Protecting Sensitive Web Content from Client-side Vulnerabilities with CRYPTONs

Xinshu Dong
Dept. of Computer Science,
National Univ. of Singapore
xdong@comp.nus.edu.sg

Zhaofeng Chen*
Inst. of Computer Science and
Technology, Peking Univ.
chenzhaofeng@pku.edu.cn

Hossein Siadati*
Dept. of Computer Science
and Engineering, Polytechnic
Inst. of New York Univ.
shs422@nyu.edu

Shruti Tople
Dept. of Computer Science,
National Univ. of Singapore
shruti90@comp.nus.edu.sg

Prateek Saxena
Dept. of Computer Science,
National Univ. of Singapore
prateeks@comp.nus.edu.sg

Zhenkai Liang
Dept. of Computer Science,
National Univ. of Singapore
liangzk@comp.nus.edu.sg

ABSTRACT

Web browsers isolate web origins, but do not provide direct abstractions to isolate sensitive data and control computation over it within the same origin. As a result, guaranteeing security of sensitive web content requires trusting all code in the browser and client-side applications to be vulnerability-free. In this paper, we propose a new abstraction, called CRYPTON, which supports *intra-origin* control over sensitive data throughout its life cycle. To securely enforce the semantics of CRYPTONs, we develop a standalone component called CRYPTON-KERNEL, which extensively leverages the functionality of existing web browsers without relying on their large TCB. Our evaluation demonstrates that the CRYPTON abstraction supported by the CRYPTON-KERNEL is widely applicable to popular real-world applications with millions of users, including webmail, chat, blog applications, and Alexa Top 50 websites, with low performance overhead.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection;
K.6.5 [**Management of Computing and Information Systems**]: Security and Protection

Keywords

Web security, browser security, data protection

1. INTRODUCTION

Presently, web browsers are designed to isolate content between web origins [38], preventing data owned by one origin from being accessed by other origins. However, in practice, information processed within a web origin's protection domain often has different levels of sensitivity. Some application data — which we refer to as *sensitive data* in this paper — is critically sensitive and more important than the rest of application data. Social security numbers, medical reports, enterprise emails with financial data, tax-related information, bank transactions, and user passwords are all examples of such sensitive data. Web application owners often wish to strongly isolate sensitive data within their applications, to protect it from vulnerabilities in the code that processes non-sensitive information.

Under the origin-based model, web applications processing sensitive data, such as password managers [43] and encrypted chat or email clients [41], have to completely trust the client-side browser's origin-based isolation mechanisms and the correctness of the client-side application code [2, 67]. Information belonging to one origin is accessible to all the application code running within the protection domain of that origin, as well as the browser code (including add-ons), even if they do not need to access such information. The client-side web application and the underlying browsers consist of millions of lines of code and have had their steady share of security vulnerabilities historically; client-side application vulnerabilities (like DOM-XSS [62]) are pervasive in client-side application code and browser add-ons [14]. Given the scope of client-side vulnerabilities, protecting sensitive data within the browser environment is a serious practical concern.

Problem. The crux of the problem is that the web lacks abstractions for information owners to specify what information is sensitive and how it must be processed in the client-side web browser environment. To address this problem, we envision new abstractions that give owners control of select sensitive data over its lifetime in the client-side browser. We term these as *intra-origin data control* abstractions. In particular, we advocate that web browsers provide these controls as *first-class* abstractions on the web, independently of strengthening the browser's origin-based isolation.

Although there has been extensive prior work on browser design, principled mechanisms for intra-origin data control have not received much direct attention. For instance, recent browser designs have significantly improved privilege separation between code components [4, 5, 16, 21, 25, 33, 46, 63, 69]. However, these techniques do not directly allow web applications to enforce control on their data; a compromised component with access to sensitive data can still launder it through its export interfaces. For example, the sandboxed renderer process in Google Chrome [5], if compromised, can

*Research done when visiting National University of Singapore.

CCS'13, November 4–8, 2013, Berlin, Germany.
Copyright 2013 ACM 978-1-4503-2477-9/13/11 ...$15.00.
http://dx.doi.org/10.1145/2508859.2516743.

still exploit legitimate network interfaces provided by the Chrome kernel to leak data to attacker-controlled domains (say via an `img` load) [17]. Although information control has been investigated through language-based enforcements [22, 34, 50, 67], these mechanisms fail to extend the protection to the lowest level (e.g. against attacks targeting low-level memory access). Piecemeal techniques (such as SFI [68], JIT hardening [59], and CSP [67]) could be a plausible basis for sensitive data protection. However, we argue to build more direct information control primitives on the present web.

Our solution. We design and implement a data abstraction and browser primitive, called CRYPTON, which enables protecting sensitive data throughout its lifetime in the browser. CRYPTONs are programmed directly in existing web languages (HTML/JavaScript), without requiring major re-engineering of existing application logic. A CRYPTON explicitly marks a unit of sensitive data in a web page; its semantics enforce that sensitive data and information computed from them are tightly isolated at the lowest level (in the raw program memory). To support rich web applications that need to execute flexible operations on sensitive data, CRYPTONs tie sensitive data with functions that are allowed to compute over it. These few select CRYPTON functions are trusted to be statically verified by web developers and are the only channels for releasing information about sensitive web content. In addition, a CRYPTON provides other important capabilities for protecting sensitive data in web applications — guaranteed rendering (with a proof-of-impression), and certified delivery of user inputs to CRYPTON functions in the browser.

Building trustworthy implementations of these abstractions is an important, but challenging goal. The main challenge stems from the *monolithic* trust model of today's web — application servers have to completely trust the client-side code in web browsers, add-ons and applications.

To address this practical concern, we rethink the monolithic trust model that web applications have on client-side code. In our design, the web server trusts a small piece of client-side code called CRYPTON-KERNEL— a small, trusted standalone engine, which acts as a root-of-trust for the server on the client device. The CRYPTON-KERNEL runs in a separate OS process and sandboxes the untrusted browser, which invokes the CRYPTON-KERNEL on-demand to securely interpret CRYPTONs. A compromised browser can deny access to sensitive information (denial-of-service) by refusing to invoke the CRYPTON-KERNEL, but it cannot subvert the integrity and confidentiality of sensitive data. In effect, our design of the CRYPTON-KERNEL allows web applications to specify control on sensitive information, bootstrap rich computation on it and build trusted paths between the user and the sensitive application code [53].

Unlike prior research on origin-based isolation in browsers, we do not isolate code components or memory regions; instead we isolate sensitive data within a web application's origin using authenticated encryption. CRYPTONs make use of *memory encryption* [45], wherein data computed from sensitive data remains encrypted throughout its lifetime inside the untrusted browser's memory. This ensures that any unintended malicious or compromised code, whether part of the browser or the application logic, can only access sensitive data in its encrypted form. Memory encryption allows sensitive data to be opaquely accessed by an untrusted browser functionality (such as the network stack, data stores, language parsers, rendering logic, and so on) as well as application logic, without risking its confidentiality and integrity.

Therefore, the CRYPTON-KERNEL enforces its security guarantees while reusing browser functionality — it does not rely on the browser's same-origin policy enforcement, its components such as the HTML parser, the JavaScript/CSS engine, and the network engine to ensure its security properties. The CRYPTON-KERNEL has a TCB about $30\times$ - $40\times$ smaller than that of a real web browser, such as Firefox and Google Chrome, consisting of roughly $27K$ lines of code. Through manual analysis, we find that this design can prevent more than 92.5% of known security vulnerabilities from exfiltrating or tampering with sensitive information, in a web browser such as Firefox (Section 4).

Evaluation on real-world applications. First, we manually modify three popular web applications to selectively use CRYPTONs to protect sensitive web content within them, including a webmail, a web messenger and a web forum. The results show that the CRYPTON-KERNEL strengthens their security against a broad range of client-side vulnerabilities, with a small (around 3-5 days of) adoption effort. Next, we conduct an extensive macro evaluation of the applicability of CRYPTONs on real-world web sites including Alexa Top 50 websites, such as account signup pages like Gmail, social networking sites like Facebook, banking sites like Bank of America, e-commerce applications like eBay, and several others. We treat all user inputs (form fields, text-boxes, selections, etc.) and other manually marked content in these applications as sensitive information. Our macro study shows the effectiveness of our solution in these real-world web applications. Although not a "turn-key" mechanism, our solution requires only moderate adoption effort in the order of a few days for each application. Moreover, web developers can "opt-in" by selectively converting part of their application logic to secure specific sensitive data or for specific users, without breaking the rest of the application. Finally, we evaluate the performance of our prototype. Although our WebKit-based prototype implementation is still an unoptimized proof-of-concept, we do not perceive noticeable slowdown when running it over real-world applications. Further evaluation also shows that performance overhead from encryption/decryption operations is also modest, less than 8% when 100% of the texts on five test pages from the Dromaeo web benchmarks are marked as sensitive.

Contributions. In summary, we make the following contributions in this work.

- To the best of our knowledge, we are the first to propose a data-centric abstraction for end-to-end data protection in web browsers. We employ memory encryption to protect sensitive data in web browsers, offering strong protection at a low overhead.

- With the new abstraction, we propose a novel solution for protecting sensitive web content with a small trusted computing base (TCB), roughly $30\times$ - $40\times$ smaller than a full browser in our prototype. Our design advocates rethinking the monolithic (all-or-nothing) trust model between servers and clients on the present web.

- We perform a large-scale evaluation of the applicability of intra-origin abstractions like CRYPTONs in existing popular applications. We demonstrate the adoptability of these abstractions into a large number of real-world web sites today with a small developer effort.

2. PROBLEM & OVERVIEW

To illustrate the need for new abstractions for sensitive data, consider the example of a webmail application, such as Gmail. Currently, all code and data in client-side web applications are treated equally in protection. However, this does not match the security requirements in practical settings. Users may use a webmail account to access both leisure emails and corporate emails. As corporate emails may contain sensitive information pertaining to trade secrets or financial information, for corporate security, a user Alice may be willing to instruct the webmail application to mark her corporate emails as sensitive and strongly protect them from being leaked or tampered with. Unfortunately, even if her webmail service provider is willing, it is difficult for it to guarantee the privacy of sensitive emails when they are processed and displayed in the client's web browser. The current web provides no abstractions for webmail applications to isolate corporate emails and specify controlled (but rich) computation on them at the client side.

2.1 Threat Model

In web applications like the above webmail example, sensitive data is exposed to a large threat landscape due to vulnerabilities in the client-side web stack. Below we discuss some of the in-scope threats.

- *Browser components*. Vulnerabilities in browser components can lead to violation of the confidentiality and integrity of sensitive information. We measure the number of such vulnerabilities in Mozilla Firefox. As part of a larger study [24], we manually examine security vulnerabilities of Firefox over the past seven years from its bug database [29]. We find that out of the 360 vulnerabilities accessible to us, at least 288 (80%) ones could expose sensitive information to arbitrary malicious script or binary code running in the browser's chrome privilege.

- *Browser add-ons*. Browser add-ons and extensions may access sensitive data processed in browsers. Although web browsers may prompt users to approve the permissions requested by add-ons during their install time, many users may "click-through" to grant them to the add-ons. We surveyed the Top 30 Chrome extensions, and find that 23 request access to Gmail; if these are installed and have vulnerabilities, they can be compromised by attackers to access sensitive email. 15 out of these 23 have been confirmed to contain code-injection vulnerabilities in Carlini et al.'s recent work [14].

- *Application components*. Web application components can also access sensitive data in web applications. The Gmail Inbox page contains 171.1 KB code (including HTML, CSS, JS), while only 1.7 KB of it needs to access email content. However, client-side web application vulnerabilities are pervasive, including mixed content, unsafe `eval`s, capability leaks, DOM-XSS, and insufficient origin validation.[1]

In this paper, we focus on providing two security guarantees on sensitive data processed in the client-side browser: *integrity* and *confidentiality*. Our design assumes that there are client-side vulnerabilities in web applications (including add-ons) and browsers, but trusts the underlying OS, its windowing and graphics interfaces. Our defenses preclude many vulnerabilities (outlined above) from corrupting sensitive data or leaking unprotected sensitive data to adversaries. For instance, attackers can exploit client-side vulnerabilities to leak sensitive emails to remote adversaries. However, in our solution, those emails are in encrypted form and the decryption keys are inaccessible to attackers. Several attacks are outside the scope of the guaranteed security properties; we discuss them in Section 2.4.

2.2 CRYPTON: A New Abstraction

We introduce a new data protection abstraction called CRYPTON. Conceptually, CRYPTONs are akin to classes in object-oriented languages, which encapsulate sensitive data together with the operations that can legitimately operate on it. CRYPTONs provide the following 4 main capabilities.

Information isolation. We enable transparent isolation of sensitive data from other data and code in web browsers. In our webmail example discussed earlier, the confidentiality of the sensitive corporate emails must be maintained by authenticated encryption. Other examples include credit card numbers, contents of shopping carts in e-commerce applications, user passwords in login pages, payee accounts in online banking sites, and so on. Our assumption is that preventing malicious code from running in the complex web browser is not tractable; however, we aim to prevent unauthorized leakage or tampering of sensitive data.

Controlled operations. If we isolate sensitive data from all computation, it would severely disincentivize web applications from outsourcing rich functionality to the client-side browser. There is a trade-off between the information isolation guarantees and the richness of functionality that can operate on sensitive information. In our webmail example, Alice should be able to use the webmail's interfaces on her corporate emails for reading, composing, formatting (configuring fonts, colors, HTML formatting or alignments), archiving, forwarding and so on. For practical usage, we aim to allow such owner-specified operations to compute output from sensitive data. However, such output must be protected from confidentiality and integrity attacks by all other code in the browser, and thus is encrypted by default. The data owner (e.g., the webmail server) can also specify which operations reveal computed information; however, we leave the design and choice of these functions up to the owner. In this work, we make encryption possible on simple data types, such as strings and integers, to limit the TCB size — of course, more complex data types (such as arrays, formatted HTML) can still include encrypted data in them (see Section 3.4).

Certified user inputs. In this work, we focus on enabling users to enter sensitive text strings via a trusted keyboard[2]. Specifically, we provide a trusted path for user inputs between the OS keyboard events and the trusted web application event handlers (specified as CRYPTON functions), through the untrusted web browser. Our focus on keyboard events is guided by the observation that a significant fraction of sensitive data on the web pertains to user keyboard inputs. In our webmail example, Alice signs in by using her password, composes a sensitive email by typing, and searches for keywords also by using the keyboard.

Proof of impression. Malicious client-side code can attempt to disable critical messages that are supposed to warn users of potential threats. For example, the webmail server may show a warning for emails suspected to be scams and banks may want to warn of fraudulent activities. As another example, when the webmail server detects suspicious login activities, such as simultaneous logins from different geolocations, it alerts users to change their passwords. Such messages are crucial, and may be helpful for users to prevent attacks or to stop them at an early stage.

[1]Gmail does not include external libraries and does not mix content from HTTP sites; nevertheless, several other web applications we study in Section 4 include external libraries and mixed content, which poses serious threat to sensitive information.

[2]Supporting trusted paths for other input devices, such as the mouse, touch screen and stylus, is future work.

In this work, we focus on providing a "proof of impression" of simple data types such as strings and byte streams. This includes verifiable rendering of string content and bitmaps, during which rendering of other untrusted content into the same region is temporarily blocked for t seconds (t=3 by default) [36]. Such a guarantee is useful beyond the webmail example, such as online advertisements (ads). Proof of impression can help advertisers that bill publishers for ad impressions to disambiguate real ad impressions from ad fraud with "laundered" traffic [26].

2.3 Design Overview

Our design changes the monolithic trust model where web servers trust the entire browser (including add-ons) and client-side application code. In our design, developers encapsulate sensitive data, such as corporate emails or bank statements, into CRYPTONs and directly embed them in HTML documents. To enforce the security guarantees and semantics of CRYPTONs, the web server trusts a small, standalone engine at the client called the CRYPTON-KERNEL. The CRYPTON-KERNEL executes outside the web browser and is protected from a compromised browser by standard OS sandboxing mechanisms, similar to those used by existing browsers such as Google Chrome [56,57]. A CRYPTON-compliant web browser recognizes CRYPTONs during HTML document parsing and delegates their handling to the CRYPTON-KERNEL, as detailed in Section 3.

The CRYPTON-KERNEL acts as a root-of-trust to protect sensitive CRYPTONs at the client-side. Specifically, the CRYPTON-KERNEL builds three trusted paths passing through the untrusted browser: (a) a secure communication channel between the web server and the CRYPTON-KERNEL, (b) a trusted path from the CRYPTONs to the user's screen (i.e., to the software rendering pixel maps or the GPU display buffers), and (c) a trusted path between the user's keyboard inputs to CRYPTON functions that handle keyboard events. These trusted (data) paths run through the untrusted browser code, reusing existing browser functionality. As we introduce later, we use authenticated encryption to protect data on these trusted paths [45].

Secure Channel to Server. At the start of a web session exchanging sensitive data, the web server uses an HTTPS frontend to establish a secure channel with the trusted CRYPTON-KERNEL. This secure channel enables the web server to share a session key set $\mathcal{K}$ with the CRYPTON-KERNEL, used by both parties to encrypt/decrypt CRYPTONs in the session. To avoid bloating our TCB, the CRYPTON-KERNEL delegates all network communication to the untrusted browser. Thus, it is important to establish a secure channel against man-in-the-middle attacks, in case the browser gets compromised. We use the standard mutual SSL authentication protocol to establish this channel [28]. SSL certificate verification for server authentication is a common, backward-compatible mechanisms for HTTPS sites today. To enable the server to authenticate the CRYPTON-KERNEL's certificate, we assume the CRYPTON-KERNEL uses a certificate self-signed by an RSA private key stored in the CRYPTON-KERNEL. The corresponding public key is uploaded by the user to the server via an out-of-band secure channel. We discuss the potential usability challenges of this mechanism in Section 3.5.

Secure Display. When being displayed to users, sensitive content, such as corporate emails, must remain encrypted in the untrusted browser until the final rendering of the content. The CRYPTON-KERNEL provides a trusted data path between the CRYPTON content received from the server to the GPU buffer [5], which finally render bitmaps and pixels to the screen. This trusted path is implemented as part of the CRYPTON-KERNEL. To maintain the browser's functionality, sensitive content must be given opaque access to the untrusted browser, say for deciding the layout dimensions or for storing them in the DOM or untrusted JavaScript heaps. Hence, the CRYPTON-KERNEL encapsulates encrypted sensitive content into *opaque objects* before passing them to the web browser. These opaque objects (e.g. encrypted string class) allow the browser to process sensitive data as opaque blobs of text, determine the length of the underlying plaintext for layout, but do not permit any malicious operation that would leak plaintext data. Opaque objects are finally rendered by the CRYPTON-KERNEL using a "sandboxed GPU buffer" mechanism [5], which enables it to ensure that sensitive content is rendered on top for at least t seconds.

Trusted Path For User Input. For sensitive content generated at the client, such as by user keyboard inputs, the CRYPTON-KERNEL provides a trusted path between the OS keyboard events to the CRYPTON functions designated to handle input events. To enable it, the CRYPTON-KERNEL intercepts all keyboard events from the OS, similar to Chrome's sandboxing mechanism [5]. It uses opaque string objects to establish the trusted data paths via the untrusted browser code.

We expect the user Alice to recognize and enter sensitive keyboard inputs (e.g., passwords or email recipients in TO and CC list) through the CRYPTON-enabled trusted path. To help users securely interact on and across multiple CRYPTON-enabled sessions, the CRYPTON-KERNEL displays two security indicators above the untrusted browser UI. For distinguishing when her user inputs are going to be encrypted (vs. when she is interacting with a non-CRYPTON-enabled session), the CRYPTON-KERNEL displays a user-selected secret image icon. This image icon indicates that the user's key events will be encrypted throughout the session. This indicator informs users whether their sensitive keystrokes are going to be encrypted; we expect users to enter sensitive information only when seeing the right image icon.

Second, the CRYPTON-KERNEL displays the URL of the website whose keys are being used to encrypt the keystrokes, in a special unspoofable URL bar above the browser chrome. This allows Alice to distinguish between multiple CRYPTON sessions and identify to which of these sessions her present keystrokes are being delivered. This indicator informs the user of the intended recipient of the encrypted keystrokes; users are expected to verify this URL indicator and ensure that it is the right recipient for the sensitive data they are about to enter.

Both UI indicators are securely rendered to the screen directly by the CRYPTON-KERNEL and can not be overlaid by the attacker. The CRYPTON-KERNEL does not allow the untrusted browser to inject key sequences or read sensitive GPU buffers using standard sandboxing techniques. We discuss more details in Section 3.

Assumptions. A CRYPTON-compliant server uses an HTTPS frontend as the server-side interface to the CRYPTON-KERNEL. HTTPS frontends often handle authentication and are commonly used for load balancing, before delegating the session to backend server — we believe our design can be deployed in existing HTTPS frontends. CRYPTON-compliant servers default-deny access to protected sensitive content outside the secure channel established with a trusted CRYPTON-KERNEL. Otherwise, this would incentivize attackers to fool users into accessing the sensitive content over non-CRYPTON-enabled sessions.

To protect the CRYPTON-KERNEL, our solution leverages OS processes and system call sandboxing to isolate the untrusted browser from the CRYPTON-KERNEL. Robust sandboxing mechanisms are known and used by existing browsers such as Chrome [5]. For additional protection, especially in emerging Web OS stacks [51, 55], hardware-assisted dynamic root of trust measurement (DRTM) (such as those provided by Intel TXT [37, 52]) can be investigated in the future to ensure dynamic code integrity of the CRYPTON-KERNEL.

2.4 Out-of-scope Threats

Our focus is on providing the integrity and confidentiality of data. So, for example, our defenses cannot be used to protect data used in server-side authorization such as CSRF tokens, session cookies and authentication tokens (for single-sign on mechanisms), as blocking these attacks require ensuring the authenticity of the sender of sensitive data. Similarly, browser vulnerabilities may be used to completely deny access to sensitive data (denial-of-service); our defenses do not enforce data availability to the intended recipient.

There are various attacks that can elicit sensitive information from the user *outside* the trusted paths, say by confusing the user with fake security indicators [39, 71], exploiting time-of-check-to-time-of-use (TOCTTOU) windows in our security indicators, clickjacking [3, 36], shoulder surfing or through social coercion [8]. We recognize that these are important channels of information loss to consider for guaranteeing end-to-end security; however, defenses for these specific attacks are of independent interest [20, 58].

Moreover, we point out that our design requires developers to carefully design CRYPTON functions to avoid direct data leakage as well as side-channels via timing and control flow, which are declassification interfaces to sensitive data. There are also well-studied classes of attacks on information flow and encryption systems [11, 15, 18]; tools to detect and minimize the capacity of side channels are useful for developers [42].

3. DESIGN

We propose a novel solution with CRYPTON abstractions to protect sensitive data in an untrusted web browser. In this section, we first introduce the definition of the CRYPTON, which can be directly programmed into web pages. Then we elaborate the design of the CRYPTON-KERNEL, and security invariants it enforces.

3.1 CRYPTON Definition

The CRYPTON abstraction incorporates sensitive data protected with integrity and confidentiality, the operations permitted to decrypt and process the data, as well as the functional policy that determines which keys to use for decrypting sensitive information and for encrypting outputs.

More formally, we define a CRYPTON as a 5-tuple $\mathcal{W} = (\vec{\mathcal{D}}, \vec{\mathcal{F}}, \mathcal{U}, \mathcal{V}, \mathcal{I})$. As defined later, $\vec{\mathcal{D}}$ is a sequence of information blocks and $\vec{\mathcal{F}}$ is a sequence of CRYPTON functions. $\mathcal{U}$ is the web address URI of the CRYPTON-enabled web server, the HTTPS frontend of the web server. The external verifier $\mathcal{V}$ is the HTTPS web URI for the proof-of-impression functionality, as we explain shortly. $\mathcal{I}$ is a unique identifier (ID) for the CRYPTON.

A CRYPTON $\mathcal{W}$ syntactically binds permitted operations and sensitive data in its definition; this binding is signed by the SSL public key corresponding to $\mathcal{U}$ and is verified and maintained by the CRYPTON-KERNEL. When the CRYPTON-KERNEL first processes a CRYPTON, it communicates with $\mathcal{U}$ over a secure channel and fetches a set of symmetric encryption keys $\mathcal{K}$ corresponding to that specific CRYPTON. Note that CRYPTON keys $\mathcal{K}$ are not sent as part of $\mathcal{W}$, which would otherwise permit them to be read by adversaries. The set $\mathcal{K}$ consists of a default encryption/decryption key κ_0, an HMAC key κ_{hmac}, and optionally other keys $\kappa_1, ..., \kappa_n$, for authenticated encryption using symmetric key ciphers (256-bit AES-GCM). These keys are kept secret and known only to the

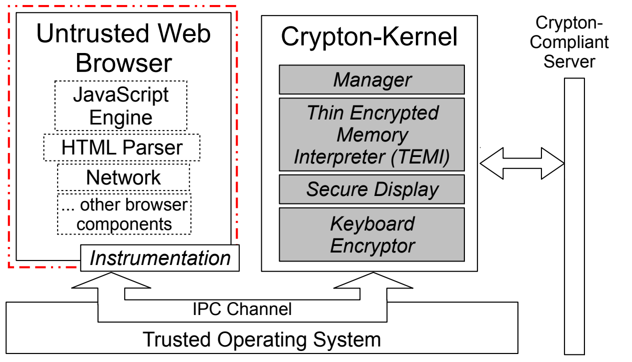

Figure 1: Overview of the CRYPTON-KERNEL, *consisting of 4 components in gray, and the sandbox shown dashed.*

CRYPTON-KERNEL; they are referenced by their key indices in CRYPTON functions and information blocks as explained below.

Conceptually, each CRYPTON function $\mathcal{F}_i$ is a 3-tuple $\mathcal{F}_i = (\mathcal{P}, \mathcal{R}, \tau_{\mathcal{F}})$. The syntactic definition of $\mathcal{F}_i$ binds tuple elements together; these bindings are signed to ensure their integrity. $\mathcal{P} : \mathcal{K} \times \mathcal{K}$ is a set of policies defined for each $\mathcal{F}_i$ over keys in $\mathcal{K} \cup \{\perp, \kappa_{int}\}$, where $\perp$ means public (no encryption)[3], and κ_{int} is a CRYPTON-KERNEL-specific key to protect user inputs, such as keystrokes. Such a policy dictates how a CRYPTON function encrypts and decrypts sensitive data. A policy $\kappa_1 \rightarrow \kappa_2$ for a function $\mathcal{F}_i$ indicates that $\mathcal{F}_i$ decrypts sensitive data using the key κ_1. Global or local variables written during the function execution, arguments of calls to the untrusted browser, as well as return values, are encrypted with the key κ_2. Policies are optional; the default policy enforced is $\kappa_0 \rightarrow \kappa_0$, i.e., decryption and encryption both with the default key. We explain the semantics of the execution of each function in detail in Section 3.4. $\mathcal{R}$ optionally lists the arguments to the function and the return value. Finally, functions have a type field $\tau_{\mathcal{F}}$ which separates 2 kinds of functions: (a) a proof-of-impression (`PoI`) function that must be invoked to handle impression tokens, and (b) all other functions. `PoI` functions compute encrypted messages over the nonce token and the impressions of sensitive messages and send them to the external verifier $\mathcal{V}$.

An information block is a 6-tuple $\mathcal{D}_i = (\mathcal{I}, \kappa, \mathcal{I}_{\mathcal{D}}, \mathcal{A}, \tau_{\mathcal{D}}, \mathcal{F}_{\mathcal{D}})$. $\mathcal{I}$ and $\mathcal{I}_{\mathcal{D}}$ are identifiers for the owner CRYPTON and the information block, respectively. The encryption key for $\mathcal{D}_i$ is specified by $\kappa \in \mathcal{K}$. $\mathcal{A}$ is the encrypted ciphertext of the enclosed sensitive data. Finally, if a certain information block requires proof-of-impression for its render value $\mathcal{A}$, the field $\tau_{\mathcal{D}}$ indicates the nonce token for this information block, and $\mathcal{F}_{\mathcal{D}}$ refers to the function to be invoked with the impression token by the CRYPTON-KERNEL.

3.2 CRYPTON-KERNEL Design

The CRYPTON-KERNEL provides secure processing of sensitive web content for a CRYPTON-compliant web browser, while never exposing decrypted data to the browser. As shown in Figure 1, our solution needs to instrument the untrusted browser, which communicates with the CRYPTON-KERNEL running in another process via IPC channels. We develop a new thin engine called the *Thin Encrypted Memory Interpreter* (*TEMI*) to execute CRYPTON functions over sensitive data. We now present a brief overview of our solution with a hypothetical example demonstrating protecting sensitive emails in a webmail application.

[3]The CRYPTON function can also call the `enc(data, key)` programmatically to encrypt part of the outputs on demand.

When the CRYPTON-compliant web browser parses an HTML document, it encounters the CRYPTON header, functions or information blocks. At this time, the browser invokes the CRYPTON-KERNEL via an IPC interface. Once invoked, the Manager component of the CRYPTON-KERNEL processes these special tags, including validating their integrity, and retrieving CRYPTON keys via a secure channel from the server URLs embedded in the CRYPTON header. The Manager component returns opaque objects to the untrusted browser for processed CRYPTONs. These opaque objects encapsulate encrypted values and metadata such as the corresponding CRYPTON ID and decryption key. They are propagated internally by the untrusted browser, akin to ordinary browser objects. (Step *A* in Figure 2).

The untrusted browser handles the layout of web content, and composes intermediate constructs for rendering. Another CRYPTON-KERNEL component, called the Secure Display, intercepts rendering requests from the untrusted browser, and converts the layout constructs into pixel maps written into the GPU buffer. When opaque objects enclosing encrypted texts traverse through untrusted code paths in the browser, and finally reach the Secure Display component, the Secure Display decrypts them before transforming them into pixel data for the GPU buffer. For example, sensitive emails in our example are encrypted when they are downloaded into the browser. When they are being rendered, the Secure Display component of the CRYPTON-KERNEL decrypts them into plaintext that is displayed to users (Step *B*).

To protect user inputs, the Keyboard Encryptor component of the CRYPTON-KERNEL encrypts all user inputs for the CRYPTON-enabled webmail application. For example, it encrypts the search keywords while the user enters them in the untrusted browser. The resulted opaque object containing the encrypted user inputs is finally dispatched into the event handler function `searchEmail` triggered by a user click on the "Search" button. The webmail developer marks `searchEmail` as a CRYPTON function with a policy $\kappa_{int} \rightarrow \kappa_0$, allowing it to decrypt user inputs (such as search keywords) encrypted with the CRYPTON-KERNEL's internal key κ_{int}. When `searchEmail` is invoked, the untrusted browser transmits the execution to the TEMI, a thin execution environment in the CRYPTON-KERNEL. The TEMI decrypts the search keywords encrypted in the opaque object during the execution of `searchEmail`. This allows the function to preprocess the search terms, such as replacing "+" with "ADD", and encode the texts. Then `searchEmail` constructs an XMLHttpRequest containing the preprocessed search terms and sends it to the webmail server. When the webmail server receives the request, it decrypts the search terms and searches for them in the email database (Step *C*).

Subsequently, search results are returned to the untrusted browser in JSON format where innermost email headings are encrypted. The registered XMLHttpRequest handler function `parseJson` is invoked to tokenize and validate the JSON string, and return a composed HTML fragment to be rendered to the user. The email headings in the HTML fragment remain as opaque objects to untrusted code in the browser, and the Secure Display decrypts them when rendering the headings for the webmail user (Step *D*).

3.3 Security Invariants

To protect the confidentiality and integrity of sensitive web content in an untrusted browser, our solution enforces the following semantics as security invariants for CRYPTONs.

For information isolation:

P0: Maintaining Syntactic Bindings. A CRYPTON syntactically binds its elements to each other (such as $\mathcal{K}$ to $\mathcal{I}$, $\mathcal{P}$ to each $\mathcal{F}_i$ and so on) with mentioned keys. These bindings are checked and enforced throughout the lifetime of the CRYPTON. Authenticated encryption ensures that any attempt to tamper with such bindings by malicious code will be prevented.

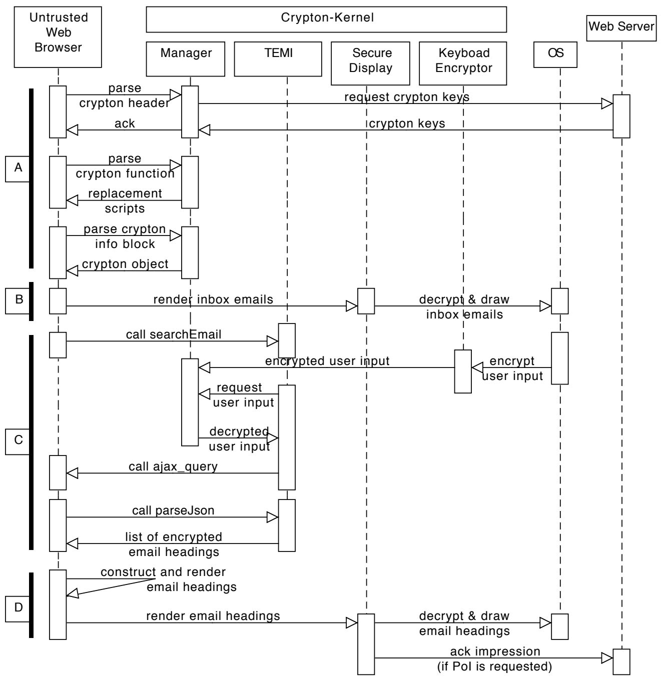

Figure 2: Sequence of Operations in A CRYPTON-Enabled Webmail Session

P1: Secure Storage. All CRYPTON information blocks, functions and intermediate computation by CRYPTON functions is stored in private memory outside the browser, or encrypted when stored in memory shared with the browser. The standard guarantees of authenticated encryption apply to these.

P2: Secrecy of $\mathcal{K}$. The keys $\mathcal{K}$ are only stored in the CRYPTON-KERNEL and not accessible to the CRYPTON-compliant browser.

For controlled operations:

P3: Functional Policy. If a function $\mathcal{F}_i$ is bound to a policy $\kappa \rightarrow \kappa'$, then all sensitive data processed by it is decrypted using κ, and all values written by $\mathcal{F}_i$ are encrypted with κ', e.g., modified global variables, local variables, arguments of calls to untrusted browser functions, and return values. See **P5** as the only exception.

P4: Non-interfering Execution. The execution of two functions $\mathcal{F}_1$ and $\mathcal{F}_2$ are non-interfering on the CRYPTON-KERNEL, if they are bound to policies $\kappa_1 \rightarrow \kappa'_1$ and $\kappa_2 \rightarrow \kappa'_2$ respectively, and $\{\kappa_1, \kappa'_1\} \cap \{\kappa_2, \kappa'_2\} = \emptyset$.

P5: Information Release. Only functions bound to policies $\_ \rightarrow \perp$ output plaintext to the browser[4].

For certified user inputs:

P6: User Input Encryption. User keyboard inputs are encrypted with the CRYPTON-KERNEL's internal key κ_{int}, and only CRYPTON functions bound to policy $\kappa_{int} \rightarrow \_$ can decrypt the inputs.

P7: Restricted Keyboard Access. The CRYPTON-KERNEL intercepts all keyboard events before they are delivered to the browser.

[4] _ denotes any key or $\perp$.

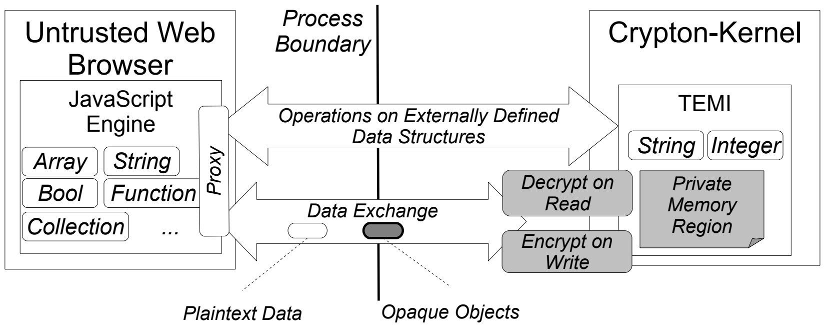

Figure 3: TEMI: a Thin Execution Environment for CRYPTON Functions to Access Sensitive Info

For proof of impression:

P8: Proof of Impression. When an information block $\mathcal{D}_i$ is *rendered*, it should be displayed for a certain period of time without obstruction or overlay by other UI content. The `PoI` function specified by the developer is invoked to compute an encrypted message acknowledging the rendering to the verifier $\mathcal{V}$ in a secure channel.

P9: Restricted Display Access. The CRYPTON-KERNEL mediates the rendering operations initiated by the browser.

Summary. With these security invariants, CRYPTONs ensure the confidentiality and integrity of enclosed sensitive data throughout its lifecycle in untrusted web browsers.

3.4 Key Techniques and Security Analysis

To ensure the security invariants above, the CRYPTON-KERNEL provides a thin engine for permitted operations on sensitive data, and interposes browser display and keyboard events to support trusted paths for display and user inputs.

Thin Encrypted Memory Interpreter. To execute controlled operations on sensitive data, we design the Thin Encrypted Memory Interpreter (TEMI) — a thin scripting engine. In designing TEMI, we face a tradeoff between supporting rich functionality and minimizing the TCB. A full-fledged JavaScript engine typically has 200,000 (JavaScriptCore) to 400,000 (the V8 engine) LOC. However, we observe in our case studies that most of potential sensitive data types on the web are integers and strings, such as user names, passwords, entries in medical records, credit card numbers, etc. These data types account for only a very small fraction of the implementation of the JavaScript engine. Of course, these primitive types are aggregated into higher-level abstract data types (such as arrays, lists, objects, and so on), but much of the logic that operates on the higher-level data types is agnostic or independent of the inner nested data. By leveraging this observation, we design a "thin" TEMI that natively supports integers and strings[5], which has a TCB of only $19,000$ lines of code.

The TEMI runs inside the CRYPTON-KERNEL's process, communicating with the process(es) of the untrusted web browser via IPC messages (shown in Figure 3). The TEMI has a private virtual register set (for local computation) in its private memory region. It decrypts sensitive data in opaque objects when they are loaded into virtual registers, and re-encrypt all values written from virtual registers to opaque objects, including global variables, arguments, and return values. By default, the TEMI uses the default CRYPTON key κ_0 for encryption and decryption; a developer-specified policy $\kappa_1 \rightarrow \kappa_2$ of a CRYPTON function dictates κ_1 used for decryption, and κ_2 for encryption (**P3**)[6]. A CRYPTON function that is allowed to disclose information has a policy $\_ \rightarrow \perp$ (**P5**). The TEMI retrieves CRYPTON keys from the CRYPTON-KERNEL's private memory storing keys transferred from the web server (**P2**).

During the execution of a CRYPTON-KERNEL function, any access to higher-level abstract data types results in an IPC message to a proxy in the JavaScript engine of the untrusted browser with opaque objects as arguments. The proxy translates the message into operations that are executed in the untrusted JavaScript engine. For operations initiated by the CRYPTON-KERNEL, any string and integer operation that emerges during execution in the untrusted JavaScript engine is tunneled back to the CRYPTON-KERNEL for processing opaque objects. In our running example in Section 3.2, the `parseJson` function sends a message to the untrusted browser to access complex data structures, such as arrays to perform parsing operations. The execution then switches back to the TEMI when an `Array.toString` function is invoked so that operations on sensitive strings can still be processed in the decrypted form. This unmodified CRYPTON function runs in our TEMI as it switches back and forth between the untrusted browser and the CRYPTON-KERNEL. For data exchange, the TEMI and the untrusted browser passes integer and string data in the IPC messages, while only pass references to high-level data structures as they are processed exclusively at the untrusted browser side.

In our design, accessing higher-level data types defined in the untrusted browser from the TEMI incurs no explicit data leakage, since all arguments are encrypted (**P1**). Implicit control flow leakage is possible, while we assume it can be largely ignored in legitimate code [15], such as CRYPTON functions. For compatibility with existing browsers, when one process is running JavaScript, we halt the other process until the running process returns.

To prevent interference between different CRYPTON functions, the TEMI executes each CRYPTON function in a separate context, which is destroyed upon returning of the function (**P4**). This includes virtual registers stored in the private memory region, as well as encrypted intermediate variables written in the shared memory region. Similar to garbage collectors, the TEMI clears all these values when a CRYPTON function returns.

Secure Display. The CRYPTON-KERNEL allows automatic decryption of opaque objects when they are being rendered, but prevents untrusted browser code to access the decrypted information.

[5]including regular expressions and basic indexing operations on static arrays of integer or string elements.

[6]Note that in principle it is possible to have explosive volumes of encrypted data from the computation over mixed sensitive and non-sensitive data. We did not observe such cases in our experiments with real-world web applications, where computation over sensitive data does not yield new sensitive data, but just outputs features or portions of original data.

The Secure Display component of the CRYPTON-KERNEL can use sandboxing techniques similar to Google Chrome [5], such as restricted tokens and Windows job and desktop objects on Windows [57] and `seccomp` on Linux [56], to intercept untrusted browser's access to the GPU buffer and graphic libraries on the OS (**P9**). To render any UI content, the untrusted browser must pass the rendering requests to the Secure Display. The Secure Display in turn converts web contents and browser widgets into intermediate rendering constructs (such as `Glyphs` that are basic shapes in text rendering [13]) and sends them to the graphics library for rendering. During this process, the Secure Display decrypts opaque objects containing sensitive strings, with the keys specified in the opaque objects. To prevent UI obstruction, while rendering such opaque objects, for t seconds, the Secure Display temporarily suspends other rendering requests to the same destination GPU buffer location. It also invokes the `PoI` function for the opaque string being rendered, if applicable. The `PoI` function then computes an acknowledgement token encrypted with the CRYPTON's default key κ_0 and sends it to the external verifier (**P8**).

Keyboard Encryptor. The Keyboard Encryptor interposes on keyboard events from the graphics toolkit that receives keystrokes. It encrypts each keystroke event with a CRYPTON-KERNEL-specific internal key κ_{int} before it reaches the web browser code (**P6, P7**). Therefore, web applications cannot read the clear-text of user inputs unless by calling the CRYPTON functions with a policy like $\kappa_{int} \rightarrow \_$. Together with **P3**, this ensures that keyboard inputs are protected, and are available only to intended functions.

Resilience against malicious code in web browsers. Although the CRYPTON-KERNEL closely integrates the untrusted web browser, the security guarantee ensured by the CRYPTON-KERNEL are independent from the untrusted web browser, i.e., all disclosure of the sensitive information is controlled by CRYPTON functional policy.

To allow non-invasive operations that do not require plaintext, the encrypted data in CRYPTON information blocks is stored on the browser's storage as opaque objects. These objects are protected by authenticated encryption for their integrity and confidentiality. The keys are available only to the CRYPTON-KERNEL. Untrusted code has no access to the keys (**P2**), and thus cannot decrypt sensitive data, unless by explicitly invoking information releasing CRYPTON functions (**P5**). User inputs are automatically encrypted before they reach the untrusted web browser (**P6**), so they are under the same protection as sensitive data coming from web servers. Web servers can also verify whether sensitive messages are properly displayed via proof of impression (**P8**).

Although malicious code cannot decrypt the sensitive data in ciphertext, it can tamper with it. However, this will be detected by integrity verification built into authenticated encryption. This ensures the integrity of all CRYPTONs from the web and sensitive user inputs at the client.

Attacks on CRYPTON granularity. Our solution enables web servers to distinguish data from client side at a binary granularity — i.e., between authentic user inputs and everything else. Consider the following attack. Malicious code injected into a webmail application can create a fake login box in the victim session. This fake login box makes the email composing input box appear as its Password field. A victim user may then enter her password into the fake Password field, which is sent to the attacker in an email. This is an instance of self-exfiltration attacks [17]. To defeat this attack, the web server can mark the TO field for email recipients as sensitive, and require its value to come from authentic user inputs at the client side. Therefore, malicious client-side code has no control over to where the email will be delivered, even if it can trick the user into entering her password as the email body. However, our current solution only distinguishes authentic user inputs from other data. Attackers may still confuse users between the semantics of two CRYPTON-protected user input fields, such as luring the user into entering the email body into the TO field by phishing [31,72,73]. In certain applications, this could lead to delivering emails to unintended recipients. However, we believe such attacks are application-specific and difficult to launch. Further, extensions of CRYPTONs with finer-grained isolation can be explored to defeat such attacks.

3.5 Usability Implications

We discuss the usability implications that may arise from our design. First, our solution assumes that users only enter their sensitive data when security indicators state that their inputs are being encrypted for the expected web application. Previous research has proposed various alternatives, such as Bumpy [49], where a special key stroke sequence is required to start a secure input session. Instead of relying on users to initiate the securing of their inputs, in this work, we automatically encrypt all user inputs within a CRYPTON-compliant web session. As our solution requires users to pay attention to secure UI indicators during keyboard input, we expect our solution to have similar usability challenges as reported in a prior evaluation of various alternative secure input mechanisms [44]. We anticipate our solution to be usable in enterprise settings or in mission-critical applications.

Secondly, the initial key setup in establishing a CRYPTON-KERNEL-to-server secure channel could pose a usability challenge to non-security-experts, as it requires key upload. Other alternatives could be considered in a full deployment, such as adopting a trust-on-first-use model [70] as with SSH, or the PAKE protocol [27] to establish a secure channel based on a shared secret, which have their own pros and cons [9]. Adopting such alternatives does not affect our core approach and mechanisms proposed in this work. We choose the current setting for its strong security guarantees and its compatibility with HTTPS. More recently, various online hosting services, such as Github [32] and BitBucket [6], have adopted such a mechanism to authenticate clients for more adept users. We expect a similar user experience for using a CRYPTON-enabled website in enterprise settings, and a gradual broader adoption among general web application users.

3.6 Implementation & Deployment

We implement a prototype of the CRYPTON-KERNEL integrated with WebKit-GTK (rev 45311) with JIT disabled, for the ease of implementation. Our prototype reuses OpenSSL code for AES-GCM encryption/decryption. As the WebKit-GTK that we use does not leverage hardware-accelerated rendering, we implement text decryption and proof of impression via the software paths by intercepting calls to the bridge between WebKit and the cairo graphics library. Similarly, we intercept signal dispatch from the GTK+/Glib to the browser to encrypt user inputs. As user input events are fired for each keystroke, we use a stream cipher to encrypt user inputs. Thus, positions of user input characters are maintained as original, and this naturally supports mouse text selection. When user inputs are sent to web servers, we use a CRYPTON function to re-encrypt them with the AES-GCM block cipher with the specified key. Table 2 lists the sizes of components in our prototype implementation of the CRYPTON-KERNEL. Comparing to a web browser, such as Firefox and Chromium, which typically has around 800K to 1,100K lines of source code, the TCB in our unoptimized prototype is about $30\times$ - $40\times$ smaller.

App Name	Sensitive Data	Server-side Change	Client-side Change	Est. Total Conversion Effort
RoundCube	Mails with subject marked as "[sensitive]"	8 LOC php	9 Functions, 0.5K LOC JS	2-3 Man-Days
AjaxIM	Instant messages	6 LOC php	9 Functions, 0.4K LOC JS	2-3 Man-Days
WordPress	Blogs containing "[sensitive]" in titles and search keywords	10 LOC php	19 Functions, 1.2K LOC JS	4-5 Man-Days

Table 1: Summary of Case Studies on 3 Popular Web Apps, demonstrating modest adoption effort

CRYPTON-KERNEL **Component**	**LOC**
Manager	2.9K
TEMI	18.7K
Secure Display	4.8K
Keyboard Encryptor	0.5K
Total	26.9K

Table 2: Size of TCB in CRYPTON-KERNEL **Prototype**

Our prototype implements functionalities sufficient to support our studies with real-world web applications. In a full deployment, several browser components need to be modified to propagate the opaque objects with encrypted data, including support for untrusted browser sandboxing, browser spell checkers, all CSS styling features, web page printing, WebGL and GPU accelerations. Such support has not been implemented in our current prototype.

For deployment into real web browsers, we consider an alternative deployment into Google Chrome's browser kernel, leveraging existing privilege separation and sandboxing mechanisms in Google Chrome. Google Chrome partitions more vulnerable components into the renderer processes, and leave more security-sensitive components in the browser kernel process. It has also implemented sandboxing mechanisms to prevent renderer processes from directly accessing UI rendering, user inputs, file systems, and network. All such access has to go via interfaces exposed and checked by the browser kernel. Thus, we expect it to be relatively straightforward to implement the CRYPTON-KERNEL into the browser kernel of Google Chrome, leveraging the existing UI and user input sandboxing mechanisms. If we consider such a modification to Chrome 12, the code size that can access sensitive data will be significantly reduced. Essentially, it would eliminate more than 900K lines of code constituting the renderer processes and all client-side application code from the TCB for ensuring data protection.

4. EVALUATION

We apply our solution to real-word web applications to study the applicability and adoption cost. We first manually convert 3 popular web applications to use our solution to protect typical sensitive data in those applications, and then extend our study to Alex Top 50 web pages. Both micro and macro studies show the effectiveness of our solution in protecting typical sensitive data on the present web with modest adoption effort required.

4.1 Applicability to Real-world Applications

Micro-study on open-source web applications. We perform case studies on three open-source web applications to measure two aspects, i.e., *a)* how effectively our solution can protect sensitive content in real-word applications, and *b)* how much developer effort is required for adopting our solution. We choose applications of different categories: RoundCube [65], a webmail server, AjaxIM[7], a web-based instant messenger, and WordPress [30], a web blog service. We manually convert the source code of the three applications by: 1) modifying the server-side code to encrypt sensitive content before sending it to clients; 2) identifying client-side JavaScript functions that need to decrypt sensitive data for operations and converting them into CRYPTON functions; and 3) rewriting client-side JavaScript functions that receive and process user inputs into CRYPTON functions; these CRYPTON functions encrypt user inputs before sending them to web servers. We check all tests with a server-side proxy.

We find that with modest effort in converting these applications, our solution can effectively protect typical sensitive data, such as sensitive emails, instant messages, blog entries and comments. We write a 450-line custom PHP library for common functionalities to process CRYPTONs in PHP applications. We manually rewrite the three web applications, leveraging the custom library to wrap sensitive web content into CRYPTONs. Table 1 summarizes the results of our case studies on the three applications. Typical client-side operations on sensitive data we observe include trimming whitespaces in strings, serializing HTML content, emotion text replacement, URI encoding, etc. We mark them as CRYPTON functions in our experiments to support the legitimate functionalities. For brevity, we leave out detailed steps here, and a summary of our modification to application source-code is available online [1].

Macro-study on real-word web applications. To further evaluate the applicability of the CRYPTON-KERNEL to other web applications, we perform a larger-scale macro study. First, we select Alexa Top 50 web pages, and identify all of those fields that require sensitive username / password inputs for signup – 18 out of the 50 applications have signup pages (Figure 4). We choose these applications because they often have client-side checking code on sensitive passwords (e.g. checking strength requirements). Next, we select 20 popular web applications of five categories, and identify scenarios where the CRYPTON-KERNEL can strengthen security against real attacks:

- *Web search pages.* We select websites that allow users to search terms. We mark search terms as sensitive because leaking search terms may permit third-party tracking.
- *Social networking sites.* These sites can be used to exchange private messages, or post comments that may be politically-sensitive. Hence, we mark posts and comments as sensitive.
- *Banking sites.* Banking sites are prime targets of browser-based attacks and several malware disguise as browser extensions. We select a local bank that uses additional authentication mechanisms such as one-time PassKeys. In its web pages, username, passwords and the one-time PassKey are marked as sensitive. The one-time PassKey is interesting because banks are increasingly considering this as a second-factor authentication beyond long-lived passwords.
- *E-commerce sites.* We test eBay, Amazon and Babylon online commerce sites. On eBay, we mark the auction listing created by a seller as sensitive. Different auction listings can

[7]We use the AjaxIMRPG fork [23] that is better maintained than the trunk.

Web App Detail	Sensitive Info	# & Size of Functions Requiring Decrypted Sensitive Info [Percentage of Total Code Size]	# & Size of All JavaScript Functions	# Browser and TEMI Interactions vs. # All Calls to JS Data Types
Gmail Login	Username, Password	2 (0.29 KB) [**0.43%**]	196 (66.5KB)	**0.7%** (956/141492)
Gmail Compose Email	To, Subject, Content	9 (1.84KB) [**0.98%**]	745 (186.8KB)	**0.01%** (89/541883)
Gmail Read Email	From, Subject, Content	2 (1.1KB) [**0.64%**]	730 (171.1KB)	**0.02%** (92/390206)
Ask Search	Search Terms	3 (1.3KB) [**0.31%**]	569 (416.2KB)	**0.01%** (40/218922)
Google Search	Search Terms	6 (0.92KB) [**0.26%**]	581 (352.9KB)	**0.2%** (437/206675)
Google+ Post	Post Content	5 (1.01KB) [**0.13%**]	1150 (750KB)	**0.005%** (48/947254)
NetFlix	Username, Password	1 (0.6KB) [**0.2%**]	419 (289.9KB)	**0.1%** (170/162853)
IMDb Search	Search Terms	1 (0.24KB) [**0.02%**]	1131 (1.1MB)	**0.1%** (422/323208)
Facebook Read Post	Post Content	1 (0.5KB) [**0.18%**]	730 (268KB)	**0.02%** (347/1216952)
Facebook Friend Search	Search Terms	9 (1.7KB) [**0.50%**]	959 (336.6KB)	**6.4%** (60305/937108)
eBay Item List	Item's Description	8 (1.49KB) [**0.62%**]	725(239.2KB)	**0.1%** (326/241901)
Amazon Login	Username, Password	2 (1.87KB) [**1.28%**]	240 (146KB)	**0.5%** (1629/321275)
Amazon Search	Search Terms	9 (1.77KB) [**0.24%**]	485 (732.7KB)	**0.1%** (544/433110)
MSN Search	Search Terms	5 (1.4KB) [**0.08%**]	1185 (1.7MB)	**0.5%** (2096/435124)
StackOverflow	Post Content	5 (2.36KB) [**1.18%**]	429 (198.9KB)	**0.2%** (830/406328)
Twitter Login	Username, Password	3 (0.61KB) [**0.27%**]	472 (223.6KB)	**0.004%** (27/622874)
Wikipedia Search	Search Terms	6 (1.9KB) [**0.23%**]	438 (794.1KB)	**0.45%** (382/84818)
Babylon Purchase	Email Address, Credit Card Info	3 (1.17KB) [**0.70%**])	485 (165KB)	**0.6%** (3232/515586)
Local Bank Login	Username, Password, PassKey	3 (1.49KB) [**1.01%**]	250 (146.3KB)	**~0%** (1/100037)
BankOfAmerica Login	Username, Password	2 (1.01 KB) [**0.23%**]	780 (430.4KB)	**~0%** (3/441534)

Table 3: Study of 20 Popular Web Applications. ***< 1% of web application code needs to run in the TEMI to access sensitive info; < 1% of all calls to JS data types trigger browser-TEMI interactions.***

be loaded in the same page, and each may contain its own JavaScript. If the listing is not protected, the script from one seller may deface or tamper with that from another seller. We also consider Babylon, since users can purchase items without creating an account with Babylon. Therefore, by tampering with the purchaser's email in the browser, an attacker can have a software license purchased by a legitimate user, but delivered to the attacker's email address. We mark the email address as a sensitive field on Babylon. We mark product search terms as sensitive for Amazon.

- *Gmail*. It is a rich webmail client used as our running example. We mark search terms and certain emails as sensitive.

We manually (with browser instrumentation) identify all functions that legitimately operate on sensitive information, including event handlers. Our analysis details are shown in Table 3. We assign the default policy $\kappa_0 \rightarrow \kappa_0$ to these functions, and all user keyboard inputs are marked as sensitive. We mark the Gmail spam warning message as requiring proof-of-impression. We also mark a post on Facebook as requiring proof-of-impression.

Results. We dynamically modify these web pages to encode sensitive contents into CRYPTONs using a proxy server. We find that these applications render correctly in our CRYPTON-compliant browser implementation, and the applications remain functional. This confirms our hypothesis that existing applications can be easily upgraded to using CRYPTON-KERNEL's functionality. Our test proxy decodes encrypted content returned from the browser and checks them against the expected values. It also verifies that it receives proof-of-impression tokens.

Developer effort. Table 3 demonstrates that the developer effort to enable this functionality is modest. On average, only **1%** of the total functions in the applications need to be included in CRYPTONs; this amounts to **2** KB of minified JavaScript per web page. The effort required to verify such code as CRYPTON functions is feasible, around **1-9** functions for each application. Figure 4 shows similar results that about **0-9** functions need to be verified for each signup page.

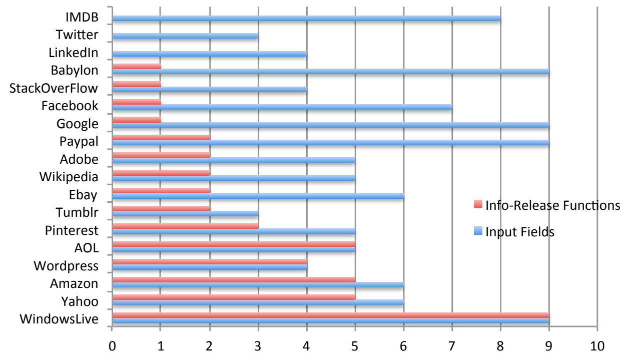

Figure 4: Number of User Input Fields and CRYPTON Functions Required in Signup Pages

4.2 Reduction in Attack Surface

The isolation and controlled operations provided by CRYPTONs significantly reduce the size of code that sensitive information is exposed to. We evaluate this reduction in both web application code as well as in browser code.

Exposure to untrusted application code reduces by 99%. As we show in Table 3, on average less than 1% of web application code actually needs access to sensitive information, and is thus run in CRYPTON functions. All the rest of web application code only has access to opaque objects with encrypted data. On the contrary, in current web browsers, all information is exposed to the entire web application. Any malicious or compromised JavaScript library can steal the information and leak it to external parties. By isolating sensitive information into CRYPTONs, the CRYPTON-KERNEL places **99%** of web application outside the TCB.

Exposure to browser vulnerabilities reduced by 92.5%. In existing web browsers, sensitive information being processed is largely accessible to browser code running in the same process. To evaluate how the CRYPTON-KERNEL reduces such over-exposure of sensitive information to browser vulnerabilities, we study how many historical browser security vulnerabilities are prevented from affecting the security guarantees provided by the CRYPTON-KERNEL. From our study with historical security vulnerabilities in Firefox reported separately [24], in total 333 vulnerabilities (92.5%) cannot be exploited to violating the security guarantees provided by the CRYPTON-KERNEL. The remaining vulnerabilities either reside in our TCB (7 of them), or compromise our assumptions (another 20 vulnerabilities). The vast reduction in the exposure to browser vulnerabilities verifies the effectiveness of the CRYPTON-KERNEL in information protection.

4.3 Performance

We run the CRYPTON-KERNEL over the 20 web applications listed in Table 3, with sensitive information transformed into CRYPTONs. The web sites remain responsive, without any *perceivable* slowdown. To further evaluate potential performance bottlenecks, we apply microbenchmarks to measure the performance overheads (averaged over 20 runs each) arising from encryption/decryption operations and browser-TEMI interactions when the TEMI executes CRYPTON functions.

Encryption/decryption overhead. Figure 5 shows the performance overhead of the CRYPTON-KERNEL compared to a vanilla WebKit-GTK browser. We use five CPU-intensive test pages from the Dromaeo benchmark. To vary the workload of encryption and decryption, we mark different portions of texts in the pages as sensitive, ranging from zero-sensitive pages (no sensitive text) to fully sensitive pages (all texts marked as sensitive). During these tests, the browser remains responsive and shows a maximum of 7.5% performance overhead. We also manually craft a test page to evaluate the performance of the CRYPTON-KERNEL under pathological scenarios. The performance overhead reaches 11.7% when all texts on the page are marked as sensitive, which requires heavy encryption/decryption operations. Considering such performance overhead is measured from an initial unoptimized prototype, it is reasonable.

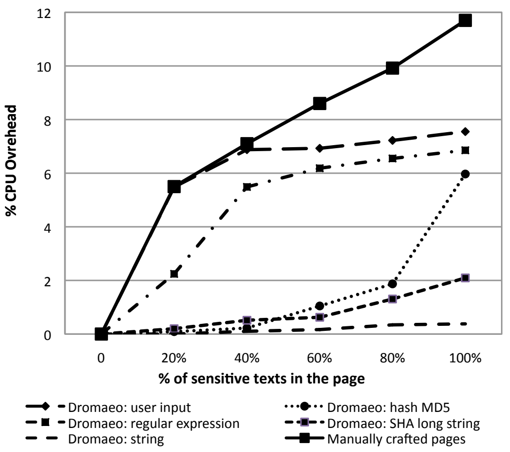

Figure 5: Performance Overhead of CRYPTON-KERNEL with Dromaeo Benchmarks & Manual Test Page, ***with varying portions of sensitive texts***

Overhead of browser-TEMI interactions. We run the CRYPTON-KERNEL over the 20 web sites listed in Table 3, and measure the number of IPC calls between the untrusted web browser and the TEMI. The last column in Table 3 shows that for most cases, less than 1% of all access to JavaScript data types requires crossing the browser-TEMI boundary. Given that the overhead of sending a message via Unix domain socket is around 5 microseconds, a web site with 1,000 IPC messages may only incur 0.005 second overhead.[8] Thus, the design of separating the execution of CRYPTON functions in the TEMI does not cause any performance bottleneck.

5. RELATED WORK

In this section, we discuss research related to information protection in web browsers or operating systems, and key differences with our work.

Enhancement of browser security mechanisms. There has been extensive research on enhancing security mechanisms of web browsers. One direction is to develop flexible and fine-grained access control on the web platform, so that untrusted JavaScript can be restricted to access only limited resources in a web page [40,54,74]. A recent work proposes a multi-layered architecture for enforcing mandatory access control policies, with a label-enforcing web browser [35]. As with other in-browser security mechanisms [47, 50], these solutions do not prevent compromised browser code from directly reading the data stored in the browser's memory.

Privilege separation & trusted paths. Privilege separation is a fundamental mechanism to enforce basic security principles [60]. Several research works have been proposed to facilitate [7] or automate [12] privilege separating legacy applications. Recently, improving separation of privileges in web browsers has become an active area of research [4, 5, 16, 21, 25, 33, 46, 63, 69]. These solutions isolate software components into partitions, thus reducing the potential damage to sensitive data from a compromised component. Nevertheless, they do not provide first-class abstractions to control access to sensitive data, in terms of which information can be disclosed to which entity. Further more, privilege separation does not provide sustained control on sensitive data. Once any information is disclosed to a partition, any code in the partition can further leak it to others. In contrast, our solution focuses on establishing several critical trusted paths in an untrusted web browser. Our proposal is orthogonal to the underlying mechanisms of privilege separation, and to our best knowledge, is the first in bringing data-centric trusted paths to the web. In our threat model, we trust the operating system beneath the web browser. Thus, our solution differs from other trusted path research on the x86 platform, where the operating system is untrusted, and additional sources of trust such as the hypervisor [75] or hardware root of trust [49] is required.

Information flow analysis. Information flow tracking has also been used for detecting and preventing information leakage in web applications [66]. However, without controlling the usage of sensitive information of client-side JavaScript, once disclosed, the information can be leaked by malicious scripts via self-exfiltration attacks [17]. Moreover, such solutions rely on the trust of browsers. On the contrary, our solution is based on a small TCB, and protects sensitive data against threats from all untrusted code or scripts in the browser.

Cryptographic techniques. Cryptographic techniques have long been protecting security in different systems. Lie et al. [45] pro-

[8]Outliers are Facebook Friend Search and WindowsLive Signup, with high browser-TEMI interactions. A closer look at them reveals that they are caused by a particular CRYPTON function in each of them, which processes both sensitive and non-sensitive information. We separate each function into two versions accordingly, and browser-TEMI interactions fall below 1% among all JavaScript data type accesses.

pose an abstract machine of execute-only memory (XOM) to prevent tampering of instructions stored on memory. Borders et al. propose Storage Capsules [10] to allow users to edit files on an untrusted computer without risking leakage of the file content. Although similar in concept, our solution protects data on the web, which are finer-grained, and computation over it is not packaged as applications, but scattered across different functions. This work addresses such challenges in protecting sensitive web content while being functionally compatible with existing web applications.

A recent work [19] further applies memory encryption to data confidentiality, similar to the concept of Data Capsules [48]. It allows unvetted programs to use sensitive data while enforcing policy to confine activities they can perform on the data. The high-level application-specific policy is translated into low-level tags enforced by the underlying hardware. One issue with this approach is: once the plaintext is disclosed to certain code, it is difficult to control information leakage any further, such as via implicit control flows. Our solution prevents it by limiting the plain-text only to CRYPTON functions, whose return values are also encrypted by default.

CleanOS [64] uses information flow tracking and encryption to protect mobile devices against threats to data when the mobile device is lost. Policy-sealed data [61] is a recent proposal on building trusted cloud services that protects data from unintended access. Their abstractions and cryptographic primitives (attribute-based encryption) are well suited for the cloud environment. However, unlike on the cloud, where environments can be summarized as configurations, in web browsers, there is no static context or environment. Code from multiple sources with various privileges are running in and molding the mixed browser environment. Therefore, we need more dynamic mechanisms to control data access and processing.

6. CONCLUSION

In this paper, we present a novel abstraction, CRYPTON, which protects sensitive web content and allows rich computation over it. Based on this data-centric abstraction, we propose a solution that integrates with the present web browser without trusting its code. Instead, the security guarantees provided by our solution are solely enforced by a small standalone engine, called CRYPTON-KERNEL, which interprets sensitive web content in CRYPTONs and allows trusted functions to securely compute over the sensitive data. Our large-scale evaluation demonstrates our solution can effectively protect sensitive web content in real-world web applications, with reasonable performance.

7. ACKNOWLEDGMENTS

We thank anonymous reviewers for their helpful feedback, and our shepherd Nicolas Christin for his insightful comments and suggestions on preparing the final version of the paper. This research is partially supported by research grant R-252-000-495-133 from Ministry of Education, Singapore. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the Ministry of Education, Singapore.

8. REFERENCES

[1] Summary of source code modification in case studies. `http://compsec.comp.nus.edu.sg/crypton/summary.pdf`.

[2] D. Akhawe, P. Saxena, and D. Song. Privilege separation in html5 applications. In *Proceedings of the 21st USENIX Security Symposium*, 2012.

[3] M. Balduzzi, M. Egele, E. Kirda, D. Balzarotti, and C. Kruegel. A solution for the automated detection of clickjacking attacks. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '10, 2010.

[4] A. Barth, A. P. Felt, P. Saxena, and A. Boodman. Protecting browsers from extension vulnerabilities. In *Proceedings of the 17th Annual Network and Distributed System Security Symposium*, NDSS '10, 2010.

[5] A. Barth, C. Jackson, C. Reis, and T. G. C. Team. The security architecture of the chromium browser. `http://seclab.stanford.edu/websec/chromium/chromium-security-architecture.pdf`.

[6] Bitbucket. `https://bitbucket.org/`.

[7] A. Bittau, P. Marchenko, M. Handley, and B. Karp. Wedge: splitting applications into reduced-privilege compartments. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '08, 2008.

[8] H. Bojinov, D. Sanchez, P. Reber, D. Boneh, and P. Lincoln. Neuroscience meets cryptography: designing crypto primitives secure against rubber hose attacks. In *Proceedings of the 21st USENIX Security Symposium*, 2012.

[9] J. Bonneau, C. Herley, P. C. v. Oorschot, and F. Stajano. The quest to replace passwords: A framework for comparative evaluation of web authentication schemes. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, 2012.

[10] K. Borders, E. V. Weele, B. Lau, and A. Prakash. Protecting confidential data on personal computers with storage capsules. In *Proceedings of the 18th USENIX Security Symposium*, 2009.

[11] D. Brumley and D. Boneh. Remote timing attacks are practical. In *Proceedings of the 12th USENIX Security Symposium*, 2003.

[12] D. Brumley and D. Song. Privtrans: automatically partitioning programs for privilege separation. In *Proceedings of the 13th USENIX Security Symposium*, 2004.

[13] CairoGraphics. cairo_glyph_t. `http://cairographics.org/manual/cairo-text.html#cairo-glyph-t`.

[14] N. Carlini, A. P. Felt, and D. Wagner. An evaluation of the google chrome extension security architecture. In *Proceedings of the 21st USENIX Security Symposium*, 2012.

[15] L. Cavallaro, P. Saxena, and R. Sekar. On the limits of information flow techniques for malware analysis and containment. In *Proceedings of the 5th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, DIMVA '08, 2008.

[16] E. Y. Chen, J. Bau, C. Reis, A. Barth, and C. Jackson. App isolation: get the security of multiple browsers with just one. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, 2011.

[17] E. Y. Chen, S. Gorbaty, A. Singhal, and C. Jackson. Self-exfiltration: The dangers of browser-enforced information flow control. In *Proceedings of the Workshop of Web 2.0 Security & Privacy 2012*, 2012.

[18] S. Chen, R. Wang, X. Wang, and K. Zhang. Side-channel leaks in web applications: A reality today, a challenge tomorrow. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, 2010.

[19] Y.-Y. Chen, P. A. Jamkhedkar, and R. B. Lee. A software-hardware architecture for self-protecting data. In

Proceedings of the 2012 ACM Conference on Computer and Communications Security, CCS '12, 2012.

[20] Y. Cheng and X. Ding. Virtualization based password protection against malware in untrusted operating systems. In *Proceedings of the 5th International Conference on Trust and Trustworthy Computing*, TRUST '12, 2012.

[21] R. S. Cox, S. D. Gribble, H. M. Levy, and J. G. Hansen. A safety-oriented platform for web applications. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, 2006.

[22] S. Crites, F. Hsu, and H. Chen. Omash: enabling secure web mashups via object abstractions. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, CCS '08, 2008.

[23] A. R. Developers. Ajaxim rpg. `http://ajaximrpg.sourceforge.net/`.

[24] X. Dong, H. Hong, Z. Liang, and P. Saxena. A quantitative evaluation of privilege separation in web browser designs. In *Proceedings of the 18th European Conference on Research in Computer Security*, ESORICS '13, 2013.

[25] X. Dong, M. Tran, Z. Liang, and X. Jiang. Adsentry: comprehensive and flexible confinement of javascript-based advertisements. In *Proceedings of the 27th Annual Computer Security Applications Conference*, ACSAC '11, 2011.

[26] DoubleVerify. Doubleverify uncovers ad fraud tied to copyright infringement sites, costing online advertisers $6.8 million per month. `http://www.doubleverify.com/resources/research/DV-Fraud-Lab-Report-2013-05/`, May 2013.

[27] J. Engler, C. Karlof, E. Shi, and D. Song. Is it too late for pake? In *Proceedings of Web 2.0 Security and Privacy Workshop 2009*, 2009.

[28] I. E. T. Force. Rfc 6101: The secure sockets layer (ssl) protocol version 3.0. `http://tools.ietf.org/html/rfc6101`, 2011.

[29] M. Foundation. Mozilla foundation security advisories. `http://www.mozilla.org/security/announce/`.

[30] W. Foundation. Wordpress. `http://wordpress.org/`.

[31] S. Garera, N. Provos, M. Chew, and A. D. Rubin. A framework for detection and measurement of phishing attacks. In *Proceedings of the 2007 ACM Workshop on Recurring Malcode*, WORM '07, 2007.

[32] Github. `https://github.com`.

[33] C. Grier, S. Tang, and S. King. Designing and implementing the op and op2 web browsers. *ACM Transactions on the Web*, 2011.

[34] M. Heiderich. Towards elimination of xss attacks with a trusted and capability controlled dom. `http://heideri.ch/thesis`.

[35] B. Hicks, S. Rueda, D. King, T. Moyer, J. Schiffman, Y. Sreenivasan, P. McDaniel, and T. Jaeger. An architecture for enforcing end-to-end access control over web applications. In *Proceedings of the 15th ACM Symposium on Access Control Models and Technologies*, SACMAT '10, 2010.

[36] L.-S. Huang, A. Moshchuk, H. J. Wang, S. Schechter, and C. Jackson. Clickjacking: attacks and defenses. In *Proceedings of the 21st USENIX Security Symposium*, 2012.

[37] Intel. Trusted compute pools with intel® trusted execution technology. `http://www.intel.com/txt`.

[38] Internet Engineering Task Force (IETF). Rfc 6454: The web origin concept. `http://www.ietf.org/rfc/rfc6454.txt`.

[39] C. Jackson, D. R. Simon, D. S. Tan, and A. Barth. An evaluation of extended validation and picture-in-picture phishing attacks. In *Proceedings of the 11th International Conference on Financial Cryptography and 1st International Conference on Usable Security*, FC'07/USEC'07, 2007.

[40] K. Jayaraman, W. Du, B. Rajagopalan, and S. J. Chapin. Escudo: A fine-grained protection model for web browsers. In *Proceedings of the 30th International Conference on Distributed Computing Systems*, ICDCS '10, 2010.

[41] N. Kobeissi, A. Breault, and E. Gill. Cryptocat. `https://crypto.cat/`.

[42] B. Köpf and M. Dürmuth. A provably secure and efficient countermeasure against timing attacks. In *Proceedings of the 2009 22nd IEEE Computer Security Foundations Symposium*, CSF '09, 2009.

[43] LastPass. Lastpass password manager. `https://lastpass.com/`.

[44] A. Libonati, J. M. McCune, and M. K. Reiter. Usability testing a malware-resistant input mechanism. In *Proceedings of the 18th Annual Network and Distributed System Security Symposium*, NDSS '11, 2011.

[45] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz. Architectural support for copy and tamper resistant software. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS-IX, 2000.

[46] M. T. Louw, K. T. Ganesh, and V. N. Venkatakrishnan. Adjail: practical enforcement of confidentiality and integrity policies on web advertisements. In *Proceedings of the 19th USENIX Security Symposium*, 2010.

[47] T. Luo and W. Du. Contego: capability-based access control for web browsers. In *Proceedings of the 4th International Conference on Trust and Trustworthy Computing*, TRUST '11, 2011.

[48] P. Maniatis, D. Akhawe, K. Fall, E. Shi, S. McCamant, and D. Song. Do you know where your data are?: secure data capsules for deployable data protection. In *Proceedings of the 13th Workshop on Hot Topics in Operating Systems*, HotOS-XIII, 2011.

[49] J. M. McCune, A. Perrig, and M. K. Reiter. Safe passage for passwords and other sensitive data. In *Proceedings of the 16th Annual Network and Distributed System Security Symposium*, NDSS '09, 2009.

[50] L. A. Meyerovich and B. Livshits. Conscript: Specifying and enforcing fine-grained security policies for javascript in the browser. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, 2010.

[51] Mozilla. Firefox os. `https://developer.mozilla.org/en-US/docs/Mozilla/Firefox_OS`.

[52] C. Nie. Dynamic root of trust in trusted computing. `http://www.tml.tkk.fi/Publications/C/25/papers/Nie_final.pdf`.

[53] D. of Defense Standard. Department of defense trusted computer system evaluation criteria, 5200.28-std, 1985.

[54] K. Patil, X. Dong, X. Li, Z. Liang, and X. Jiang. Towards fine-grained access control in javascript contexts. In *Proceedings of the 31st International Conference on Distributed Computing Systems*, ICDCS '11, 2011.

[55] T. C. Projects. Chromium os. `http://www.chromium.org/chromium-os`.

[56] T. C. Projects. Linuxsandboxing. `https://code.google.com/p/chromium/wiki/LinuxSandboxing#The_seccomp-bpf_sandbox`.

[57] T. C. Projects. Sandbox. `http://www.chromium.org/developers/design-documents/sandbox`.

[58] F. Roesner, T. Kohno, A. Moshchuk, B. Parno, H. J. Wang, and C. Cowan. User-driven access control: Rethinking permission granting in modern operating systems. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, 2012.

[59] C. Rohlf and Y. Ivnitskiy. Attacking clientside jit compilers. `http://www.matasano.com/research/Attacking_Clientside_JIT_Compilers_Paper.pdf`.

[60] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. In *Proceedings of the IEEE*, 1975.

[61] N. Santos, R. Rodrigues, K. P. Gummadi, and S. Saroiu. Policy-sealed data: A new abstraction for building trusted cloud services. In *Proceedings of the 21st USENIX Security Symposium*, 2012.

[62] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A symbolic execution framework for javascript. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, 2010.

[63] S. Tang, H. Mai, and S. T. King. Trust and protection in the illinois browser operating system. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '10, 2010.

[64] Y. Tang, P. Ames, S. Bhamidipati, A. Bijlani, R. Geambasu, and N. Sarda. Cleanos: limiting mobile data exposure with idle eviction. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '12, 2012.

[65] T. R. Team. Roundcube. `http://www.roundcube.net/`.

[66] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Cross-site scripting prevention with dynamic data tainting and static analysis. In *Proceedings of the 14th Annual Network and Distributed System Security Symposium*, NDSS '07, 2007.

[67] W3C. Content security policy 1.0. `http://www.w3.org/TR/CSP/`.

[68] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, SOSP '93, 1993.

[69] H. J. Wang, C. Grier, A. Moshchuk, S. T. King, P. Choudhury, and H. Venter. The multi-principal os construction of the gazelle web browser. In *Proceedings of the 18th USENIX Security Symposium*, 2009.

[70] D. Wendlandt, D. G. Andersen, and A. Perrig. Perspectives: improving ssh-style host authentication with multi-path probing. In *USENIX 2008 Annual Technical Conference*, ATC '08, 2008.

[71] Z. E. Ye and S. Smith. Trusted paths for browsers. In *Proceedings of the 11th USENIX Security Symposium*, 2002.

[72] C. Yue and H. Wang. Anti-phishing in offense and defense. In *Proceedings of the 2008 Annual Computer Security Applications Conference*, ACSAC '08, 2008.

[73] Y. Zhang, J. I. Hong, and L. F. Cranor. Cantina: a content-based approach to detecting phishing web sites. In *Proceedings of the 16th International Conference on World Wide Web*, WWW '07, 2007.

[74] Y. Zhou and D. Evans. Protecting private web content from embedded scripts. In *Proceedings of the 16th European Conference on Research in Computer Security*, ESORICS '11, 2011.

[75] Z. Zhou, V. D. Gligor, J. Newsome, and J. M. McCune. Building verifiable trusted path on commodity x86 computers. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, 2012.