applied to Bumpy. Here, Bumpy benefits from the property of the Flicker [18] system that causes attestations to cover *only* the PreP and PoPr code that was executed, and no other software at all.

5.2.2 The Attestation Protocol

Here, we describe the protocol between a user's system with Bumpy and a Bumpy-aware webserver when they connect for the first time. As an example, we consider a user who is trying to login to a webserver's SSL-protected login page. The user's browser sends a normal HTTPS request for the login page.

In response, the webserver participates in an SSL connection and delivers the login page. Embedded within the page (e.g., in a hidden input element) are several Bumpy-specific pieces of information, which must be signed by the webserver's private SSL key:

- $nonce$ – A nonce to provide replay protection for the ensuing attestation.
- $hash$ – The cryptographic hash of the PoPr. The PoPr itself can be obtained with another HTTP request and verified to match this hash.
- $favicon$ – The favicon corresponding to the webserver.
- $Cert_{ws\_enc}$ – A public encryption key signed by the webserver's private SSL key.

A well-behaved browser then passes the newly received PoPr, embedded information, and the webserver's public SSL certificate to the untrusted code module that manages the invocation of Flicker sessions with the PreP. During subsequent Flicker sessions, these data are provided as input to the PreP. The PreP verifies the webserver's SSL certificate using its own list of trusted CAs, and verifies that the other input parameters are properly signed.

If all verifications succeed, an output message is prepared for the webserver. This message requires the generation of an asymmetric keypair within the PoPr that will serve to authenticate future encrypted strings of completed input as having originated within this PoPr. This key is generated and its private component is protected in accordance with the Flicker external communication protocol [19]. Only this PoPr will ever be able to access the private key.

When key generation completes, the newly generated public signing key (K_{PoPr_sig}) is extended into a TPM Platform Configuration Register (PCR) and output from the PoPr. Untrusted code running on the legacy OS then passes this key back to the web browser, along with the user's system's public identity (e.g., an Attestation Identity Key, or the set of Endorsement Key, Platform, and Conformance Credentials) and a TPM attestation covering the relevant PCRs. These tasks can be left to untrusted code because the properties of the PCRs in the TPM chip prevent untrusted code from undetectably tampering with their values.

In steady-state, the PoPr will encrypt user input using the public key in the webserver-provided $Cert_{ws\_enc}$, and sign it with $K^{-1}_{PoPr\_sig}$ to authenticate that it came from the PoPr running on the user's computer. There is no need to perform an attestation during future communication between this PoPr and webserver.

5.2.3 Processing Attestation Results

Remote entities need to have knowledge that a set of attested measurements represents a PreP and PoPr that keep the user's input and PreP state safe (encrypted when untrusted code runs, which may include Flicker sessions with other, distrusted PALs). Prominent institutions (e.g., banks) may develop and provide their own PoPrs for protecting user input to their websites. In these cases, the institution's

webserver can easily be configured with the expected PoPr measurements, since it provided the PoPr in the first place. If one PoPr proves to be sufficient for a wide variety of websites, then its measurement may become a standard which can be widely deployed.

The webserver must also have knowledge of existing PrePs in order to make a trust decision based on the attestation result. We expect the number of PrePs to be reasonably small in practice, as most input devices adhere to a well-known (and simple) protocol.

6 The Trusted Monitor

Bumpy's input protections by themselves are of limited value unless the user can ascertain whether the protections are active when she enters sensitive data. The primary usability criticism [7] of PwdHash [25] is that it provides insufficient feedback to the user as to the state of input protections. Thus, it is of utmost importance that the user is aware of the transition between protected and unprotected input. With Bumpy, the Trusted Monitor serves as a trusted output device that provides feedback to the user concerning the state of input protections on her computer.

6.1 Feedback for the User

When input protections are active, the Trusted Monitor displays the final destination (e.g., website) whose PoPr will receive her next sensitive input. We represent this using the domain name and favicon of the currently active PoPr, as reported by the PreP. When input protections are disabled, the Trusted Monitor displays a warning that input is unprotected and that users should use `@@` to initiate sensitive input. Figure 4 shows screenshots from our implementation. In addition to changing the information on its display, the Trusted Monitor uses distinctive beeps to signal when protections become enabled or disabled.

The Trusted Monitor works in concert with the properties of the PreP's Second `@` and Enqueue Input states (Figure 3): when in these states, the PoPr is *locked in* and cannot change until after the sensitive input to a single field is processed by this PoPr (in the Invoke PoPr state). As such, the PoPr represented by the domain name and favicon that are displayed by the Trusted Monitor will remain the active PoPr until input to the current field is complete. Thus, there is no need for the user to worry about a malicious PoPr change in the middle of a string of sensitive input. However, the user must be diligent between fields. She must ensure that the Trusted Monitor responds to each unique `@@` sequence that she types (i.e., that the Trusted Monitor beeps and shows that protection is enabled) before proceeding to input her sensitive data. This is because the untrusted OS may affect the delivery of encrypted keystrokes to the PreP and PreP messages to the Trusted Monitor.

The risk is that malicious code may try to confuse the user such that she misinterprets the Trusted Monitor's display for one input field as indicating that her input is secure for additional input fields. One such attack works as follows. Malcode allows keystrokes and Trusted Monitor updates to proceed normally until the user begins typing sensitive input for one input field on a web page. The Trusted Monitor beeps and updates its display to indicate that protections are active. At this point, the malcode begins to suppress Trusted Monitor updates, but the Trusted Monitor cannot immediately distinguish between suppressed updates and a distracted user who has turned away from her computer. A user who finishes typing this secret and then transitions to another input field and proceeds to enter another secret — even after entering `@@` and glancing at the Trusted Monitor, but without waiting for confirmation of the receipt of the new `@@` by the PreP— renders the second secret vulnerable to disclosure. To expose this secret, the malicious OS plays the user's encrypted inputs to the PreP after the user is finished typing the second secret, but provides a malicious PoPr to the PreP when transitioning to the Focused and Invoke PoPr states for the second input. That is, the user provided the second secret presuming it was protected in the same way as the first, but since she did not confirm that the second `@@` was received by the PreP before she typed the second secret, it is vulnerable to disclosure to a malicious PoPr.

To help users avoid such pitfalls, it may be desirable for the Trusted Monitor to emit an audible "tick" per sensitive keystroke received by the PreP, in addition to the preceding beep when the `@@` is received. This way, the absence of ticks might be another warning to the user.

6.2 Protocol Details

To facilitate the exchange of information regarding the active PoPr, a cryptographic association is needed between the PreP and the Trusted Monitor. To establish this association, the Trusted Monitor engages in a one-time initialization protocol with the PreP, whereby cryptographic keys are established for secure (authentic) communication between the PreP and the Trusted Monitor. The protocol is quite similar to that used between the PreP and input device(s) in Section 4.2.

The initialization process for PreP-to-Trusted Monitor communication is an infrequent event (i.e., only when the user gets a new Trusted Monitor or input device). Thus, a trust-on-first-use approach is reasonable, where the Trusted Monitor simply accepts the public key claimed for the PreP. Any of a range of more secure (but more manual or more infrastructure-dependent) approaches can be employed, including ones that allow the Trusted Monitor to validate an attestation from the TPM on the user's computer as to the correct operation of the PreP and to the value of its pub-

lic key (a capability offered by Flicker [18]). The PreP can save its private key in PCR-protected storage on the user's computer, and so will be available only to this PreP in the future (as in Section 4).

The Trusted Monitor need not be a very complex device. Its responsibilities are to receive notifications from the user's computer via wired or wireless communication, and to authenticate and display those notifications. While our implementation employs a smartphone for a Trusted Monitor (Section 8), this is far more capable than is necessary (and more capable than we would recommend).

With a smartphone serving as the Trusted Monitor, there is no reason why the user's Trusted Monitor cannot perform the full gamut of verification tasks we have described as being in the webserver's purview. In fact, technically savvy and privacy-conscious users may prefer this model of operation, and it becomes significantly easier to adopt if a small number of PrePs and PoPrs become standardized across many websites. These users can learn that their input is being handled by precisely the PreP and PoPr that they have configured for their system, and that opaque third-party code is never invoked with their input.

7 Security Analysis

We discuss Bumpy's TCB, the implications of a compromised web browser, phishing attacks, and usability.

7.1 Trusted Computing Base

One of the primary strengths of Bumpy is the reduction in the TCB to which input is exposed on the user's computer. Always in the TCB are the encrypting input device and the PreP that decrypts and processes the encrypted input events on the user's computer. The PoPr associated with each website is also in the TCB for the user's interaction with that website, but the PreP isolates each PoPr from both the PreP's sensitive state and the OS (thereby preventing a malicious PoPr from harming a well-behaved OS). The encrypting input device is a dedicated, special-purpose hardware device, and the PreP is a dedicated, special-purpose software module that executes with Flicker's isolation [18]. A compromise of either of these components is fatal for Bumpy, but their small size dramatically reduces their attack surface with respect to alternatives available today, and may make them amenable to formal verification. The PoPr may be specific to the destination website, and may be considered a local extension of the remote server. It does not make sense to send protected input to a remote server that the user is unwilling to trust. Additionally, the PoPr's functionality is well-defined, leading to small code size.

Also in the TCB is the Trusted Monitor that displays authenticated status updates from the PreP, i.e., the domain name and favicon for the active PoPr. The Trusted Monitor never handles the user's sensitive input, so compromising it alone is insufficient to obtain the user's input. However, if the Trusted Monitor indicates that all is well when in fact it is not, then a phishing attack may be possible (Section 7.3).

7.2 Compromised Browser

If the user's browser or OS is compromised, then malicious code can invoke the PreP with input of its choosing. Bumpy can still keep the user's sensitive input safe provided that she adheres to the convention of starting sensitive input with `@@` and pays attention to the security indicator on her Trusted Monitor.

The cryptographic tunnel between the input device and PreP prevents malicious code from directly reading any keystrokes, and prevents the malicious code from injecting spurious keystrokes. Thus, a compromised browser's options are restricted to providing spurious inputs to the PreP, including SSL certificates, PoPrs, and browser *focus* events. None of these are sufficient to violate the security properties of Bumpy, but they can put the user's diligence in referring to the Trusted Monitor to the test.

Malicious SSL Certificates. The PreP is equipped with a list of trusted certificate authorities (CAs). Any SSL certificate that cannot be verified is rejected, causing sensitive keystrokes to be dropped. Thus, an attacker's best option is to compromise an existing site's SSL certificate (thereby reducing the incentive to attack the user's computer), or to employ a phishing attack by registering a similar domain name to that which the user expects (e.g., hotmai1.com, instead of hotmail.com) and using an identical favicon.

Malicious PoPr. The PreP will not accept a PoPr unless it can be verified with the current SSL certificate, thereby reducing this attack to an attack on the SSL certificate (as described in the previous paragraph) or webserver.

Malicious Browser Focus Events. A malicious browser may generate spurious or modified *focus* events in an attempt to confuse the PreP with respect to which field is currently active. However, regardless of which field is active, the user controls whether the current input events are considered sensitive. When they are sensitive, input to a field is always encrypted and tagged with the field's name before being released to the PoPr. A malicious *focus* event may only cause ciphertext to be tagged with the wrong field name, thereby impacting availability. However, we already consider an adversary which controls the OS on the user's computer, and is thus already in total control of availability.

7.3 Phishing

If a user is fooled by a phishing attack (e.g., she confuses similar-looking domains), she may be using Bumpy's protections to enter her sensitive data directly into a phishing website. Defeating phishing attacks is not our focus here, though Bumpy should be compatible with a wide range of phishing defenses [14]. As a simple measure, Bumpy provides an indicator on the Trusted Monitor that includes the domain name and favicon of the current website. Though we have not solved some of the intrinsic problems with certificate authorities and SSL, the PreP can enforce policies such as: only PoPrs from white-listed webservers are eligible to receive a user's input; PoPrs from blacklisted webservers can never receive a user's input; and self-signed certificates are never acceptable. These policies are enforceable in the PreP, and require the user to have a Trusted Monitor only to provide feedback to improve usability.

With a PoPr implementing PwdHash, only the hashed password is returned to the web browser. If a user is fooled into entering her password into a phishing site with a different domain name, the phishing site captures only a hash of the user's password, and must successfully perform an offline dictionary attack before any useful information is obtained about the user's password at other sites. Additionally, in the case where a user ignores the indicator but has established the habit of starting her password with `@@`, hashing of the user's password can restrict the impact of the user's being phished on one website to that website alone. With a compromised OS, malware on the user's system can observe the hashed password when it is released to the web browser, but this password is only valid at a single website.

7.4 Usability

Confusion. If users do not understand the Bumpy system, or their mental model of the system is inaccurate, then they may be fooled by a malicious web page. For example, a prompt such as the following may trick the user into believing that there is no need to prefix her password with `@@` on the current web page:

`Input your password: @@ _____`

The user may also become confused if she makes a typographical error entering `@@`, and tries to use backspace to correct it. Bumpy will not offer protections in this case, until the user changes to another input field and then comes back to the current field (i.e., causes a *blur* event and then a new *focus* event). The Trusted Monitor does indicate that protections are disabled, but it may not be obvious to the user why this is the case. We discuss editable sensitive input in Section 10.1.

Only a formal user study can ascertain the level of risk associated with this kind of attack, which we plan to pursue in future work.

Extra Mouse Clicks. When a user clicks in an input field, a *focus* event is generated for the field and conveyed to the PreP. The user's next mouse click is interpreted by the PreP as a *blur* event for the current input field, disabling input protection. An attack may be possible if the user clicks the mouse in an input field after already typing part of her input into the field. This click could be interpreted as a *blur* event, and cause the rest of the user's keystrokes to be sent unencrypted. This may arise when, e.g., the user forgot her credit card number after entering the first few digits from memory, and needs to go lookup the remainder. The Trusted Monitor will beep and update its display to indicate that input protections are disabled when this *blur* event happens, but this may be a source of user confusion.

8 Implementation

Our implementation of Bumpy supports verification by the remote webserver with a smartphone as Trusted Monitor to provide feedback to the user. We implement two PoPrs: one encrypts sensitive input as-is for transmission to a Bumpy-aware webserver, and the other hashes passwords with the PwdHash algorithm [25] for transmission to an unmodified webserver.

We have been unable to find any commercially available keyboards or mice that enable programmable encrypted communication. However, myriad wireless keyboards do implement encrypted communication with their host adapter (e.g., encrypted Bluetooth packets are decrypted in the Bluetooth adapter's firmware, and not in software). Thus, the problem is not technical, but rather a reflection of the market's condition. Indeed, Microsoft's NGSCB was originally architected to depend on USB keyboards capable of encryption [8,23]. In our system, we have developed a USB Interposer using a low-power system-on-a-chip. Our USB Interposer supports a USB keyboard and mouse and manages encryption for use with Bumpy.

We have implemented Bumpy using an HP dc5750 with an AMD Athlon64 X2 at 2.2 GHz and a Broadcom v1.2 TPM as the user's computer, with a USB-powered BeagleBoard [4] containing a 600 MHz ARM CPU running embedded Linux serving as the USB Interposer. We use a Nokia E51 smartphone running Symbian OS v9.2 as the Trusted Monitor. Our USB Interposer supports encryption of all keyboard events, and mouse click events. Mouse movement events (i.e., X and Y delta information) are not encrypted, since only mouse clicks trigger *blur* events in the web browser GUI.

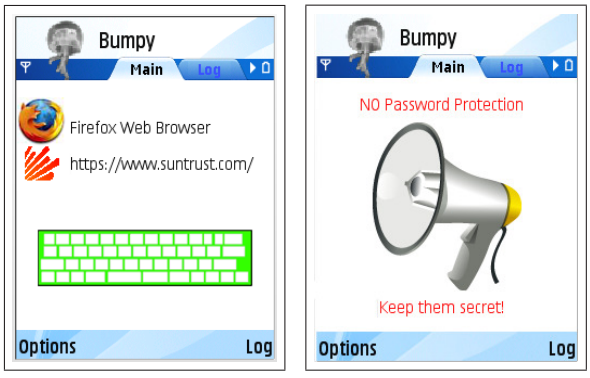
8.1 Bumpy Components

Our implementation includes the PreP and two PoPrs that run with Flicker's protections on the user's computer, the USB Interposer (BeagleBoard), the Trusted Monitor

running on a smartphone, and an untrusted web browser extension and Perl script. We begin by describing the components that are in Bumpy's TCB, and then treat the additional untrusted components that are required for availability (which we are forced to surrender since we consider the OS as untrusted).

PreP and PoPrs. We implemented the PreP as a Piece of Application Logic that runs with the protection of the Flicker system [18] and (1) receives encrypted keystroke events from the encrypting input device (i.e., the USB Interposer), (2) invokes one of our PoPrs to process the encrypted keystrokes for the webserver, either by re-encrypting them or performing the PwdHash [25] operation on passwords, and (3) sends encrypted messages to the Trusted Monitor that provide the favicon and domain of the active web page and PoPr. In our implementation, the PreP and both PoPrs are all part of the same PAL that runs using Flicker. An input parameter controls which PoPr is active.

USB Interposer. Our USB Interposer is built using a BeagleBoard featuring an OMAP3530 processor implementing the ARM Cortex-A8 instruction set [4], and a Prolific PL-25A1 USB-to-USB bridge [24]. We currently run embedded Debian Linux to benefit from the Linux kernel's mature support for both USB host and client operation. While this adds considerable code-size to our TCB, the interposer executes in relative isolation with a very specific purpose. We implement a small Linux application that receives all keyboard and mouse events (using the kernel's `evdev` interface), and encrypts all keyboard and mouse click events, letting mouse movement information pass in the clear. We describe the cryptographic protocol details in Section 8.2.



(a) Protection enabled visiting SunTrust bank. (b) Protection disabled.

Figure 4. Screenshots of the Trusted Monitor.

Trusted Monitor. We implemented a Symbian C++ application that runs on the Nokia E51 smartphone and serves as the Trusted Monitor. The Trusted Monitor updates its display in response to authenticated messages from the PreP, as described in Section 6. Figure 4 shows screen shots of the Trusted Monitor in action. When a session is active between the Trusted Monitor and PreP, the Trusted Monitor displays the domain name and favicon of the active web page's PoPr. It also displays a green keyboard (Figure 4(a)) as a unified indicator that protections are enabled. When input protections are disabled, it displays a warning message that input is unprotected and that `@@` should be used for sensitive input (Figure 4(b)). The Trusted Monitor uses distinctive beeps whenever input protections transition between enabled and disabled.

Note that after the initial configuration of the Trusted Monitor and PreP (Section 8.2), no further configuration is necessary during subsequent input sessions. The long-term symmetric keys encrypted under the master key that is kept in PCR-protected TPM NV-RAM will only be accessible to the correct PreP. Thus, only the PreP will be able to send authentic messages to the Trusted Monitor.

Untrusted Components. We developed an untrusted Firefox *Browser Extension* that communicates a web page's SSL certificate and embedded PoPr, and all *focus* events to the PreP. An untrusted Perl script facilitates communication between all components, manages the invocation of Flicker sessions, injects decrypted keystrokes into the OS using the Linux kernel's Uinput driver, and provides TPM *Quotes* in response to attestation requests. Note that the Flicker architecture provides the property that the code requesting the attestation from the TPM chip need not be trusted [18]. To convey encrypted data from the PreP to the USB Interposer, Trusted Monitor, or browser extension, the PreP must exit and release the ciphertext to the Perl script.

8.2 Secure Communication with the PreP

Both the USB Interposer and the Trusted Monitor require the ability to exchange secret, integrity-protected messages with the PreP. We implement the Flicker external communication protocol for both, with a trust-on-first-use model for accepting the respective public keys created in the PreP. Neither the USB Interposer nor the Trusted Monitor is pre-configured with knowledge of the identity of the TPM in the user's computer or the identity of the PreP installed on the user's computer.

We program a dedicated button on the USB Interposer to bootstrap association with a PreP, whereas the Trusted Monitor exposes a menu option to the user to connect to her computer to perform the initial configuration. The USB Interposer communicates with the user's computer via USB, and we use the AT&T 3G cellular network or WiFi to con-

nect the Trusted Monitor to the user's computer using a standard TCP/IP connection. An untrusted Perl script running on the user's computer handles reception of these messages and invokes Flicker sessions with the PreP so that the messages can be processed.

Both the USB Interposer and Trusted Monitor send a request to initiate an association with the PreP, passing in the command to bootstrap Flicker's external communication protocol [19], as well as a nonce for the subsequent attestation. The PreP then uses TPM-provided randomness to generate a 1024-bit RSA keypair. In accordance with Flicker's external communication protocol, the PreP extends PCR 17 with the measurement of its newly generated public key. The public key is then output from the PreP to be sent to the Trusted Monitor, and PCR 17 is *capped* (extended with a random value) to indicate the end of the Flicker session. At this point, PCR 17 on the user's computer contains an immutable record of the PreP executed and public key generated during execution.

8.2.1 PreP Authentication

Our use of a trust-on-first-use model to accept the PreP's public key dictates that no further verification of the exchanged keys is necessary. However, rigorous security goals may require the USB Interposer or Trusted Monitor to verify that the user's computer is running an approved PreP. In our current prototype, the USB Interposer and Trusted Monitor request a TPM attestation from the user's computer to ascertain the machine's public *Attestation Identity Key* (AIK) that it uses to sign attestations (TPM *Quotes* [34]), and the measurement (SHA-1 hash) of the PreP that will process input events. On subsequent connections, any change in the AIK or PreP measurement is an error. This way, it is readily extensible to allow application vendors to distribute signed lists of expected measurements, to leverage a PKI, or to a community-driven system similar in spirit to that of Wendlandt et al. (*Perspectives* [35]), and thus enable the USB Interposer and Trusted Monitor to validate the identity of the PreP themselves.

The USB Interposer and Trusted Monitor include a nonce with their initial connection requests, and expect a response that includes a TPM Quote over the nonce and PCR 17. The measurements extended into PCR 17[5] are expected to be the measurement of the PreP itself, the command to bootstrap external communication (`ExtCommCmd`), and the measurement of the public RSA key produced by the PreP:

$\texttt{PCR17} \leftarrow \texttt{SHA1(SHA1(SHA1(}0^{160}||\texttt{SHA1(PreP))}$
$||\texttt{SHA1(ExtCommCmd))}||\texttt{SHA1(PubKey))}.$

The USB Interposer and Trusted Monitor perform the same hash operations themselves using the measurement of the PreP, value of `ExtCommCmd`, and hash of the received public key. They then verify that the resulting hash matches the value of PCR 17 included in the TPM Quote.

[5]This example is specific to an AMD system. The measurements extended by Intel systems are similar.

8.2.2 Symmetric Key Generation for Communication with the PreP

We bootstrap secret and integrity-protected communication between the PreP and the USB Interposer or Trusted Monitor using the PreP's relevant public key to establish a shared master key K_{M1}. Separate symmetric encryption and MAC keys are derived for each direction of communication. We use AES with 128-bit keys in cipher-block chaining mode (AES-CBC) and HMAC-SHA-1 to protect the secrecy and integrity of all subsequent communication between the Trusted Monitor and the PreP. These keys form a part of the long-term state maintained by both endpoints.

$K_{aes1} \leftarrow \texttt{HMAC-SHA1(}K_{M1}\texttt{,'aes128.1')}^{128}$
$K_{hmac1} \leftarrow \texttt{HMAC-SHA1(}K_{M1}\texttt{,'hmac-sha1.1')}$
$K_{aes2} \leftarrow \texttt{HMAC-SHA1(}K_{M1}\texttt{,'aes128.2')}^{128}$
$K_{hmac2} \leftarrow \texttt{HMAC-SHA1(}K_{M1}\texttt{,'hmac-sha1.2')}$

8.2.3 Long-Term State Protection

The PreP must protect its state from the untrusted legacy OS while Flicker is not active. To facilitate this, the PreP generates a 20-byte master key K_{M2} using TPM-provided randomness. This master key is kept in PCR-protected non-volatile RAM (NV-RAM) on the TPM chip itself. We choose TPM NV-RAM instead of TPM Sealed Storage because of a significant performance advantage. The PCR 17 value required for access to the master key is that which is populated by the execution of the PreP using Flicker:

$\texttt{PCR17} \leftarrow \texttt{SHA1(}0^{160}||\texttt{SHA1(PreP))}.$

Flicker ensures that no code other than the precise PreP that created the master key will be able to access it [19]. Our PreP uses AES-CBC and HMAC-SHA-1 to protect the secrecy and integrity of the PreP's state while the (untrusted) legacy OS runs and stores the ciphertext. The necessary keys are derived as follows:

$K_{aes} \leftarrow \texttt{HMAC-SHA1(}K_{M2}\texttt{,'aes128')}^{128}$,
$K_{hmac} \leftarrow \texttt{HMAC-SHA1(}K_{M2}\texttt{,'hmac-sha1')}.$

This is sufficient to detect malicious changes to the saved state and to protect the state's secrecy. However, a counter is still needed to protect the freshness of the state and prevent roll-back or replay attacks. The TPM does include a monotonic counter facility [34], but it is only required to support updating once every five seconds. This is insufficient to keep up with user input. Instead, we leverage the sequence numbers used to order encrypted input events coming from the USB Interposer. The PreP is constructed such that a sequence number error causes the PreP to fall back to a challenge-response protocol with the USB Interposer, where the PreP ensures that it is receiving fresh events from the USB Interposer and reinitializes its sequence numbers.

Any sensitive input events that have been enqueued when a sequence number error takes place are discarded. Note that this should only happen when the system is under attack.

The USB Interposer and Trusted Monitor run on devices with ample non-volatile storage available.

8.3 The Life of a Keystroke

Here, we detail the path taken by keystrokes for a single sensitive web form field. It may be useful to refer back to Figures 2 and 3. At this point, symmetric cryptographic keys are established for bidirectional, secret, authenticated PreP-USB Interposer and PreP-Trusted Monitor communication. We now detail the process that handles keystroke events as the user provides input to a web page.

The user begins by directing focus to the relevant field, e.g., via a click of the mouse. On a well-behaved system, our browser extension initiates a Flicker session with the PreP, providing the name of the field, and the webserver's SSL certificate, PoPr (which includes the encryption key certificate $Cert_{ws\_enc}$), nonce, and favicon as arguments. The PreP verifies the SSL certificate using its CA list and verifies that the PoPr, nonce, and favicon are signed by the same SSL certificate. The user then types `@@` to indicate that the following input should be regarded as sensitive. The user's keystrokes travel from the keyboard to the USB Interposer, where they are encrypted for the PreP, and transmitted to the Perl script on the user's computer (Steps 1–3 in Figure 2). The script then initiates other Flicker sessions with the PreP, this time providing the encrypted keystrokes as input (Step 4 in Figure 2). The PreP decrypts these keystrokes and recognizes `@@` (Figure 3) as the sequence to indicate the start of sensitive input. The PreP outputs the `@` characters in plaintext and prepares a message for the Trusted Monitor to indicate the domain name and favicon of the current website and PoPr. The Trusted Monitor receives this message, beeps, and updates its display with the domain name and favicon.

Subsequent keystrokes are added to a buffer maintained as part of the PreP's long-term state. Dummy keystrokes (asterisks) are output for delivery to the legacy operating system (Step 5 in Figure 2) using the Uinput facility of the Linux kernel (which is also used when cleartext mouse and keyboard input events need to be injected). This enables the browser to maintain the same operational semantics and avoid unnecessary user confusion (e.g., by fewer asterisks appearing than characters that she has typed).

In the common case (after the long-term cryptographic keys are established), TPM-related overhead for one keystroke is limited to the TPM extend operations to initiate the Flicker session, and a 20-byte read from NV-RAM to obtain the master key protecting the sealed state. All other cryptographic operations are symmetric and performed by the main CPU. Section 9 offers a performance analysis.

When the user finishes entering sensitive input into a particular field, she switches the focus to another field. The PreP catches the relevant input event (a *Blur* in Figure 3) on the input stream, and prepares the sensitive input for handoff to the PoPr (Step 6 in Figure 2). We have implemented two PoPrs: encryption directly to the webserver, and PwdHash [25]. The PreP will then receive a *focus* event from the browser, indicating that focus has moved to another field. Note that form submission is a non-sensitive input event, so no special handling is required.

Encryption for Webserver. A widely useful PoPr encrypts the sensitive input for the remote webserver exactly as entered by the user (Steps 6–8 in Figure 2). This is accomplished using a public encryption key that is certified by the webserver's private SSL key. We use RSA encryption with PKCS#1v15 padding [15] to encrypt symmetric AES-CBC and HMAC-SHA-1 keys, which are used to encrypt and MAC the actual input with its corresponding field tags. The public encryption key is embedded in the PoPr.

Post-Processing as PwdHash. Another useful PoPr performs a site-specific transformation of data before submission to the webserver. We have implemented the PwdHash [25] algorithm in our PoPr. When this PoPr is active, the remote webserver need not be aware that Bumpy is in use, since the hashed password is output to the web browser as if it were the user's typed input. The PoPr manages the transformation from the user's sensitive password to a site-specific hash of the password, based on the domain name of the remote webserver.

8.4 The Webserver's Perspective

We now describe the process of acquiring sensitive input from the perspective of a Bumpy-enabled webserver. Prior to handling any requests, the webserver generates an asymmetric encryption keypair and signs the public key using its private SSL key (using calls to OpenSSL), resulting in $Cert_{ws\_enc}$. $Cert_{ws\_enc}$ can be used for multiple clients.

Our implementation consists of a Perl CGI script. When a request arrives at the webserver for a page that accepts user input, our CGI script is invoked to bundle $Cert_{ws\_enc}$ with a freshly generated nonce (for the upcoming attestation from the user's computer) and the hash and URL of the binary image of our direct-encryption PoPr. The ensuing bundle is then embedded into a hidden input field on the resulting web page. The hash and URL of the PoPr prevents wasting bandwidth on transferring the full PoPr unless it is the user's computer's first time employing this PoPr.

When the user submits the resulting page, the webserver expects to receive an attestation from the user's computer covering the PreP, the provided PoPr and nonce, and a public signing key (K_{PoPr_sig}) newly generated by the PoPr

on the user's computer. Currently, we employ trust-on-first-use to accept the Attestation Identity Key (AIK) that the user's computer's TPM used to sign the PCR register values. We have manually configured the webserver with the expected measurement of the PreP and PoPrs, as they are part of the same binary in our implementation. If the measurements in the attestation match the expected values, then K_{PoPr_sig} is associated with $K^{-1}_{ws\_enc}$ (and the user's computer's TPM's AIK) to enable decryption and authentication of subsequent strings of sensitive input encrypted by the PoPr.

9 Evaluation

We discuss the size of the trusted computing base (TCB) for our implementation, the performance impact on ordinary typing, webserver overhead, and the impact of network latency on the refresh rate of the Trusted Monitor's display.

Code Size. Bumpy provides strong security properties in part due to its small trusted computing base (TCB). Figure 5 shows the code size for our PreP and PoPrs, USB Interposer, webserver CGI script, and Trusted Monitor. Note that the TCB for the PreP and PoPrs includes *no additional code beyond the listed Flicker libraries* thanks to the properties of Flicker. Our current USB Interposer runs as a Linux application on a BeagleBoard; however, its only interface is the USB bridge to the user's computer, and its only function is to transmit encrypted keyboard and mouse events. Our Trusted Monitor includes Symbian OS in its TCB, as it runs as a normal smartphone application. We emphasize that the inclusion of Linux in the TCB of our USB Interposer and Symbian OS in the TCB of our Trusted Monitor is an artifact of our prototype implementation, and not a necessary consequence of our architecture.

PreP and PoPrs		
Func.	**Lang.**	**SLOC**
Main	.c	1044
PwdHash	.c	99
PwdHash	.h	4
Total	.c, .h	1147

Flicker libraries		
Func.	**Lang.**	**SLOC**
Crypto	.c	3980
Crypto	.h	471
TPM	.c	1210
TPM	.h	252
Util	.c	518
Util	.h	251
Util	.S	161
Total	.c, .h, .S	6854

USB Interposer		
Func.	**Lang.**	**SLOC**
Decode, Encrypt & TX	.c	489

Webserver CGI		
Func.	**Lang.**	**SLOC**
Embed & Verify	.pl	167

Trusted Monitor		
Func.	**Lang.**	**SLOC**
Protocol	.cpp	979
Protocol	.h	286
UI	.cpp	539
UI	.h	160
Util	.cpp	50
Util	.h	34
Total	.cpp, .h	2048

Figure 5. Lines of code for trusted Bumpy components obtained using SLOCCount [36]. The PreP and PoPrs include only the Flicker libraries in their software TCB. The USB Interposer, webserver, and Trusted Monitor also include their respective operating systems.

Typing Overhead with USB Interposer. We measured the round-trip-time between reception of a keypress on the USB Interposer (from the physical keyboard) and reception of an acknowledgement from the user's computer. This includes the time to encrypt and HMAC the keypress in the USB Interposer, send it to the user's computer via the USB-to-USB bridge, invoke the Flicker session on the user's computer with the PreP (unseal PreP state using the master key kept in PCR-protected TPM Non-Volatile RAM, decrypt and authenticate the newly arrived keypress, reseal PreP state, and release the new keypress to the OS), and send the acknowledgement back over the USB-to-USB bridge. In 500 trials, we experienced overhead of 141$\pm$15 ms (Figure 6). This is mildly noticeable during very fast typing, similar to an SSH session to a far-away host. It is noteworthy that the overhead consumed by Flicker (i.e., by the PreP) is 66$\pm$0.1 ms per keystroke, suggesting that more than half of the latency in our current prototype may be an artifact of the untrusted Perl script in our implementation. Indeed, the contribution of the Uinput driver used to inject keystrokes (42$\pm$8 ms) is uncharacteristically large, and grows over time. Writing to the driver from our Perl script presently involves the creation of a child process and a new virtual input device for every keystroke. The virtual input device driver was not designed

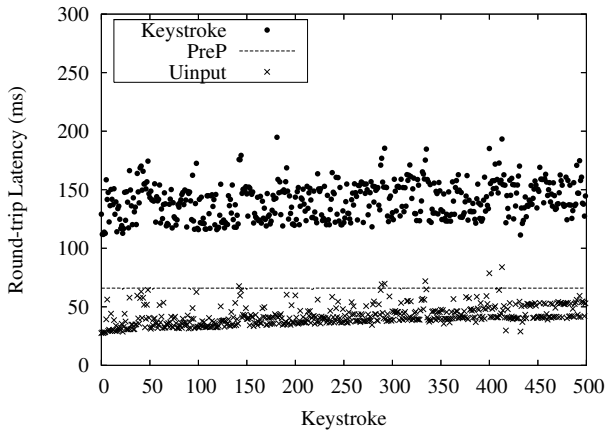


Figure 6. Latencies for 500 individual keystrokes. The PreP and Uinput latencies are components of the Keystroke latencies.

to scale so far. Our script should be modified to employ the same virtual input device throughout. Ample opportunities remain for optimization, which we plan to pursue in the course of future work in preparation for a user study.

Webserver Overhead with Encryption PoPr. With our direct-encryption PoPr enabled, the webserver must embed $Cert_{ws\_enc}$, a newly generated nonce, the hash of the desired PoPr, the URL at which the client system can obtain the PoPr, and a signature covering the favicon and all of these items into each page that may accept sensitive input. Our webserver is a Dell PowerEdge 2650 with two Intel Xeon 2.4 GHz CPUs running the Debian Linux flavor of Apache 2.2.3. In 25 trials, our CGI script induces a page-load latency of 17.0±0.4 ms, which is primarily composed of reading the cryptographic keys from disk (8.2±0.0 ms) and signing the nonce and metadata (8.6±0.5 ms). When the user submits the completed page, the webserver must verify an attestation from her platform. In 25 trials, our CGI script induces a form-submission latency of less than 2 ms to verify the signature on the attestation. Note that symmetric keys can be established that reduce the need for the signature-verification operation to a one-time overheads. Though we have not yet implemented this optimization, the only cost is a few tens of bytes of long-term state maintained on the user's computer and the webserver.

Trusted Monitor Network Latency. Our Trusted Monitor uses a TCP connection between the Nokia E51 smartphone and the user's computer. If there is significant network latency, then the Trusted Monitor may not be displaying the correct URL and favicon when the user looks at it. The smartphone can access the Internet using either its 3G/3.5G cellular radio, or using standard 802.11b/g wireless access points. To evaluate the latency impact of using these networks, we performed a simple echo experiment with an established TCP connection, where the E51 sends a series of 4-byte requests and receives 24-byte responses (excluding TCP/IP headers) from the HP workstation. We observed an average round-trip time (RTT) of 102±82 ms using the 802.11 network, and 211±25 ms using AT&T's 3.5G network. In our experience, these latencies are imperceptible to the user as she turns her head to look away from her primary display and towards the Trusted Monitor.

10 Discussion

We discuss design alternatives and other interesting features that Bumpy might be extended to offer.

10.1 Bumpy Design Alternatives

`@@` at Any Time. As presented, the secure attention sequence for Bumpy is the `@@` sequence immediately following a *focus* event from the web browser GUI. There are no technical limitations to enabling a secure attention sequence at any time, regardless of where in a field the cursor may be. However, we anticipate significant usability challenges for all but the most savvy users. This may prove to be an interesting direction for future work.

Editing Bumpy-Protected Input. As presented (Section 4.1), Bumpy ignores non-display characters that do not cause a *blur* event in the web browser GUI while the user is entering sensitive data. Examples of such characters are backspace and the arrow keys. Here too, there are no technical limitations to enabling the user to edit her opaque (from the browser's perspective) data. However, we are concerned about a malicious browser tampering with the cursor and confusing the user. Additional investigation is warranted to determine whether this attack amounts to anything beyond a denial-of-service attack (e.g., to get better data for a keystroke timing attack [32]).

Trusted Path Between Trusted Monitor and Webserver. There are many circumstances where the lack of a trusted path from a remote server to a user with a compromised computer can lead to the user's loss of sensitive information. For example, when a remote server checks an attestation from the user's computer and finds known malware installed, it is desirable to inform the user that her system is compromised. Other researchers have considered the use of PDAs or smartphones in such roles (e.g., Balfanz et al. [2]), but we consider this enhancement to Bumpy to be beyond the scope of the current paper.

PreP as Password Store. The direct-encryption PoPr breaks the web browser's ability to remember passwords on behalf of the user. This feature can be reenabled using the PreP or PoPr as a password store, and the Trusted Monitor as the interface to select a stored password.

10.2 Other Interesting Features

Password Leak Detection. A compelling feature that can readily be added to a PreP is to look for the user's password(s) in the input stream and detect whether it appears when input protections are not enabled. This may allow the system to issue a warning if, e.g., the user is about to fall victim to a phishing attack.

Hardware Keyloggers. Resistance to physical attacks is not an explicit goal of Bumpy; however, the issue warrants discussion. Bumpy's resilience to hardware keyloggers depends on the model used for associating new input devices with the user's computer. If a simple plug-and-play architecture is allowed, then a hardware keylogger inserted between the input device and the user's computer can appear as a new input device to the computer, and a new computer to the input device. One alternative is for input devices to require manufacturer certification before the user's computer will associate with them. However, this may prove to be impractical, as users may perceive all certification errors as indicative of a broken device. The core research challenge here is the problem of key establishment between devices with no prior context [3, 20, 33].

11 Conclusion and Future Work

We have described Bumpy, a system that protects users' sensitive input from keyloggers and screen scrapers by excluding the legacy OS and software stack from the TCB for input. Bumpy allows users to dictate which input is considered sensitive, thus introducing the possibility of protecting much more than just passwords. Bumpy allows webservers to define how input that their users deem sensitive is handled, and further allows users' systems to generate attestations that input protections are in place. With a separate local device, Bumpy can provide the user with a positive indicator that her input is protected. We have implemented Bumpy and show that it is efficient and compatible with existing legacy software.

We intend to continue the pursuit of a usable solution for protecting more sizeable input, e.g., composing a sensitive letter. We also plan to evaluate the current Bumpy architecture with a formal user study.

12 Acknowledgments

The authors would like to thank Karthik S. Lakshmanan and Anthony Rowe for their advice regarding embedded Linux systems and USB. Bryan Parno and Ahren Studer provided feedback and suggestions for the design, implementation, and writing. We are grateful for observations by the CyLab Student Seminar audience on October 17, 2008, and the Security Group Lunch audience at UNC on November 12, 2008. Unrestrained comments from our anonymous reviewers also proved helpful.

References

[1] Advanced Micro Devices. AMD64 architecture programmer's manual: Volume 2: System programming. AMD Publication no. 24593 rev. 3.14, Sept. 2007.

[2] D. Balfanz and E. W. Felten. Hand-held computers can be better smart cards. In *Proccedings of the USENIX Security Symposium*, Aug. 1999.

[3] D. Balfanz, D. 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)*, Feb. 2002.

[4] BeagleBoard.org. BeagleBoard revision B6 system reference manual revision 0.1. BeagleBoard.org, Nov. 2008.

[5] K. Borders and A. Prakash. Securing network input via a trusted input proxy. In *Proceedings of the USENIX Workshop on Hot Topics in Security (HotSec)*, Aug. 2007.

[6] E. Brickell, J. Camenisch, and L. Chen. Direct anonymous attestation. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, Oct. 2004.

[7] S. Chiasson, P. C. van Oorschot, and R. Biddle. A usability study and critique of two password managers. In *Proceedings of the USENIX Security Symposium*, Aug. 2006.

[8] P. England, B. Lampson, J. Manferdelli, M. Peinado, and B. Willman. A trusted open platform. *IEEE Computer*, 36(7):55–62, July 2003.

[9] N. Feske and C. Helmuth. A nitpicker's guide to a minimal-complexity secure GUI. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2005.

[10] E. Gabber, P. Gibbons, Y. Matias, and A. Mayer. How to make personalized web browsing simple, secure, and anonymous. In *Proceedings of Financial Cryptography*, 1997.

[11] E. Gabber, P. B. Gibbons, D. M. Kristol, Y. Matias, and A. Mayer. On secure and pseudonymous client-relationships with multiple servers. *ACM Trans. Inf. Syst. Secur.*, 2(4):390–415, 1999.

[12] IBM Zurich Research Lab. Security on a stick. Press release, Oct. 2008.

[13] Intel Corporation. Trusted execution technology – preliminary architecture specification and enabling considerations. Document number 31516803, Nov. 2006.

[14] M. Jakobsson and S. Myers. *Phishing and Countermeasures: Understanding the Increasing Problem of Electronic Identity Theft*. Wiley, Dec. 2006.

[15] J. Jonsson and B. Kaliski. PKCS #1: RSA cryptography specifications version 2.1. RFC 3447, Feb. 2003.

[16] B. Kauer. OSLO: Improving the security of Trusted Computing. In *Proceedings of the USENIX Security Symposium*, Aug. 2007.

[17] K. Kursawe, D. Schellekens, and B. Preneel. Analyzing trusted platform communication. In *Proceedings of the Cryptographic Advances in Secure Hardware Workshop (CRASH)*, Sept. 2005.

[18] J. M. McCune, B. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: An execution infrastructure for TCB minimization. In *Proceedings of the ACM European Conference in Computer Systems (EuroSys)*, Apr. 2008.

[19] J. M. McCune, B. Parno, A. Perrig, M. K. Reiter, and A. Seshadri. Minimal TCB code execution (extended abstract). In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2007.

[20] 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.

[21] J. M. McCune, A. Perrig, and M. K. Reiter. Bump in the ether: A framework for securing sensitive user input. In *Proceedings of the USENIX Annual Technical Conference*, June 2006.

[22] B. Parno. Bootstrapping trust in a "trusted" platform. In *Proceedings of the USENIX Workshop on Hot Topics in Security (HotSec)*, July 2008.

[23] M. Peinado, Y. Chen, P. England, and J. Manferdelli. NGSCB: A trusted open system. In *Proceedings of the Australasian Conference on Information Security and Privacy (ACISP)*, July 2004.

[24] Prolific Technology Inc. PL-25A1 hi-speed USB host to host bridge controller. PL-25A1 Product Brochure, Oct. 2006.

[25] B. Ross, C. Jackson, N. Miyake, D. Boneh, and J. C. Mitchell. Stronger password authentication using browser extensions. In *Proceedings of the USENIX Security Symposium*, Aug. 2005.

[26] A.-R. Sadeghi, M. Selhorst, C. Stüble, C. Wachsmann, and M. Winandy. TCG inside? - A note on TPM specification compliance. In *Proceedings of the ACM Workshop on Scalable Trusted Computing (STC)*, Nov. 2006.

[27] 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*, Aug. 2004.

[28] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, Sept. 1975.

[29] R. Sharp, A. Madhavapeddy, R. Want, and T. Pering. Enhancing web browsing security on public terminals using mobile composition. In *Proceeding of the Conference on Mobile Systems, Applications, and Services (MobiSys)*, June 2008.

[30] R. Sharp, A. Madhavapeddy, R. Want, T. Pering, and J. Light. Fighting crimeware: An architecture for split-trust web applications. Technical Report IRC-TR-06-053, Intel Research Center, Apr. 2006.

[31] R. Sharp, J. Scott, and A. Beresford. Secure mobile computing via public terminals. In *Proceedings of the International Conference on Pervasive Computing*, May 2006.

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

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

[34] Trusted Computing Group. Trusted platform module main specification, Part 1: Design principles, Part 2: TPM structures, Part 3: Commands. Version 1.2, Revision 103, July 2007.

[35] D. Wendlandt, D. G. Andersen, and A. Perrig. Perspectives: Improving SSH-style host authentication with multi-path probing. In *Proceedings of the USENIX Annual Technical Conference*, June 2008.

[36] D. A. Wheeler. Linux kernel 2.6: It's worth more! Available at: `http://www.dwheeler.com/essays/linux-kernel-cost.html`, Oct. 2004.

A TCG-Style Attestation and Sealed Storage

The v1.2 Trusted Platform Module (TPM) chip contains an array of 24 or more Platform Configuration Registers (PCRs), each capable of storing a 160-bit hash. These PCRs can be *Extended* with a *Measurement* (cryptographic hash) of data, such as a program binary. Given a measurement `m` $\leftarrow$ `SHA1(data)`, the extend process works as follows: $\texttt{PCR}_{new} \leftarrow \texttt{SHA1}(\texttt{PCR}_{old}||\texttt{m})$.

TPMs include two kinds of PCRs: static and dynamic. Static PCRs reset to 0^{160} when the TPM itself resets (generally during a full platform reset or power-cycle, although physical TPM-reset attacks have been demonstrated [16,17, 26]), and can only have their value updated via an Extend operation. These PCRs can be used to keep a record of measurements for all software loaded since the last reboot, as in IBM's Integrity Measurement Architecture [27].

Dynamic PCRs are present in v1.2 TPMs, and are relevant when the platform supports Dynamic Root of Trust, e.g., Intel TXT [13] or AMD SVM [1]. Dynamic PCRs reset to 1^{160} during full platform reset, and can additionally be reset to 0^{160} via a *Late Launch*, thereby establishing a *Dynamic* Root of Trust. In addition to resetting the dynamic PCRs, Late Launch resets the CPU to a known trusted state without rebooting the rest of the system. This includes configuring the system's memory controller to prevent access to the launching code from DMA-capable devices. One of the newly reset dynamic PCRs is then automatically extended with a measurement of the software that will get control following the Late Launch [1]. This enables software to bootstrap without including the BIOS or any system peripherals in the TCB. The Open Secure LOader (OSLO) performs a Late Launch on AMD systems to remove the BIOS from the TCB of a Linux system [16]. Trusted Boot[6]

[6] `http://sourceforge.net/projects/tboot`

from Intel performs similarly for Intel hardware, though it adds the ability to enforce a Launch Control Policy. The Flicker system uses Late Launch to briefly interrupt the execution of a legacy OS and execute a special-purpose code module in isolation from all other software and devices on the platform, before returning control to the legacy OS [18].

Once measurements have accumulated in the PCRs, they can be *attested* to a remote party to demonstrate what software has been loaded on the platform. They can also be used to *seal* data to a particular platform configuration. We discuss each of these in turn.

Attestation. The attestation process involves a challenge-response protocol, where the challenger sends a cryptographic nonce (for replay protection) and a list of PCR indexes, and requests a *TPM Quote* over the listed PCRs. A Quote is a digital signature computed over an aggregate of the listed PCRs using an *Attestation Identity Key* (AIK). An AIK is an asymmetric signing keypair generated on the TPM. We discuss certification of AIKs shortly. The messages exchanged between a challenger C and an untrusted system U to perform an attestation are:

$C \rightarrow U$: `nonce, PCRindexes`
$U \rightarrow C$: `PCRvals,` $\{$`PCRvals, nonce`$\}_{AIK^{-1}}$

Once the challenger receives the attestation response, it must (1) verify its nonce is part of the reply, (2) check the signature with the public AIK obtained via an authentic channel, (3) verify that the list of PCR values received corresponds to those in the digital signature, and (4) verify that the PCR values themselves represent an acceptable set of loaded software. Note that since the sensitive operations for a TPM Quote take place entirely within the TPM chip, the TPM Quote operation can safely be invoked from untrusted software. The only attack available to malicious software is denial-of-service. In the context of the Flicker system, this removes the code that causes the TPM Quote to be generated from the system's TCB.

Certifying Platform Identity. The Attestation Identity Keypair (AIK) used to perform the TPM Quote effectively represents the identity of the attesting host. We discuss options for certifying this keypair (i.e., obtaining an authentic copy of the public AIK for a particular physical host).

Multiple credentials are provided by TPM and host manufacturers that are intended to convince a remote party that they are communicating with a valid TPM installed in a host in conformance with the relevant specifications [34]. These are the TPM's Endorsement Key (EK) Credential, Platform Credential, and Conformance Credential. One option is to use these credentials directly as the host's identity, but the user's privacy may be violated. Motivated by privacy concerns, the Trusted Computing Group (TCG) has specified Privacy Certificate Authorities (Privacy CAs). Privacy CAs are responsible for certifying that an AIK generated by a TPM comes from a TPM and host with valid Endorsement Key, Platform, and Conformance Credentials.

To the best of our knowledge, there are no commercial Privacy CAs in operation today. Thus, we must either provide all of the credentials corresponding to the untrusted host to the challenger (compromising privacy), or the challenger must blindly accept the AIK without performing any verification (compromising host identity, and adopting the trust-on-first-use model). Trust-on-first-use models have been deployed successfully, e.g., for the Secure Shell (SSH) protocol. Thus, we believe the choice of which host identity mechanism to use is application-dependent. For communication with a bank or established online merchant, where an honest user almost always provides her true identity, it is not clear that there is any loss of privacy by providing the full set of TPM and host credentials.

Direct Anonymous Attestation (DAA) has also been proposed as an alternative to Privacy CAs for protecting platform identity [6]. To the best of our knowledge, no systems are available today that include TPMs supporting DAA.

Sealed Storage. TPM-protected sealed storage is a mechanism by which an asymmetric encryption keypair can be bound to certain PCR values. Data encrypted under this keypair then becomes unavailable unless the PCR values match those specified when the data was sealed. This is a relatively slow process since the asymmetric cryptographic operations are performed by the low-cost CPU inside the TPM. An alternative is to use the TPM's Non-Volatile RAM (NV-RAM) facility. NV-RAM can be configured with similar properties to sealed storage, in that a region of NV-RAM can be made inaccessible unless the PCR values match those specified when the region was defined. NV-RAM has a limited number of write cycles during the TPM's lifetime, but the use of a symmetric master key that is only read from NV-RAM in the common case can greatly extend its life. Flicker can use TPM sealed storage or NV-RAM to protect long-term state that is manipulated during Flicker sessions.