Fidelius: Protecting User Secrets from Compromised Browsers

Saba Eskandarian[1], Jonathan Cogan[1], Sawyer Birnbaum[1], Peh Chang Wei Brandon[1],
Dillon Franke[1], Forest Fraser[1], Gaspar Garcia[1], Eric Gong[1], Hung T. Nguyen[1],
Taresh K. Sethi[1], Vishal Subbiah[1], Michael Backes[2], Giancarlo Pellegrino[1,3], and Dan Boneh[1]

[1]Stanford University
[2]CISPA Helmholtz Center i.G., Saarland Informatics Campus
[3]CISPA, Saarland University, Saarland Informatics Campus

***Abstract*—Users regularly enter sensitive data, such as passwords, credit card numbers, or tax information, into the browser window. While modern browsers provide powerful client-side privacy measures to protect this data, none of these defenses prevent a browser compromised by malware from stealing it. In this work, we present Fidelius, a new architecture that uses trusted hardware enclaves integrated into the browser to enable protection of user secrets during web browsing sessions, even if the *entire underlying browser and OS* are fully controlled by a malicious attacker.**

Fidelius solves many challenges involved in providing protection for browsers in a fully malicious environment, offering support for integrity and privacy for form data, JavaScript execution, XMLHttpRequests, and protected web storage, while minimizing the TCB. Moreover, interactions between the enclave and the browser, the keyboard, and the display all require new protocols, each with their own security considerations. Finally, Fidelius takes into account UI considerations to ensure a consistent and simple interface for both developers and users.

As part of this project, we develop the first open source system that provides a trusted path from input and output peripherals to a hardware enclave with no reliance on additional hypervisor security assumptions. These components may be of independent interest and useful to future projects.

We implement and evaluate Fidelius to measure its performance overhead, finding that Fidelius imposes acceptable overhead on page load and user interaction for secured pages and has *no impact* on pages and page components that do not use its enhanced security features.

***Index Terms*—Browser Security, Trusted I/O, Hardware enclave, Malware protection.**

I. Introduction

In recent years there has been a tremendous growth in malware that infects end-user machines with the explicit goal of stealing sensitive data that users enter into their browser window. Some recent examples include TrickBot and Vega Stealer, which are man-in-the-browser malware designed to steal banking credentials and credit card numbers. Generally speaking, once malware infects the user's machine, it can effectively steal all user data entered into the browser. Modern browsers have responded with a variety of defenses aimed at ensuring browser integrity. However, once the machine is compromised, there is little that the browser can do to protect user data from a key logger.

In this paper we present a practical architecture, called Fidelius, that helps web sites ensure that user data entered into the browser cannot be stolen by end-user malware, no matter how deeply the malware is embedded into the system. When using Fidelius, users can safely enter data into the browser without fear of it being stolen by malware, provided that the hardware enclave we use satisfies the security requirements.

Hardware enclaves, such as Intel's SGX, have recently been used to provide security for a variety of applications, even in case of compromise [1]–[15]. An enclave provides an execution environment that is isolated from the rest of the system (more on this below). Moreover, the enclave can attest its code to a remote web site.

One could imagine running an entire browser in an enclave to isolate it from OS-level malware, but this would be a poor design – any browser vulnerability would lead to malware inside the enclave, which would completely compromise the design.

A. Our Contributions

Fidelius contains three components, discussed in detail in the following sections: (1) a small trusted functionality running inside an isolated hardware enclave, (2) a trusted path to I/O devices like the keyboard and the display, and (3) a small browser component that interacts with the hardware enclave.

A trusted path from the hardware enclave to I/O devices is essential for a system like Fidelius. First, this is needed to prevent an OS-level malware from intercepting the data on its way to and from the I/O device. More importantly, the system must prevent out-of-enclave malware from displaying UI elements that fool the user into entering sensitive data where the malware can read it. Beyond protecting web input fields, the system must protect the entire web form to ensure that the malware does not, for example, swap the "username" and "password" labels and cause the user to enter her password into the username field.

We implement a prototype trusted path to the keyboard using a Raspberry Pi Zero that sits between the user's machine and the keyboard and implements a secure channel between

Corresponding Author: saba@cs.stanford.edu

the keyboard and the hardware enclave. We implement a trusted path to the display using a Raspberry Pi 3 that sits between the graphics card and the display. The Raspberry Pi 3 overlays a trusted image from the hardware enclave on top of the standard HDMI video sent to the display from the graphics card. We discuss details in Section IX-A. Our trusted path system is open source and available for other projects to use. We note that we can not use SGXIO [16], an SGX trusted I/O project, because that system uses hypervisors, which may be compromised in our threat model.

Another complication is the need to run client-side JavaScript on sensitive form fields. For example, a web site may use client-side JavaScript to ensure that a credit card checksum is valid, and alert the user if not. Similarly, many sites use client-side JavaScript to display a password strength meter. Fidelius should not prevent these scripts from performing as intended. Several projects have already explored running a JavaScript interpreter in a hardware enclave. Examples include TrustJS [17] and Secureworker [18]. Our work uses the ability to run JavaScript in an enclave as a building block to enable privacy for user inputs in web applications. The challenge is to do so while keeping the trusted enclave – the TCB – small.

To address all these challenges, this paper makes the following contributions:

- The design of Fidelius, a system for protecting user secrets entered into a browser in a fully-compromised environment.
- A simple interface for web developers to enable Fidelius's security features.
- The first open design and implementation of a trusted path enabling a hardware enclave to interact with I/O devices such as a display and a keyboard from a fully compromised machine.
- A browser component that enables a hardware enclave to interact with protected DOM elements while keeping the enclave component small.
- An open-source implementation and evaluation of Fidelius for practical use cases.

Related work. The problem of safely entering data into an end-user application has been studied extensively, and we survey the related work in Section XII. We note that Microsoft's 2003 NGSCB [19] architecture, also known as Palladium, was a commercial attempt at a solution. However, none of the proposed systems were deployed widely, despite the importance of the problem. We believe that the new hardware capabilities used in Fidelius make it a practical and deployable solution that can be integrated into existing web sites with relatively little effort.

Security of hardware enclaves. We built Fidelius using the hardware enclave provided by Intel's SGX. SGX has recently come under several side-channel attacks [20], [21], making the current implementation of SGX insufficiently secure for Fidelius. However, Intel is updating SGX using firmware and hardware updates with the goal of preventing these side-channel attacks. In time, it is likely that SGX can be made sufficiently secure to satisfy the requirements needed for Fidelius. Even if not, other enclave architectures are available, such as Sanctum for RISC-V [22] or possibly a separate co-processor for security operations.

II. Trusted Hardware Background

A *hardware enclave* provides developers with the abstraction of a secure portion of the processor that can verifiably run a trusted code base (TCB) and protect its limited memory from a malicious or compromised OS [23], [24]. The hardware handles the process of entering and exiting an enclave and hiding the activity of the enclave while non-enclave code runs. Enclave code invariably requires access to OS resources such as networking and user or file I/O, so developers specify an interface between the enclave and the OS. In SGX, the platform we use for our implementation, the functions made available by this interface are called *OCALLs* and *ECALLs*. OCALLs are made from inside the enclave to the OS, usually for procedures requiring resources managed by the OS, such as file access or output to a display. ECALLs allow code outside the TCB to call the enclave to execute trusted code.

An enclave proves that it runs an untampered version of the desired code through a *remote attestation* mechanism. Attestation loosely involves an enclave providing a signed hash of its initial state (including the running code), which a server compares with the expected value and rejects if there is any evidence of a corrupted program. In order to persist data to disk when an enclave closes or crashes, SGX also provides a data *sealing* functionality that encrypts and authenticates the data for later recovery by a new instance of the enclave.

Finally, one of the key features of enclaves is the protection of memory. An enclave gives developers a small Enclave Page Cache (EPC), a memory region hidden from the OS and cleared when execution enters or exits an enclave. In this memory, the trusted code can keep secrets from an untrusted OS that otherwise controls the machine. SGX provides approximately 90MB of EPC. Unfortunately, a number of side-channel attacks have been shown to break the abstraction of fully-protected enclave memory. We briefly discuss these attacks and accompanying defenses in Section XII.

III. Threat Model

We leverage a trusted hardware enclave to protect against a network attacker who additionally has full control of the operating system (OS) on the computer running Fidelius. We assume that our attacker has the power to examine and modify unprotected memory, communication with peripherals/network devices, and communication between the processor and enclave. Moreover, it can maliciously interrupt the execution of an enclave. Note that an OS-level attacker can always launch an indefinite denial of service attack against an enclave, but such an attack does not compromise privacy.

We assume that the I/O devices used with the computer are not compromised and that the dongles we add to keyboards/displays follow the behavior we describe. We could assume that there is a trusted initial setup phase where the devices can exchange keys and other setup parameters with the enclave. This corresponds to a setting where a user buys a new computer, sets it up with the necessary peripherals, and then connects to the internet, at which point the machine immediately falls victim to malware. Alternatively, this honest setup assumption could easily be avoided with an attestation/key exchange step between the peripherals and the enclave. We discuss both options in Section VI-A.

Overview of Security Goals. We would like to provide the security guarantee that any user data entered via a trusted input will never be visible to an attacker, and, except in the case of denial of service, the data received by the server will correspond to that sent by the user, e.g. it will not be modified, shuffled, etc. Moreover, the enclave will only send data to an authenticated server, and a server will only send data to a legitimate enclave. Finally, we wish for all the low-level protocols of our system to be protected against tampering, replay, and other attacks launched by the compromised OS.

The remote server in our setting cooperates to secure the user by providing correct web application code to be run in the enclave. We are primarily concerned with the security of user secrets locally on a compromised device, but this does include ensuring that secrets are not sent out to an attacker.

Overview of Usability Goals. Although our work is merely a prototype of Fidelius, we intend for it to be fully functional and to defend not only against technical attacks on security but also against user interface tricks aiming to mislead a user into divulging secrets to a malicious party. This task looms particularly important in our mixed setting where trusted input/output come through the same channels as their untrusted counterparts. In particular, we must make sure a user knows whether the input they are typing is protected or not, what data the remote server expects to receive, and where the private data will eventually be sent. We leave the task of optimizing the user experience to future work, but also aim to provide a tool which can be used "as-is."

We also want to provide a usable interface for developers that deviates only minimally from standard web development practices. As such, we endeavor to add only the minimal extensions or limitations to current web design techniques to support our security requirements.

Enumeration of Attacks. After describing the system in detail in subsequent sections, we discuss why Fidelius satisfies our security goals. Here we briefly list the different classes of non-trivial attacks against which we plan to defend. Refer to Section VIII for details on the attacks and how we defend against them.

- *Enclave omission attack*: The attacker fakes use of an enclave.
- *Enclave misuse attack*: The attacker abuses Enclave ECALLs for unexpected behavior.
- *Page tampering attack*: The attacker modifies protected page elements or JavaScript.
- *Redirection attack*: The attacker fakes the origin to which trusted data is sent.
- *Storage tampering attack*: The attacker reads, modifies, deletes, or rolls back persistent storage.
- *Mode switching attack*: The attacker makes unauthorized entry/exits from private keyboard mode.
- *Replay attack*: The attacker replays private key presses or display overlays.
- *Input manipulation attack*: The attacker forges or manipulates placement of protected input fields.
- *Timing attack*: The attacker gains side-channel information from the timing of display updates or keyboard events.

Security Non-Goals. Fidelius provides the tools necessary to form the basis of a secure web application, focusing on protecting user inputs and computation over them. We do not provide a full framework for secure web applications or a generic tool for protecting existing web applications. In particular, we do not protect against developers who decide to run insecure or leaky JavaScript code inside an enclave, but we do provide a simple developer interface to protect security-critical components of applications.

We assume the security of the trusted hardware platform and that the enclave hides the contents of its protected memory pages and CPU registers from an attacker with control of the OS, so side channel attacks on the enclave [20], [21] are also out of the scope of this work. We discuss side channel attacks and mitigations for SGX in Section XII. Physical attackers who tamper with the internal functionality of our devices also lie outside our threat model, but we note that our trusted devices seem to be robust against *opportunistic* physical attackers that do not tamper with hardware internals but can, for example, attach a usb keylogger to a computer. The SGX hardware itself is also designed to resist advanced hardware attackers.

Finally, we do not address how the honest server protects sensitive data once the user's inputs reach it. Our goal is to protect data from compromise on the client side or in transit to the server. Once safely delivered to the correct origin, other measures must be taken to protect user data. For example, we do not defend against a server who receives secrets from the user and then displays them in untrusted HTML sent back to the browser.

IV. Architecture Overview

The goal of Fidelius is to establish a trusted path between a user and the remote server behind a web application. To achieve this goal, Fidelius relies on two core components: a trusted user I/O path and a *web enclave*. In practice, this involve subsystems for a secure keyboard, a secure video display, a browser component to interact with a hardware enclave, and the enclave itself. Figure 1 gives an overview of the components of Fidelius.

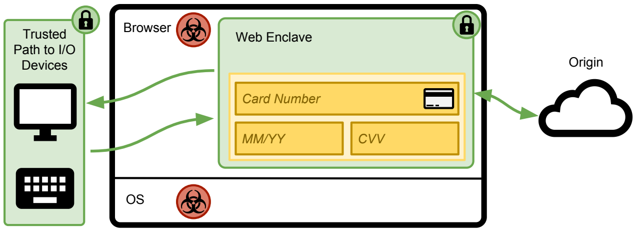

Fig. 1. Overview of Fidelius. The web enclave, embedded in a malicious browser and OS, communicates with the user through our trusted I/O path and securely sends data to a remote origin. We assume that both the web browser and the OS are compromised.

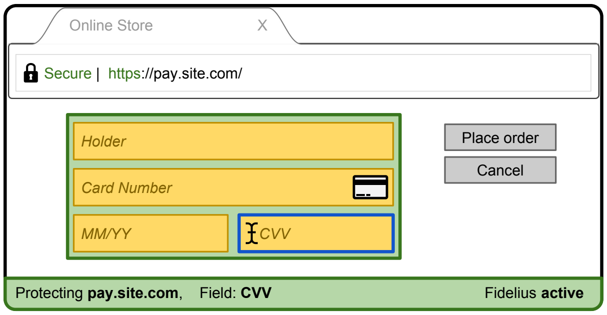

Fig. 2. Design of Fidelius's user interface. The green area is the trusted display overlay.

A. Trusted User I/O Path

The trusted user I/O path consists of a keyboard and display with a trusted dongle placed between them and the computer running Fidelius. Each device consists of trusted and untrusted modes. The untrusted modes operate exactly the same as in an unmodified system. The trusted keyboard mode, when activated, sends a constant stream of encrypted keystrokes to the enclave. The enclave decrypts and updates the state of the relevant trusted input field. The trusted and untrusted display modes are active in parallel, and the trusted mode consists of a series of overlays sent encrypted from the enclave to the display. Overlays include rendered DOM subtrees (including, if any, the protected user inputs) placed over the untrusted display output as well as a dedicated portion of the screen inaccessible to untrusted content. We cover these functionalities and details of the protocols used to secure them in Section VI. Finally, both trusted devices have LEDs that notify the user when a trusted path is established and ready to collect user input. Our system relies, in part, on users not typing secrets on the keyboard when these security indicator lights are off. This ensures that only the enclave has access to secrets entered on the keyboard. We note, however, that several works have studied the effectiveness of security indicators in directing user behavior [25], [26] and found that users often ignore them. We briefly discuss potential alternatives in Section XI, but leave the orthogonal problem of designing a better user interface – one that is more difficult to ignore – to future work.

B. Web Enclave

A web enclave is essentially a hardware enclave running a minimalistic, trusted browser engine bound to a single web origin. A browser using a web enclave delegates the management and rendering of portions of a DOM tree and the execution of client-side scripts, e.g. JavaScript and Web Assembly, to the enclave. In addition, the web enclave can send and receive encrypted messages to and from trusted devices and the origin server. Finally, the web enclave provides client-side script APIs to access the DOM subtree, secure storage, and secure HTTP communication.

When a user loads a web page, Fidelius checks whether the page contains HTML tags that need to be protected, e.g., secure HTML forms. If it does, it initiates a web enclave, runs remote attestation between that enclave and the server, and validates the identity of the server. Once this process completes, Fidelius loads the HTML tags it needs to protect into the web enclave and verifies their signatures. Then, when the user accesses a protected tag, e.g. with a mouse click, Fidelius gives control to the enclave, which in turn activates the devices' trusted mode. The trusted mode LEDs are turned on, informing the user that the trusted path is ready to securely collect user input.

Web enclaves provide two main ways to send protected messages to a remote server: directly through an encrypted form submission or programmatically via an XMLHttpRequest API. When a user clicks a form's submit button, the web browser notifies the enclave of this event. Then, the web enclave encodes the form data following HTML form norms[1], encrypts that data, and signs it. The encrypted form is passed to the web browser, which sends it to the remote server. When a script needs to send messages to the server, it can use the XMLHttpRequest web API. The web enclave XMLHttpRequest API interface is similar to that implemented by web browsers; however, it encrypts sensitive fields such as the request body and custom HTTP headers. HTTP responses are sent by the server in encrypted form. The enclave will automatically decrypt responses and resume execution of the JavaScript function waiting for the response.

V. Interface Design

This section describes the interfaces that Fidelius provides for end-users and developers who wish to consume or create protected web applications. Here we describe only how Fidelius appears to users and developers, deferring technical details of how it works to subsequent sections.

A. User Interface

The primary challenge in designing an interface for a system with a mix of trusted and untrusted components lies in distinguishing the trusted parts from the untrusted parts in a way that cannot be faked by an attacker. Our solution is to dedicate a small part of the screen to the web enclave,

[1] See, https://www.w3.org/TR/html5/sec-forms.html

rendering that portion of the screen inaccessible to the OS while the trusted display is active, as indicated by an LED outside the display. Outside of this region, user interaction with Fidelius does not differ at all from interactions with a typical web application. Figure 2 shows the design of Fidelius's user interface in use on a sample payment page. Trusted input fields do not have any special visual features that distinguish them from other inputs. Instead, the dedicated trusted region of the screen displays information that defends against attacks which make use of UI manipulation to fool a user into giving sensitive data to an attacker.

There are two important pieces of information shown in the protected display region. First, we must ensure that the user sends sensitive information only to the intended destination and avoids attacks like changing the contents of the url bar or picture-in-picture attacks [27]. We achieve this by including the origin of the web enclave in the trusted region. In Figure 2, the trusted region shows that the web enclave is connected to `pay.site.com`.

Second, we must ensure that users can distinguish real trusted inputs from untrusted ones and that an attacker cannot fool the user by changing the untrusted text surrounding a trusted input field. This could include attacks where untrusted input fields are made to look just like trusted ones (which in fact is the case by default in Fidelius) or, for example, where the username and password prompts before two inputs are switched, causing the user's password to be processed as a username, which potentially receives far less protection after being sent to the server. We protect against this class of attacks by displaying a name for each trusted input in the dedicated display region when that field has focus. This serves to indicate to the user that the current input field is trusted. It also protects against any attack involving shuffling of input field labels to fool a user or cause incorrect data to be sent to the server because the descriptive name for each input field lies outside the reach of an attacker.

B. Developer Interface

Design of a developer interface must provide an easy to use and backwards compatible way for developers to access the features of Fidelius. Our developer interface requires *no changes* for pages or components of pages that do not make use of Fidelius's features. Developers who wish to provide stronger security guarantees to Fidelius users include additional attributes in existing HTML tags directing Fidelius to use the web enclave in rendering and interacting with the content of those tags. Listing 1 shows an example of an HTML page supporting Fidelius.

```html
<html>
 <head> [...] </head>
 <body>
  <form action="submit_data"
        name="payment"
        method="POST"
        secure="True" sign="tX5ReRzE42Qw">
    <input type="text"
           value="Holder" name="holder" />
    <input type="text"
           value="Card Number" name="card"/>
    <input type="text"
           value="MM/YY" name="exp"/>
    <input type="text"
           value="CVV" name="cvv"/>
  </form>
  <div class="btn"><p>Place order</p></div>
  <div class="btn"><p>Cancel</p></div>
  <script type="text/JavaScript"
          src="validator.js"
          secure="True" sign="Fi3Rt9mq2ff0">
  </script>
 </body>
</html>
```

Listing 1. Fidelius-enabled code for the online payment web page. In red, the new HTML attributes required by Fidelius.

Fidelius currently supports `<form>`, `<input>`, and `<script>` tags. To mark any of these tags as compatible with Fidelius, developers add a `secure` attribute to the tag. In the case of `<script>` and `<form>` tags, a signature over the content of the tag is included in a `sign` attribute, to be verified with respect to the server's public key inside the enclave as described in Section VII. The signature ensures that the form and script contents have not been modified by malware before they were passed to the enclave. The signature is not needed for `<input>` tags because the signature on a form includes the inputs contained within it. `<input>` tags also require a `name` attribute to be shown in the trusted component of the display when that input has focus.

JavaScript included in secure `<script>` tags runs on an interpreter inside the web enclave with different scope than untrusted code running in the browser. Trusted JavaScript has access to its own memory and its own web APIs for secure storage and secure HTTP requests, but it cannot directly access the memory or web APIs available to untrusted JavaScript. Trusted and untrusted JavaScript can, however, make calls to each other and pass information between each other as needed using an interface similar to the `postMessage` cross-origin message passing API.

Fidelius enforces a strict same-origin policy for web enclaves, so network communication originating or ending in an enclave can only come from its specified origin. By default, the origin of HTML tags is inherited from the web page. In general, the origin is derived from the initial URL of the page. However, for tags such as `<form>` and `<script>`, the origin is derived from the `action` and `src` attributes respecively. The origin specified here is not authenticated and therefore susceptible to tampering. We discuss the process by which a web enclave connects to remote servers and verifies their legitimacy in Section VII.

VI. Trusted Path for User I/O

In this section, we describe the building blocks to create and manage a trusted path connecting a keyboard, display, and web enclave. Specifically, we cover device setup, communication patterns between devices, and the structure of individual messages passed between devices.

Although we develop our trusted I/O path in the context of the larger Fidelius system and focus our discussion on web applications, it is important to note that the trusted path is fundamentally a separate system from the web enclave. In other words, although the two systems interact closely in the design of Fidelius, the trusted path has applications outside the web and can be run on its own as well. To our knowledge, this is the first system to provide a trusted path to the user for both input and output relying only on assumptions about enclave security. We cover the details of how we realize the trusted peripherals in hardware dongles in Section IX.

A. Setup

In order to securely communicate, the web enclave and peripherals (or the dongles connected to them) must have a shared key. One option is to operate in a threat model with an initial trusted phase where we assume the computer is not yet compromised. Pre-shared keys are exchanged when the user configures the computer for the first time. Devices store the key in an internal memory, and the enclave seals the shared keys for future retrieval. The key can be accessed only by the enclave directly and not by user-provided JavaScript running inside it.

In the more realistic setting where new peripherals can be introduced to a computer over time, we must protect against attacks that involve introduction of malicious periphal devices. In this setting, we need Fidelius-compatible devices to include a trusted component that can perform an attestation with the enclave to prove its legitimacy before exchanging keys.

B. Trusted Communication

The process of switching between trusted and untrusted modes presents an interesting security challenge. An authentication procedure between the enclave and the trusted devices can ensure that only the enclave initiates switches between trusted and untrusted modes, but this ignores the larger problem that the enclave must rely on the untrusted OS to inform it when an event has happened that necessitates switching modes. Avoiding that necessity would require moving a prohibitively large fraction of the browser and UI into an enclave. Our solution has two parts and relies on making the user aware of when key presses produce trusted or untrusted input. First, we include a light on each dongle that turns on only when the keyboard or display are in trusted mode. This alone, however, does not suffice to solve the problem, as an attacker could mount a "rapid switching" attack where it jumps in and out of trusted mode faster than the user can perceive or react, leading to parts of the user's input being leaked by untrusted input. Even worse, rapid switching between modes may occur quickly enough to not be noticable to a user monitoring the lights. To prevent this attack, we force a short delay when switching out of trusted mode. This ensures the user will have time to notice and react when a switch occurs.

The enclave switches devices in and out of trusted mode by sending one of two reserved messages which are simply fixed strings that they interpret as commands to change the trust setting. When in trusted mode, messages between the enclave and the peripherals are encrypted as described in Section VI-C.

Since the timing of key presses can reveal sensitive information about what keys are being pressed [28], we must also avoid leaking timing information while in trusted input mode. We do this by having the keyboard send a constant stream of key presses where most contain only an encryption of a dummy value that indicates no key pressed. As long as the fixed frequency of key presses exceeds the pace at which a user types, the user experience is unaffected by this protection. Since user key presses typically result in changes on the display, we update the display contents at the same rate as we read keyboard inputs.

Our trusted input design in many ways mirrors that of Bumpy [29] and SGX-USB [30], which also provide generic trusted user input using similar techniques but do not provide the web functionality that we do. In contrast to our work, Bumpy does not display any trusted user input. SGX-USB allows for generic I/O but does not solve the problem of mixing trusted and untrusted content in a user interface as we do in both our keyboard and display. Neither system has source code available. We improve on the features of both works by protecting against timing attacks on encrypted data sent from trusted input devices.

C. Message structure

Messages sent in the trusted communication protocol described above must include safeguards against replay attacks. To do this, we include a counter in every message sent, so that the same count never repeats twice. Counters are maintained on a per-device and per-origin basis, so every message between the enclave and the keyboard or display must include a counter value and the name of the origin in addition to the encrypted key press or overlay itself.

VII. Web Enclave

In this section we cover the details of the web enclave. First, we provide an overview of the state transitions of a web enclave. Next, we present the protocols for remote attestation, origin authentication, and exchange of key material. Finally, we present the details of the operations: secure HTML forms, JavaScript code execution, secure network communication, and persistent storage across web enclave executions.

A. Web Enclave State Machine

The web enclave implements the state machine in Figure 3. At any point, it can be in one of the following five states: *initial*, *authenticated*, *ready*, *end*, and *fail*. Transitions are caused by ECALLs. Each state has a list of accepted ECALLs. For example, the initial state accepts only ECALLs for the remote attestation and origin validation. Other ECALLs bring the web enclave to the fail state. No other transition is possible from this state, and the enclave needs to be terminated after reaching it.

Fidelius creates a web enclave when it finds any `<form>` or `<script>` tags with the `secure` attribute set. Then it derives the origin of the tags that need to be protected. By default, the origin of the tags are inherited from the web page

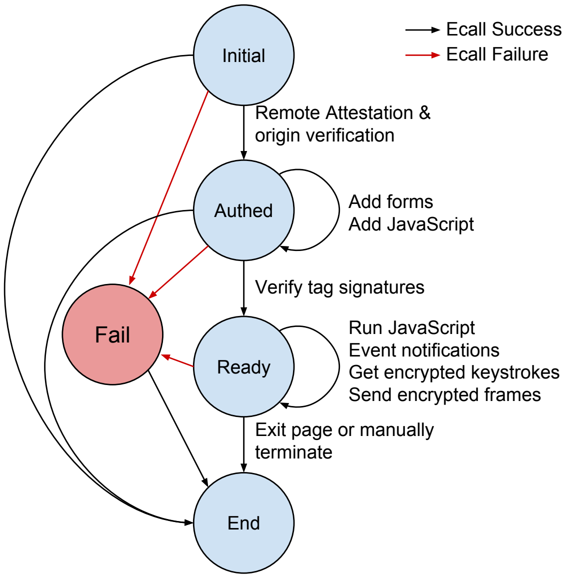

Fig. 3. Finite state machine representing web enclave behavior.

they belong to, i.e., the domain and port of the URL. However, for tags such as `<form>` and `<script>`, the origin is derived from the `action` and `src` attributes respecively. Tags can have different origins. While it is possible to create one web enclave for each origin, the current version of the web enclave assumes that all protected components on a page communicate with the same origin.

Once the origin has been determined, Fidelius passes the origin to the web enclave and performs remote attestation and origin validation. At this point, the web enclave is in the *authenticated* state. Fidelius retrieves the tags with the `secure` attribute set and loads them into the enclave. These operations do not cause a state transition. The only ECALL that causes a valid transition from this state is verification of the signatures. If the validation of all signatures succeeds, the enclave enters the *ready* state. From this point on, the enclave is fully operational and can decrypt keyboard inputs, prepare encrypted outputs for the display, execute JavaScript functions, and take/release control of the trusted path upon focus and blur events respectively.

B. Features

Once an enclave has successfully entered the ready state, the full functionality of Fidelius becomes available to the web application. Fidelius supports secure HTML forms, JavaScript execution, secure network communication, and persistent protected storage.

1) Secure HTML Forms: When parsing a page, Fidelius finds `<form>` tags with the secure attribute and, after verifying the provided signature using the server's public key, creates a `form` data structure inside the enclave to keep track of the form and each of the inputs inside it. We currently store server public keys inside the enclave but could replace this with root certificates instead. When the user highlights an input inside a given form, the browser notifies the enclave. The enclave switches the keyboard from untrusted to trusted input mode (see Section VI for details), and subsequent user key presses modify the state of the highlighted input field. As mentioned in Section V, various defenses at the interface level protect against attacks that an attacker could mount by modifying untrusted content between the enclave and the user. By pushing these defensess into the UI, we allow ourselves to keep many components of the browser outside of the enclave and dramatically reduce Fidelius's TCB. For example, monitoring of mouse movements and placement of forms on the page can be managed outside the enclave, and tampering/dishonesty with these elements will be detected by a user who notices the inconsistency between what she sees on the screen and the content of the trusted overlay.

Submission of HTML forms involves encrypting the content of the form as one blob using the shared key negotiated during attestation and sending that to the server.

2) Javascript: We run a JavaScript interpreter inside the enclave but leave out heavy components like the event loop. When a trusted JavaScript function is called, the enclave provides the interpreter with function inputs and any other state that should be available to the code about to run.

Javascript running in the enclave can access the content of protected HTML forms via the the global variable `forms`. The forms variable contains a property for each form name. For example, with reference to the HTML code in Listing 1, the payment form can be accessed via `forms.payment` where `payment` is the value of the attribute `name` of the `<form>` tag. Developers can implement custom input validation procedures. For example, a very simple form of validation can be checking if the credit card field contains forbidden characters such as white spaces. The JavaScript function that verifies the presence of white spaces can be implemented as shown in Listing 2.

```
function cardNumberHasWhiteSpaces() {
  return /\s/g.test(forms.payment.card);
}
```

Listing 2. Simple form validation

3) Network Communication: In order for protection of user data on the local machine to translate into a useful web application, there must be a mechanism for transmitting data out from the enclave without tampering by the compromised browser or OS. We provide a basic mechanism for doing this by supporting HTML forms, but web applications in general need to send back data to the server programmatically in a variety of contexts, not just when a user submits a form. To support this need, we provide support for XMLHttpRequests (as shown in Listing 3) where requests are encrypted inside the enclave using the shared key from the attestation process before leaving the enclave.

```
function doPay(e) {
  // input form to JS associative array
  d = toDict(forms.payment);

  // validate payment data
  if (validate(d)) {
      return false;
  }

  // prepare raw messages
  json_str = JSON.stringify(d);

  // create SecureXMLHttpRequest
  var xhr = new SecureXMLHttpRequest();
  xhr.open("POST",
    "https://pay.site.com/submit_data",
    false); // only sync calls

  // use sec_json content type
  xhr.setRequestHeader('Content-Type',
    'application/sec_json; charset=UTF-8');

  // encrypt, sign, and send
  xhr.send(json_str);

  // seal data for possible future reuse
  storeCreditCardData(d);
}
```

Listing 3. XMLHTTPRequest example

The problem of defending against replay of messages over the network is not unique to the trusted hardware setting and must be handled separately by applications built on Fidelius.

4) Persistent Storage: Fidelius provides developers with a web storage abstraction similar to the standard web storage provided by unmodified web browsers. Secure web storage can be accessed via `localStorage`, as shown in Listing 4.

```
function storeCreditCardData(d){
  localStorage['holder'] = d.holder;
  localStorage['cc']     = d.card;
  localStorage['exp']    = d.expiry;
  localStorage['cvv']    = d.cvv;
}
```

Listing 4. Web storage

When the need for persistent storage arises, Fidelius encrypts the data to be stored using a *sealing key* and stores it on disk (it could equivalently use existing browser storage mechanisms to hold the encrypted data). The sealing key is a feature provided by SGX to an enclave in order to store persistent data across multiple runs of the enclave.

This approach raises two problems we must resolve. First, every instance of the same enclave shares the same sealing key, so we must ensure that different enclaves created by the same browser cannot read each others' secrets. We can prevent this problem by including the associated origin as additional authenticated data with the encrypted data to be stored. This way an enclave can find and restore data associated with the origin it connects to but, as a matter of policy, does not allow the user to access data associated with any other origin. The integrity guarantees of our trusted hardware platform ensure that our code will abide by this policy.

The second issue is that of rollback attacks. A malicious operating system could roll back or delete data that is stored to disk, so, for applications that rely on maintaining sensitive state, the enclave must have a way to determine whether it has the most up-to-date stored data. A generic solution to this problem, such as ROTE [31], would suffice, but ROTE requires a distributed setting which may not be available to a user browsing the web from home. We can solve this problem by enlisting the assistance of the server to ensure protection against rollbacks, especially in situations where an enclave is connected to a server that already keeps information about the user. The idea is to keep a *revision number*, one for each origin, that gets sent from the server to the enclave at the end of the attestation process and is incremented whenever changes are made to locally stored data. Since the attacker cannot change the number stored on the server or in the enclave during execution, we can detect whenever a rollback attack has been launched or stored data has been deleted by observing a mismatch between the number on the data reloaded by the enclave and the number sent by the server.

Our generic approach for storage of user secrets and network connections could easily be extended to include storage of cookies, resulting in a separate cookie store, accessible only to the enclave, that otherwise provides the same functionality available from cookies in unmodified browsers.

VIII. Security Analysis

In this section we give a clear enumeration of the different kinds of threats against which we expect Fidelius to defend and argue that Fidelius does indeed protect against these attacks. We first discuss attacks on the core features of Fidelius and then move on to attacks targeted specifically at the trusted I/O path and user interface.

A. Attacks on Core Features

Enclave omission attack. An attacker with full control of the sofware running on a system may manipulate the browser extension and enclave manager software to pretend to use an enclave when in fact it does not. This attack will, however, fail because of defenses built into our user interface via the keyboard and display dongles. Absent a connection to a real enclave, the trusted input lights on the keyboard and display will not light, alerting the user that entered data is unprotected.

Enclave misuse attack. A more subtle attack of this form uses the enclave for some tasks but fakes it for others. For example, to circumvent the defense above, trusted input from the user could use the real enclave functionality, but trusted output on the display could be spoofed without the enclave. As such, it is necessary for each I/O device to separately defend against fake use of an enclave. The defenses described for the previous attack suffice to protect against this attack as well, but both lights are needed.

An attacker could also use the genuine trusted I/O path but attempt to omit use of the enclave when running JavaScript inside the browser. This attacker could clearly not access persistent storage, trusted network communication, or user inputs because those features require keys only available inside the enclave. On the other hand, the JavaScript to be run inside the enclave is not encrypted, so an attacker could potentially also run it outside the enclave, so long as it does not make use of any other resources or features offered by Fidelius. At this point, however, the JavaScript becomes entirely benign because it cannot give the attacker running it any new information or convince the user or remote server of any falsehoods because the trusted paths to all private information or trusted parties are barred.

A last variant of this attack would omit certain ECALLs that perform necessary setup operations like initializing a form and its inputs before the user begins to enter data. Omission of these ECALLs would result in the system crashing but would not leak secrets in the process. As mentioned before, we cannot conceivably protect against a denial of service attack where the compromised OS refuses to allow any access to the system. We can only ensure that normal or abnormal use of the enclave does not leak user secrets.

Page tampering attack. Failing to omit an enclave entirely or even partially, the attacker can turn to modifying the inputs given to various ecalls. In particular, the names and structure of forms and their inputs or the JavaScript to be run inside the enclave could be modified. Mounting this attack, however, would require an adversary who can break the unforgeability property of the signatures used to sign secure `<form>` and `<script>` tags. Those tags are verified with an origin-specific public key (either hard-coded in the enclave or verified with a certificate) that lies out of reach of our attacker.

Since trusted JavaScript is the only way to access trusted user inputs from within the browser, the fact that we have separate scope for execution of trusted and untrusted JavaScript means that any attempt to directly access user secrets stored in protected inputs will necessarily be thwarted.

Redirection attack. This attack resembles a straightforward phishing attempt. Instead of tampering with the operation of Fidelius, a browser could navigate to a malicious website designed to look legitimate in an attempt to send user secrets to an untrusted server. Here again the persistent overlay added by our display dongle prevents an attack by displaying the origin to which the enclave has connected. The strict same-origin policy within the enclave means that the origin displayed in the trusted portion of the screen is the only possible destination for network connections originating withing the enclave. While an attacker could establish a connection with a server other than the declared origin, the data sent to that server will be encrypted with a key known only to the intended origin, rendering the data useless to others. As such, the only way for an attacker to have legitimate-looking text appear there is to send user data only to legitimate destinations.

Storage tampering attack. Although authenticated encryption with a sealing key tied to the enclave protects persistently stored data from tampering, an attacker can still delete or roll back the state of stored data. We detail our solution to protect against this attack in Section VII-B4, where we enlist the assistance of the server to keep an up-to-date revision number for the enclave's data out of reach of the attacker. Attacks where the browser connects to a malicious site whose trusted JavaScript tries to read or modify persistent storage for other sites are prevented by our policy of strict separation between stored data associated with different origins.

B. Attacks on Trusted I/O Path and UI

We now consider attacks against the trusted I/O path to the user. Direct reading of private key presses and display outputs is prevented by encryption of data between the enclave and keyboard/display dongles, but we also consider a number of more sophisticated attacks. Since the I/O path to the user closely relates to the user interface, we discuss attacks against both the protocols and the interface together. We discuss security considerations involved in the setup of trusted I/O devices in Section VI-A.

Mode switching attack. As discussed in Section VI, the decision to switch between trusted and untrusted modes ultimately lies with the untrusted browser because it decides when an input field receives focus or blurs or when to activate Fidelius in the first place. We defend against this type of tampering with the light on the dongles and the delay when switching from trusted to untrusted modes. These defenses protect against both a standard unauthorized exit from the enclave as well as a rapid switching attack that tries to capture some key presses by quickly switching between modes.

A more involved variant of the mode-switching attack is an enclave-switching attack. In this attack, the untrusted sytem rapidly switches between different *enclaves*, one connecting to a legitimate site and the other to a malicious site controlled by the attacker. The mode-switching delay also protects against this variant of the attack because the display always shows the origin associated with the enclave currently in use.

Replay attack. We defend against replay of trusted communications between the enclave and display by including a non-repeating count in every message that is always checked to make sure an old count does not repeat. An attacker could, however, eavesdrop on key presses destined for one enclave, switch to a second enclave connected with a site it controls, and replay the key presses to the second enclave in an attempt to read trusted key presses. We defend against this attack by including the name of the origin along with the count in encrypted messages, so they cannot be replayed across different enclaves. Likewise, since the keyboard and display use different keys to encrypt communications with the enclave(s), messages cannot be replayed across sources.

Input manipulation attack. Attackers can attempt to make untrusted input fields appear where a user might expect trusted input fields and thereby fool users into typing trusted information in untrusted fields. Since the attacker has almost full control of what gets placed on the display, this grants considerable freedom in manipulating the display to mimic visual queues that would indicate secure fields. Fortunately, our display dongle reserves a strip at the bottom of the screen for trusted content directly from the enclave. This area informs the user what trusted input is currently focused, if any.

An attacker could also manipulate the placement of actual trusted input fields or the labels that precede them on a page in order to confuse or mislead a user as to the purpose of each field. By using the trusted display area to show *which* trusted input currently has focus, if any, we give developers the opportunity to assign inputs descriptive trusted names that will alert a user if there is a mismatch between an input's name and its stated purpose in the untrusted section of the display.

Timing attack. The fact that key presses originate with the user means that the timing of presses and associated updates to content on the screen may leak information about user secrets [28]. We close this timing side channel by having the keyboard send encrypted messages to the enclave at a constant rate while in trusted mode, sending null messages if the user does not press a key during a given time period and queueing key presses that appear in the same time period. A high enough frequency for this process ensures that the user experience is not disrupted by a backlog of key presses. Updates to display overlay contents also happen at a constant rate, so timing channels through key presses and display updates cannot leak information about user secrets.

IX. Implementation

We implemented a prototype of Fidelius, including both the trusted path described in Sections V and VI and the Web Enclave features described in Section VII[2]. Our prototype is fully functional but does not include the trusted setup stage between the enclave and devices, which we carry out manually. Figure 4 gives an overview of our prototype.

Since Fidelius requires few changes on the server side and our evaluation therefore focuses on browser overhead, we do not implement a server modified to run Fidelius. This would consist mainly of having the server verify a remote attestation and decrypt messages from the web enclave.

A. Trusted Path

Our prototype runs on an Intel Nuc with a 2.90 GHz Core i5-6260U Processor and 32 GB of RAM running Ubuntu 16.04.1 and SGX SDK version 2.1.2. We produced dongles to place between the Nuc and an off-the-shelf keyboard and display using a Raspberry Pi Zero with a 1 GHz single core Broadcom BCM2835 processor and 512 MB of RAM running Raspbian GNU/Linux 9 (stretch) for the keyboard and a Raspberry Pi 3 with a 1.2 GHZ quad-core ARM Cortex A53 processor and 1GB RAM running Raspbian GNU/Linux 9 (stretch) at a display resolution of 1280x720. Figures 5 and 6 show our input and output dongle devices.

The Raspberry Pi Zero simulated two input devices to the Nuc, one standard keyboard and one secure keyboard, with only one device active at any time based on the state of the application being run. The RPI 3 uses a B101 rev. 4 HMDI to CSI-2 bridge and the Picamera Python library [32] to treat the HDMI output from the Nuc as a camera input on which it overlays trusted content before rendering to the real display. Trusted content is sent over a separate bluetooth channel. The bluetooth channel exists as a matter of convenience for implementation, as HDMI does allow for sending auxiliary data, but we were unable to programmatically access this channel through existing drivers.

When an encrypted overlay packet reaches the RPI3 display device from the Nuc, it is first decrypted and decoded from a flat black and white encoding used to transfer data back to a full RBG color representation. Next, the image is transferred from the decryption/decoding program to the rendering code, which places it on the screen. We introduce a refresh delay between sending frames to give the Picamera library adequate time to render each frame before receiving the next one.

Although we have built a working Fidelius prototype, a number of improvements could make for a more powerful and complete product. These changes include miniaturization of dongle hardware, faster transfer protocols, e.g. USB 3.0 instead of Bluetooth, and custom drivers to reduce latency between the dongles and the keyboard/display. We leave the engineering task of optimizing Fidelius to future work.

B. Browser and Web Enclave

On the Intel Nuc device, Fidelius is implemented as a Chrome browser extension running on Chrome version 67.0.3396 communicating with a native program via Chrome's Native Messaging API[3] for web enclave management. The extension activates on page load and checks whether the page contains components that need to be protected, e.g., secure HTML forms and JavaScript. If it does, it communicates with the native program to initiate the web enclave and perform remote attestation with the server. Once this process completes, the user can interact with secure components on the page, and secure JavaScript code can be run in the enclave. Since the page setup process occurs independently of the page loading in the browser, only the secure components of a page are delayed by the attestation process – non-secure elements of a page have no loading penalty as a result of running Fidelius.

The majority of the work of enclave management is handled by the native code. For symmetric encryption of forms, bitmaps, and keystrokes we use AES-GCM encryption and for signing forms we use ECDSA signatures. JavaScript inside the enclave is run on a version of the `tiny-js` [33] interpreter that we ported to run inside the enclave.

X. Evaluation

We evaluate Fidelius in order to determine whether the overheads introduced by the trusted I/O path and web enclave are acceptable for common use cases and find that Fidelius outperforms display latency on some recent commercial devices by as much as 2.8× and prior work by 13.3×. Moreover, communication between the browser and enclave introduces a delay of less than 40ms to page load time for a login page. We also identify which components of the system contribute the most overhead, how they could be improved for a production deployment, and how performance scales for larger and more complex trusted page components.

[2] Our open source implementation of Fidelius, the instructions to build the dongles and accompanying sample code are available at https://github.com/SabaEskandarian/Fidelius.

[3] See https://developer.chrome.com/apps/nativeMessaging

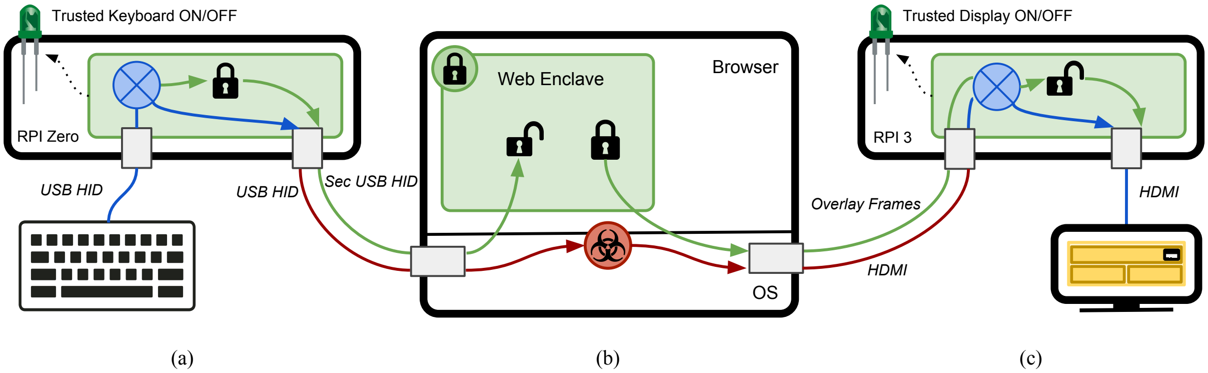

Fig. 4. Prototype of the trusted path: (a) standard USB keyboard connected to our RPI Zero dongle to encrypt keystrokes, (b) Computer with a Fidelius-enabled browser, and (c) standard HDMI display connected to our RPI 3 dongle to overlay secure frames.

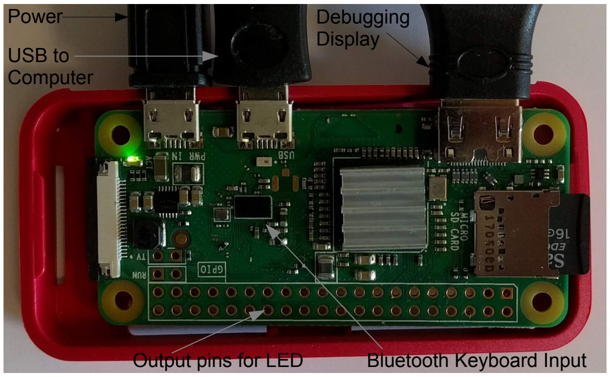

Fig. 5. Trusted keyboard dongle built from Raspberry Pi Zero. In untrusted mode, the dongle forwards key presses from the keyboard to the computer. In trusted mode, the dongle sends a constant stream of encrypted values to the enclave. The values correspond to key presses if there has been any input or null values otherwise.

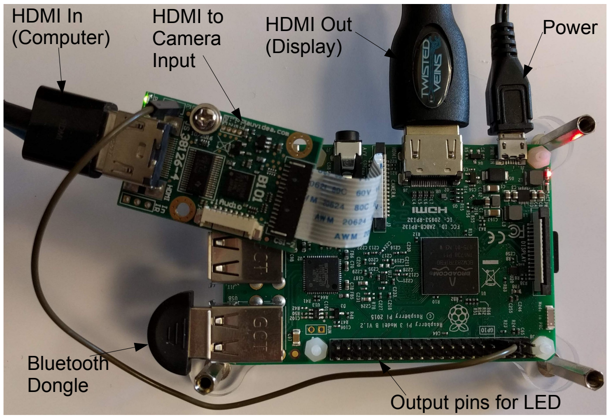

Fig. 6. Trusted display dongle built from Raspberry Pi 3. Frames arrive on the RPI3 over HDMI in, which connects through a board that treats the frames to be displayed as camera inputs. Overlays are transmitted over Bluetooth and decrypted on the RPI3. The combined frame and overlay go to the display through the HDMI out cable.

1) TCB Size: The trusted code base for Fidelius consists of 8,450 lines of C++ code, of which about 3200 are libraries for handling form rendering and another 3800 are our enclave port of `tiny-js`. This does not include native code running outside the enclave or in the browser extension because our security guarantees hold even if an attacker could compromise those untrusted components of the system. It also excludes dongle code which runs on the Raspberry Pi devices and not the computer running the web browser. Compared to the 18,800,000 lines of the Chrome project[4], Fidelius supports many of the important functionalities one may wish to secure in a web browser while exposing an attack surface orders of magnitude smaller than a naive port of a browser into a trusted execution environment.

2) Comparison to Commercial Devices: For a standard login form with username and password fields, Fidelius's key press to display latency is 201.8 ms. We exclude the time it takes to transfer the encrypted key press from the keyboard to the enclave over USB 2.0 (480 Mbps) and the encrypted bitmap from the enclave to the display over bluetooth (3 Mbps) from these figures. This is a reasonable omission because the size of the data being transferred is small compared to the transfer speed of these two protocols. Figure 7 compares the latency between a key press and display update in Fidelius to measurements of the display latency on several commercial mobile devices [34]. Although not competitive with high-performance devices, Fidelius performs comparably or even faster than some popular commercial devices, running 2.8× faster than the latency on the most recent Kindle. Fidelius's efficiency arises from leaving the majority of a page unmodified and only using encrypted overlays for trusted components.

3) Comparison to Prior Work: We also compared Fidelius to Bumpy [29], which provides a trusted input functionality but no corresponding display. For this comparison, we compared Bumpy to Fidelius's trusted path without the display component, which accounts for the vast majority of the latency. Bumpy's source code is not available, so we compare to the reported performance values measured on an HP dc5750 with

[4] https://www.openhub.net/p/chrome/analyses/latest/languages_summary

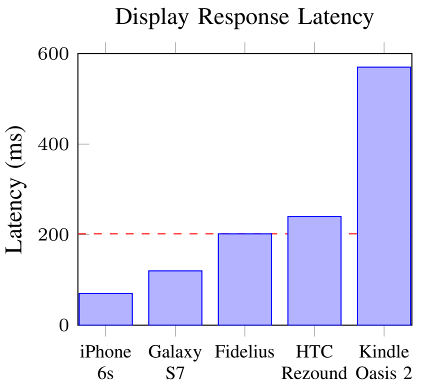

Fig. 7. Fidelius key press to display latency compared with the screen response time on various commercial devices.

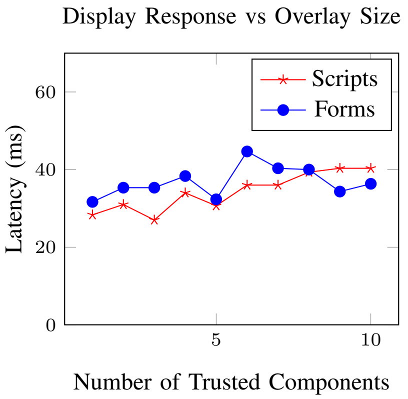

Fig. 8. Fidelius's impact on page load time as the number of trusted components varies. Adding components does not significantly affect load time.

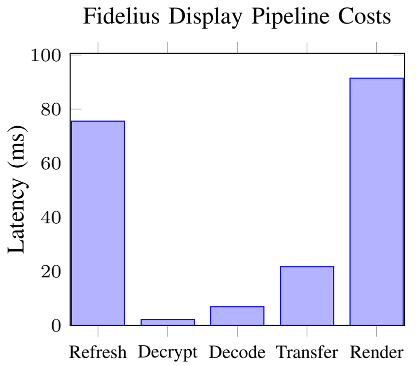

Fig. 9. Breakdown of display costs by component. Render/refresh delays are an artifact of our hardware and could be dramatically reduced.

Field size(s)	W	H	W×H px	Time (ms)	Incr. (ms)
1 Small	171	50	8,550	195.83	-
1 Medium	342	50	17,100	199.20	3.38
1 Large	683	50	34,150	209.65	10.45
1 Extra large	911	50	45,550	214.74	-
2 Extra large	911	100	91,100	227.02	12.28

Fig. 10. Key press to display latency when rendering forms. Widths are fractions of the most popular screen width ($w = 1366$px): S=$\frac{1}{8}w$, M=$\frac{1}{4}w$, L=$\frac{1}{2}w$, XL=$\frac{2}{3}w$. Increments calculated from the previous row.

an AMD Athlon64 X2 Processor at 2.2 GHz and a Broadcom v1.2 TPM. Fidelius outperforms Bumpy's reported performance by 13×, running with a latency of 10.59ms compared to Bumpy's 141ms. We believe this more than compensates for differences in the computing power used to evaluate the two systems. Although SGX-USB [30], whose source code is also unavailable, was developed on more recent hardware, we cannot compare directly to their reported performance results because they report generic USB data transfer rates into an enclave whereas we care about the latency of reading and processing key presses.

4) Page Load Overhead: Figure 8 shows the page load overhead incurred by Fidelius, not including remote attestation. Fidelius's overhead includes the time for the browser to inform the enclave of secure components and for the enclave to verify signatures on them, totaling 35.3ms. We do not report time for remote attestation, which depends on the latency to the attestation service. Fortunately, waiting for the attestation server to respond can occur in parallel with other page load operations because notifying the enclave of the existence of trusted components and verifying signatures do not involve sensitive user data. Moreover, attestation time is independent of page content, so our measurements fully capture Fidelius's page load time increase as trusted components are added. As seen in Figure 8, adding components does not significantly increase page load time.

5) Performance Factors: Figure 9 shows the cost of various components of our trusted display pipeline, described in Section IX-A, which makes up almost all of Fidelius's performance overhead. The two most expensive operations that take place on the display are rendering the overlay using the Picamera module and the refresh delay we introduce in order to allow the Picamera module to process frames without forming a queue of undisplayed frames. The Picamera module and associated hardware on the Raspberry Pi 3 is not optimized to add a dynamic overlay to the camera feed. A better approach would involve directly manipulating the data from the Nuc computer's HDMI output instead of using it to simulate a camera and placing overlays on top of the camera feed. This could easily be achieved in a production deployment of Fidelius and would dramatically reduce display latency.

We also considered how performance varies as the size of the trusted components on a page increase. Figure 10 shows that latency increases linearly with the size of the trusted component. This happens because as the size of the overlay increases, it takes longer to decrypt, decode and transfer the overlays. Taking steps to optimize the display pipeline would further mitigate latency increase. However, even under our current implementation, for two full-page width input fields (See the two extra large input field experiments in Figure 10), Fidelius has a display latency of only 227ms. Also, a tenfold increase in pixels (from one small field to two extra large fields) results in only a 31ms latency increase.

XI. Discussion and Extensions

Fidelius opens the door to a new class of secure web applications supported by the widespread availability of hardware enclaves in modern computers. The fundamental problems solved by Fidelius – reliably establishing a path from I/O devices to an enclave residing in an otherwise untrusted system and of protecting web applications without moving large portions of a browser into an enclave – have applications well beyond the login and payment examples described thus far.

Fidelius's techniques and architecture can also support more complex applications such as online tax filing or even web-based instant messaging. The trusted I/O path has applications beyond the web as well and could be adapted to secure logins or desktop applications that use enclaves for their core functionality but require interaction with a local user on the machine. We anticipate that Fidelius's I/O approach will be very useful, as hardware enclaves are most widely available on consumer desktop and laptop computers.

We close with a discussion of possible extensions that could broaden the applicability of our architecture or would be important considerations in a widespread deployment.

1) Usability of Trusted Devices: We have implemented Fidelius with a user and developer interface that provides users with the necessary tools to interpret their interaction with Fidelius properly and avoid UI-based attacks. However, our interface represents only one of many possible designs for interaction between users and the core Fidelius functionality. A great deal of work has studied the effectiveness of security indicators such as our indicator lights [25], [26]. Other possible designs may, for example, use secure attention sequences or separate trusted buttons to initiate communication with trusted components. Future work could explore this space to determine what approach works best for this application in practice.

2) Event Loop: Fidelius leaves the JavaScript event loop outside the enclave to optimize the tradeoff between TCB size and functionality. A number of additional applications could be enabled by moving the event loop into an enclave, especially if there is a way to accomplish this more efficiently than with a direct port that executes the loop as-is in trusted hardware.

3) HTML Rendering: In order to render HTML forms, we implemented a custom library that, given a description of a form and its inputs, produces a bitmap that represents the form. In order to extend support to other HTML tags, we need to integrate a more versatile rendering engine into our web enclaves. Existing libraries such as Nuklear [35] provide a solid first step in this direction.

4) Root Certificate Store: Our current implementation of the web enclave uses a limited number of public keys. To scale to supporting any web site, the web enclave needs to have a root certificate store inside the enclave.

XII. Related Work

NGSCB. In 2003 Microsoft announces the Palladium effort, later renamed NGSCB [19]. In that design, attestation is provided by a TPM chip and enclave isolation is provided by hardware memory curtaining. The project was scaled back in 2005 presumably due to the difficulty of adapting applications to the architecture. In contrast, as we explained, web sites can take advantage of Fidelius by simply adding an HTML attribute to web fields and forms that it wants to protect.

SGX and the Web. TrustJS [17] explores the potential for running JavaScript inside an enclave, demonstrating that running trusted JavaScript on the client-side can expedite form input validation. SecureWorker [18] provides the developer abstraction of a web worker while executing the worker's JavaScript inside an enclave. Our work uses the ability to run JavaScript in an enclave as a building block to enable privacy for user inputs in web applications. JITGuard [36] uses SGX to protect against vulnerabilities in Firefox's JIT compiler.

Unmodified Applications on SGX. A handful of works aim to allow execution of unmodified applications inside an SGX enclave. Haven runs whole applications inside an enclave [1], while SGXKernel [3], Graphene [5], and Panoply [2] provide lower level primitives on which applications can be built. Scone [4] secures linux containers by running them inside an enclave. Flicker [37] and TrustVisor [38] use older hardware to provide features similar to SGX on which general applications can be built, albeit with weaker performance due to the older and more limited hardware features on which they build. We focus on directly solving the problem of hiding user inputs in an untrusted browser without using generic solutions in order to minimize TCB and avoid the potential pitfalls of porting a monolithic browser into a trusted environment.

SGX Attacks and Defenses. A number of side channel attacks on SGX have been shown to take advantage of, among other things, memory access patterns [39]–[41], asynchronous execution [42], branch prediction [43], speculative execution [20], [21], and even SGX's own security guarantees [44] to compromise data privacy. There do, however, exist many defenses that have been shown to evade these side channels, often generically, without a great deal of overhead [31], [45]–[49]. Even more promising, researchers have proposed a series of other architectures [22], [50], [51] which defend against weaknesses in SGX *by design* and are therefore invulnerable to broad classes of attacks. As our work is compatible with generic defenses and concerns itself primarily with higher level functionalities built over enclaves, we do not consider side channels in the presentation of Fidelius.

Protection Against Compromised Browsers. A number of software-based solutions for protection against compromised browsers offer tradeoffs between security, performance, and TCB size. Shadowcrypt [52] uses a Shadow DOM to allow encrypted input/output for web applications, but is vulnerable to some attacks [53]. Terra [54] uses VMs to allow applications with differing security requirements to run together on the same hardware. Tahoma [55], IBOS [56], and Proxos [57] integrate support for browsers as OS-level features, allowing smaller TCBs and stronger isolation/security guarantees than in a general-purpose OS. Cloud terminal [58] evades the problem of local malware and protects against attackers by only running a lightweight *secure thin terminal* locally and outsourcing the majority of computation to a remote server.

Trusted I/O Path. While many works study how to leverage a trusted hypervisor to achieve a trusted path to users (e.g. [59]–[63]), little work has been done in the trusted hardware setting. SGXIO [16] provides a hybrid solution that combines SGX with traditional hypervisor techniques to allow a trusted I/O path with unmodified devices. In contrast, our work relies *only* on hardware assumptions with no need for a hypervisor, but does require modified keyboard and display devices. Intel has alluded to an internal tool used to provide a trusted display from SGX [64], [65], but no details, source code, or applications are available for public use. SGX-USB [30] allows for generic I/O but does not solve the problem of mixing trusted and untrusted content in a user interface as we do in both our keyboard and display.

Bumpy [29] (and to a lesser extent its predecessor BitE [66]) use the trusted execution environment provided by Flicker [37] to provide a secure input functionality very similar to ours. Aside from the larger web architecture which we build over our trusted I/O features, we go beyond these works by 1) enabling interactivity with the trusted input via the trusted display (Bumpy does not display characters the user types) and 2) closing timing side channels on user input (an improvement we also offer over SGX-USB).

XIII. Conclusion

We have presented Fidelius, a new architecture for protecting user secrets from malicious operating systems while interacting with web applications. Fidelius protects form inputs, JavaScript execution, network connections, and local storage from malware in a fully compromised browser. In addition, Fidelius features the first publicly available system for a trusted I/O path between a user and a hardware enclave without making additional assumptions about hypervisor security. Our open source implementation of Fidelius, accompanying sample code, and a video demo are available at https://github.com/SabaEskandarian/Fidelius.

Acknowledgment

We thank Amit Sahai and Keith Winstein for helpful conversations about this work. This work was supported by NSF, DARPA, a grant from ONR, the Simons Foundation, a Google faculty fellowship, and the German Federal Ministry of Education and Research (BMBF) through funding for the CISPA-Stanford Center for Cybersecurity (FKZ: 13N1S0762).

References

[1] A. Baumann, M. Peinado, and G. C. Hunt, “Shielding applications from an untrusted cloud with haven,” in *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI ’14, Broomfield, CO, USA, October 6-8, 2014.*, 2014, pp. 267–283. [Online]. Available: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/baumann

[2] S. Shinde, D. L. Tien, S. Tople, and P. Saxena, “Panoply: Low-tcb linux applications with SGX enclaves,” in *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*, 2017. [Online]. Available: https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/panoply-low-tcb-linux-applications-sgx-enclaves/

[3] H. Tian, Y. Zhang, C. Xing, and S. Yan, “Sgxkernel: A library operating system optimized for intel SGX,” in *Proceedings of the Computing Frontiers Conference, CF’17, Siena, Italy, May 15-17, 2017*, 2017, pp. 35–44. [Online]. Available: http://doi.acm.org/10.1145/3075564.3075572

[4] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O’Keeffe, M. Stillwell, D. Goltzsche, D. M. Eyers, R. Kapitza, P. R. Pietzuch, and C. Fetzer, “SCONE: secure linux containers with intel SGX,” in *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016.*, 2016, pp. 689–703. [Online]. Available: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/arnautov

[5] C. Tsai, D. E. Porter, and M. Vij, “Graphene-sgx: A practical library OS for unmodified applications on SGX,” in *2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017.*, 2017, pp. 645–658. [Online]. Available: https://www.usenix.org/conference/atc17/technical-sessions/presentation/tsai

[6] A. Papadimitriou, R. Bhagwan, N. Chandran, R. Ramjee, A. Haeberlen, H. Singh, A. Modi, and S. Badrinarayanan, “Big data analytics over encrypted datasets with seabed,” in *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016.*, 2016, pp. 587–602. [Online]. Available: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/papadimitriou

[7] S. Brenner, C. Wulf, D. Goltzsche, N. Weichbrodt, M. Lorenz, C. Fetzer, P. R. Pietzuch, and R. Kapitza, “Securekeeper: Confidential zookeeper using intel SGX,” in *Proceedings of the 17th International Middleware Conference, Trento, Italy, December 12 - 16, 2016*, 2016, p. 14. [Online]. Available: http://dl.acm.org/citation.cfm?id=2988350

[8] B. A. Fisch, D. Vinayagamurthy, D. Boneh, and S. Gorbunov, “Iron: Functional encryption using intel sgx,” *IACR Cryptology ePrint Archive*, 2016. [Online]. Available: http://eprint.iacr.org/2016/1071

[9] A. Bittau, Ú. Erlingsson, P. Maniatis, I. Mironov, A. Raghunathan, D. Lie, M. Rudominer, U. Kode, J. Tinnés, and B. Seefeld, “Prochlo: Strong privacy for analytics in the crowd,” in *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*, 2017, pp. 441–459. [Online]. Available: http://doi.acm.org/10.1145/3132747.3132769

[10] C. Priebe, K. Vaswani, and M. Costa, “Enclavedb: A secure database using sgx,” in *2018 IEEE Symposium on Security and Privacy, SP (Oakland)*, 2018.

[11] S. Sasy, S. Gorbunov, and C. W. Fletcher, “Zerotrace : Oblivious memory primitives from intel SGX,” *IACR Cryptology ePrint Archive*, vol. 2017, p. 549, 2017. [Online]. Available: http://eprint.iacr.org/2017/549

[12] T. Hunt, Z. Zhu, Y. Xu, S. Peter, and E. Witchel, “Ryoan: A distributed sandbox for untrusted computation on secret data,” in *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016.*, 2016, pp. 533–549. [Online]. Available: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/hunt

[13] M. Russinovich, “Introducing Azure confidential computing,” 2017, https://azure.microsoft.com/en-us/blog/introducing-azure-confidential-computing/.

[14] W. Zheng, A. Dave, J. G. Beekman, R. A. Popa, J. E. Gonzalez, and I. Stoica, “Opaque: An oblivious and encrypted distributed analytics platform,” in *14th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2017, Boston, MA, USA, March 27-29, 2017*, 2017, pp. 283–298. [Online]. Available: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/zheng

[15] P. Mishra, R. Poddar, J. Chen, A. Chiesa, and R. A. Popa, “Oblix: An efficient oblivious search index,” in *2018 IEEE Symposium on Security and Privacy, SP (Oakland)*, 2018.

[16] S. Weiser and M. Werner, “SGXIO: generic trusted I/O path for intel SGX,” in *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy, CODASPY 2017, Scottsdale, AZ, USA, March 22-24, 2017*, G. Ahn, A. Pretschner, and G. Ghinita, Eds. ACM, 2017, pp. 261–268. [Online]. Available: http://doi.acm.org/10.1145/3029806.3029822

[17] D. Goltzsche, C. Wulf, D. Muthukumaran, K. Rieck, P. R. Pietzuch, and R. Kapitza, “Trustjs: Trusted client-side execution of javascript,” in *Proceedings of the 10th European Workshop on Systems Security, EUROSEC 2017, Belgrade, Serbia, April 23, 2017*, C. Giuffrida and A. Stavrou, Eds. ACM, 2017, pp. 7:1–7:6. [Online]. Available: http://doi.acm.org/10.1145/3065913.3065917

[18] mitar, wh0, and C. V. Wiemeersch, “Secureworker,” https://github.com/luckychain/node-secureworker, 2018.

[19] Wikipedia contributors, “Next-generation secure computing base — Wikipedia, the free encyclopedia,” 2018, accessed August 2018. [Online]. Available: https://en.wikipedia.org/wiki/Next-Generation_Secure_Computing_Base

[20] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai, “Sgxpectre attacks: Leaking enclave secrets via speculative execution,” *CoRR*, vol. abs/1802.09085, 2018. [Online]. Available: http://arxiv.org/abs/1802.09085

[21] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, “Foreshadow:

Extracting the keys to the Intel SGX kingdom with transient out-of-order execution," in *Proceedings of the 27th USENIX Security Symposium*. USENIX Association, August 2018.

[22] V. Costan, I. A. Lebedev, and S. Devadas, "Sanctum: Minimal hardware extensions for strong software isolation," in *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016.*, 2016, pp. 857–874. [Online]. Available: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/costan

[23] V. Costan and S. Devadas, "Intel SGX explained," *IACR Cryptology ePrint Archive*, vol. 2016, p. 86, 2016. [Online]. Available: http://eprint.iacr.org/2016/086

[24] "Intel software guard extensions sdk for linux os, developer reference." [Online]. Available: https://download.01.org/intel-sgx/linux-1.8/docs/Intel_SGX_SDK_Developer_Reference_Linux_1.8_Open_Source.pdf

[25] T. Whalen and K. M. Inkpen, "Gathering evidence: use of visual security cues in web browsers," in *Proceedings of the Graphics Interface 2005 Conference, May 9-11, 2005, Victoria, British Columbia, Canada*, 2005, pp. 137–144.

[26] S. E. Schechter, R. Dhamija, A. Ozment, and I. Fischer, "The emperor's new security indicators," in *2007 IEEE Symposium on Security and Privacy (S&P 2007), 20-23 May 2007, Oakland, California, USA*, 2007, pp. 51–65.

[27] C. Jackson, D. R. Simon, D. S. Tan, and A. Barth, "An evaluation of extended validation and picture-in-picture phishing attacks," in *Financial Cryptography and Data Security, 11th International Conference, FC 2007, and 1st International Workshop on Usable Security, USEC 2007, Scarborough, Trinidad and Tobago, February 12-16, 2007. Revised Selected Papers*, ser. Lecture Notes in Computer Science, S. Dietrich and R. Dhamija, Eds., vol. 4886. Springer, 2007, pp. 281–293. [Online]. Available: https://doi.org/10.1007/978-3-540-77366-5_27

[28] D. X. Song, D. A. Wagner, and X. Tian, "Timing analysis of keystrokes and timing attacks on SSH," in *10th USENIX Security Symposium, August 13-17, 2001, Washington, D.C., USA*, D. S. Wallach, Ed. USENIX, 2001. [Online]. Available: http://www.usenix.org/publications/library/proceedings/sec01/song.html

[29] J. M. McCune, A. Perrig, and M. K. Reiter, "Safe passage for passwords and other sensitive data," in *Proceedings of the Network and Distributed System Security Symposium, NDSS 2009, San Diego, California, USA, 8th February - 11th February 2009*. The Internet Society, 2009. [Online]. Available: http://www.isoc.org/isoc/conferences/ndss/09/pdf/18.pdf

[30] Y. Jang, "Building trust in the user i/o in computer systems," Ph.D. dissertation, Georgia Tech, 2017.

[31] S. Matetic, M. Ahmed, K. Kostiainen, A. Dhar, D. Sommer, A. Gervais, A. Juels, and S. Capkun, "ROTE: rollback protection for trusted execution," in *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017.*, 2017, pp. 1289–1306. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/matetic

[32] D. Jones, "Picamera," https://github.com/waveform80/picamera, 2017.

[33] G. Williams, "tiny-js," https://github.com/gfwilliams/tiny-js, 2015.

[34] D. Luu, "Computer latency: 1977-2017," 2017, https://danluu.com/input-lag/.

[35] M. Mettke, "Nuklear," https://github.com/vurtun/nuklear, 2018.

[36] T. Frassetto, D. Gens, C. Liebchen, and A. Sadeghi, "Jitguard: Hardening just-in-time compilers with SGX," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, B. M. Thuraisingham, D. Evans, T. Malkin, and D. Xu, Eds. ACM, 2017, pp. 2405–2419. [Online]. Available: http://doi.acm.org/10.1145/3133956.3134037

[37] J. M. McCune, B. Parno, A. Perrig, M. K. Reiter, and H. Isozaki, "Flicker: an execution infrastructure for tcb minimization," in *Proceedings of the 2008 EuroSys Conference, Glasgow, Scotland, UK, April 1-4, 2008*, J. S. Sventek and S. Hand, Eds. ACM, 2008, pp. 315–328. [Online]. Available: http://doi.acm.org/10.1145/1352592.1352625

[38] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. D. Gligor, and A. Perrig, "Trustvisor: Efficient TCB reduction and attestation," in *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA*. IEEE Computer Society, 2010, pp. 143–158. [Online]. Available: https://doi.org/10.1109/SP.2010.17

[39] J. Götzfried, M. Eckert, S. Schinzel, and T. Müller, "Cache attacks on intel SGX," in *Proceedings of the 10th European Workshop on Systems Security, EUROSEC 2017, Belgrade, Serbia, April 23, 2017*, 2017, pp. 2:1–2:6. [Online]. Available: http://doi.acm.org/10.1145/3065913.3065915

[40] Y. Xu, W. Cui, and M. Peinado, "Controlled-channel attacks: Deterministic side channels for untrusted operating systems," in *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, 2015, pp. 640–656. [Online]. Available: https://doi.org/10.1109/SP.2015.45

[41] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A. Sadeghi, "Software grand exposure: SGX cache attacks are practical," in *11th USENIX Workshop on Offensive Technologies, WOOT 2017, Vancouver, BC, Canada, August 14-15, 2017.*, 2017. [Online]. Available: https://www.usenix.org/conference/woot17/workshop-program/presentation/brasser

[42] N. Weichbrodt, A. Kurmus, P. R. Pietzuch, and R. Kapitza, "Asyncshock: Exploiting synchronisation bugs in intel SGX enclaves," in *Computer Security - ESORICS 2016 - 21st European Symposium on Research in Computer Security, Heraklion, Greece, September 26-30, 2016, Proceedings, Part I*, 2016, pp. 440–457. [Online]. Available: https://doi.org/10.1007/978-3-319-45744-4_22

[43] S. Lee, M. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado, "Inferring fine-grained control flow inside SGX enclaves with branch shadowing," in *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017.*, 2017, pp. 557–574. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/lee-sangho

[44] M. Schwarz, S. Weiser, D. Gruss, C. Maurice, and S. Mangard, "Malware guard extension: Using SGX to conceal cache attacks," in *Detection of Intrusions and Malware, and Vulnerability Assessment - 14th International Conference, DIMVA 2017, Bonn, Germany, July 6-7, 2017, Proceedings*, 2017, pp. 3–24. [Online]. Available: https://doi.org/10.1007/978-3-319-60876-1_1

[45] A. Rane, C. Lin, and M. Tiwari, "Raccoon: Closing digital side-channels through obfuscated execution," in *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015.*, 2015, pp. 431–446. [Online]. Available: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/rane

[46] S. Shinde, Z. L. Chua, V. Narayanan, and P. Saxena, "Preventing page faults from telling your secrets," in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security, AsiaCCS 2016, Xi'an, China, May 30 - June 3, 2016*, 2016, pp. 317–328. [Online]. Available: http://doi.acm.org/10.1145/2897845.2897885

[47] J. Seo, B. Lee, S. M. Kim, M. Shih, I. Shin, D. Han, and T. Kim, "Sgx-shield: Enabling address space layout randomization for SGX programs," in *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*, 2017. [Online]. Available: https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/sgx-shield-enabling-address-space-layout-randomization-sgx-programs/

[48] M. Shih, S. Lee, T. Kim, and M. Peinado, "T-SGX: eradicating controlled-channel attacks against enclave programs," in *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*, 2017. [Online]. Available: https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/t-sgx-eradicating-controlled-channel-attacks-against-enclave-programs/

[49] R. Sinha, S. K. Rajamani, S. A. Seshia, and K. Vaswani, "Moat: Verifying confidentiality of enclave programs," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015*, 2015, pp. 1169–1184. [Online]. Available: http://doi.acm.org/10.1145/2810103.2813608

[50] C. Liu, A. Harris, M. Maas, M. W. Hicks, M. Tiwari, and E. Shi, "Ghostrider: A hardware-software system for memory trace oblivious computation," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '15, Istanbul, Turkey, March 14-18, 2015*, Ö. Özturk, K. Ebcioglu, and S. Dwarkadas, Eds. ACM, 2015, pp. 87–101. [Online]. Available: http://doi.acm.org/10.1145/2694344.2694385

[51] M. Maas, E. Love, E. Stefanov, M. Tiwari, E. Shi, K. Asanovic, J. Kubiatowicz, and D. Song, "PHANTOM: practical oblivious computation in a secure processor," in *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, A. Sadeghi, V. D. Gligor, and M. Yung, Eds. ACM, 2013, pp. 311–324. [Online]. Available: http://doi.acm.org/10.1145/2508859.2516692

[52] W. He, D. Akhawe, S. Jain, E. Shi, and D. X. Song, “Shadowcrypt: Encrypted web applications for everyone,” in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, 2014, pp. 1028–1039. [Online]. Available: http://doi.acm.org/10.1145/2660267.2660326

[53] M. Freyberger, W. He, D. Akhawe, M. L. Mazurek, and P. Mittal, “Cracking shadowcrypt: Exploring the limitations of secure I/O systems in internet browsers,” *PoPETs*, vol. 2018, no. 2, pp. 47–63, 2018. [Online]. Available: https://doi.org/10.1515/popets-2018-0012

[54] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh, “Terra: a virtual machine-based platform for trusted computing,” in *Proceedings of the 19th ACM Symposium on Operating Systems Principles 2003, SOSP 2003, Bolton Landing, NY, USA, October 19-22, 2003*, M. L. Scott and L. L. Peterson, Eds. ACM, 2003, pp. 193–206. [Online]. Available: http://doi.acm.org/10.1145/945445.945464

[55] R. S. Cox, S. D. Gribble, H. M. Levy, and J. G. Hansen, “A safety-oriented platform for web applications,” in *2006 IEEE Symposium on Security and Privacy (S&P 2006), 21-24 May 2006, Berkeley, California, USA*. IEEE Computer Society, 2006, pp. 350–364. [Online]. Available: https://doi.org/10.1109/SP.2006.4

[56] S. Tang, H. Mai, and S. T. King, “Trust and protection in the illinois browser operating system,” in *9th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2010, October 4-6, 2010, Vancouver, BC, Canada, Proceedings*, R. H. Arpaci-Dusseau and B. Chen, Eds. USENIX Association, 2010, pp. 17–32. [Online]. Available: http://www.usenix.org/events/osdi10/tech/full_papers/Tang.pdf

[57] R. Ta-Min, L. Litty, and D. Lie, “Splitting interfaces: Making trust between applications and operating systems configurable,” in *7th Symposium on Operating Systems Design and Implementation (OSDI ’06), November 6-8, Seattle, WA, USA*, B. N. Bershad and J. C. Mogul, Eds. USENIX Association, 2006, pp. 279–292. [Online]. Available: http://www.usenix.org/events/osdi06/tech/ta-min.html

[58] L. Martignoni, P. Poosankam, M. Zaharia, J. Han, S. McCamant, D. Song, V. Paxson, A. Perrig, S. Shenker, and I. Stoica, “Cloud terminal: Secure access to sensitive applications from untrusted systems,” in *2012 USENIX Annual Technical Conference, Boston, MA, USA, June 13-15, 2012*, 2012, pp. 165–182. [Online]. Available: https://www.usenix.org/conference/atc12/technical-sessions/presentation/martignoni

[65] R. Lal and P. M. Pappachan, “An architecture methodology for secure video conferencing,” in *Technologies for Homeland Security (HST)*. IEEE, 2013.

[59] A. Brandon and M. Trimarchi, “Trusted display and input using screen overlays,” in *International Conference on ReConFigurable Computing and FPGAs, ReConFig 2017, Cancun, Mexico, December 4-6, 2017*, 2017, pp. 1–6. [Online]. Available: https://doi.org/10.1109/RECONFIG.2017.8279826

[60] Z. Zhou, V. D. Gligor, J. Newsome, and J. M. McCune, “Building verifiable trusted path on commodity x86 computers,” in *IEEE Symposium on Security and Privacy, SP 2012, 21-23 May 2012, San Francisco, California, USA*, 2012, pp. 616–630. [Online]. Available: https://doi.org/10.1109/SP.2012.42

[61] M. Yu, V. D. Gligor, and Z. Zhou, “Trusted display on untrusted commodity platforms,” in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015*, 2015, pp. 989–1003. [Online]. Available: http://doi.acm.org/10.1145/2810103.2813719

[62] K. Borders and A. Prakash, “Securing network input via a trusted input proxy,” in *2nd USENIX Workshop on Hot Topics in Security, HotSec’07, Boston, MA, USA, August 7, 2007*. USENIX Association, 2007. [Online]. Available: https://www.usenix.org/conference/hotsec-07/securing-network-input-trusted-input-proxy

[63] N. Feske and C. Helmuth, “A nitpicker’s guide to a minimal-complexity secure GUI,” in *21st Annual Computer Security Applications Conference (ACSAC 2005), 5-9 December 2005, Tucson, AZ, USA*. IEEE Computer Society, 2005, pp. 85–94. [Online]. Available: https://doi.org/10.1109/CSAC.2005.7

[64] M. Hoekstra, R. Lal, P. Pappachan, V. Phegade, and J. del Cuvillo, “Using innovative instructions to create trustworthy software solutions,” in *HASP 2013, The Second Workshop on Hardware and Architectural Support for Security and Privacy, Tel-Aviv, Israel, June 23-24, 2013*, 2013, p. 11. [Online]. Available: http://doi.acm.org/10.1145/2487726.2488370

[66] J. M. McCune, A. Perrig, and M. K. Reiter, “Bump in the ether: A framework for securing sensitive user input,” in *Proceedings of the 2006 USENIX Annual Technical Conference, Boston, MA, USA, May 30 - June 3, 2006*, A. Adya and E. M. Nahum, Eds. USENIX, 2006, pp. 185–198. [Online]. Available: http://www.usenix.org/events/usenix06/tech/mccune.html