Starlight: a clean-slate unified security architecture for the web

Abstract

Starlight is a new security architecture for the web that provides policies that can be enforced uniformly across a wide range of subsystems. Leveraging decentralized information flow control (DIFC), Starlight allows data-centered policies that directly capture higher-level user requirements, such as privacy, and can be enforced across both clients and servers. A key advancement over previous DIFC systems is that Starlight infers policies and automatically sets labels and privileges based on user interactions. To evaluate Starlight, we built a number of web applications that take advantage of its security and functionality including audio chat, search, a VOIP client, a peer-to-peer video player, an editor and networked Quake.

1 Introduction

The web is a prominent platform for deploying distributed, large-scale applications. A consequence of its ubiquity is the need to address many security and privacy issues. Despite this, most of the security mechanisms in core web components, including web browsers and web servers, were designed without analytical foundations [14]. In the case of web browsers, different technologies such as HTML5, Javascript and plugins address security policies separately, assuming different attack models [39].

For instance, Java applets are executed according to an "all-or-nothing" security model. A signed applet has the power of a stand-alone program: it can connect to arbitrary third-party servers, access the local filesystem, call native code, etc. On the other hand, an unsigned applet is constrained to a restrictive sandbox (e.g., it cannot access the filesystem). Similarly, Flash applications, like unsigned applets, are sandboxed, but have the ability to access media devices such as microphones when given the user's permission [12]. Regardless of these sandbox mechanisms the user is provided with no real security guarantees: they cannot be sure that data sent to the web-server is protected according to any sensible policy.

In effect, each piece of web technology—Javascript, Java, Flash, browser extensions, and especially server-side frameworks—introduces its own, ad hoc security mechanism. Worse yet, these mechanisms do not directly capture higher-level security properties end-users are concerned with, such as data privacy. For example, the same origin policy might partially address some privacy concerns over data within the browser, but offers little help once data leaves the client. Because the web lacks a coherent security architecture, every new feature requires a re-think of the enforcement mechanisms. Arguments for the correctness and adequacy of these mechanisms are entirely *subjective*. For instance, in the case of HTML5 websockets, Firefox [27] decided to disable them after vulnerabilities were revealed, but other browser vendors felt it wasn't serious enough to warrant this action.

This paper presents Starlight, a clean-slate security architecture for the web. Starlight provides a unified security policy that can be used across technologies, browsers and servers. It allows speaking *objectively* about the security of an individual feature, without the need to consider every interaction it may have with any other present or future subsystem. In Starlight, policies center around protecting *data* rather than restricting particular *actions*. Such data-centric policies can unify security across different abstractions and directly capture many end-user security concerns. Starlight's approach is based on decentralized information flow control (DIFC). A key contribution of this work to the existing body of DIFC work is the ability to set policy implicitly based on existing user interactions with the application.

Starlight's policies make use of a global namespace of *principals*, which include end-users, web application authors, and hosting providers. Starlight associates a *label* with every piece of data encoding who may modify

the data, who may download and further disseminate the data, and who may observe but not further disseminate the data. Therefore, in addition to discretionary access control, Starlight introduces the ability to specify mandatory access control to the web.

Starlight subsystems label all data and enforce the restrictions encoded in labels. Principals can issue certificates stating that they trust other principals to handle their data. Starlight defines a network protocol through which machines can exchange labeled data, provided the endpoints have certificate chains sufficient to satisfy the requirements of the labels.

Starlight also provides a facility for setting labels and privileges based on user interface actions. It provides a small number of *privileged widgets* that infer user intent from UI interactions and appropriately bestow permissions on software. For example, in the case of audio conferencing, selecting which other users to call implicitly configures the microphone handler to label its output for exactly those users.

We implemented Starlight in both a web server and Google's Native Client (NaCl) browser sandbox. We have also adapted BFlow [46] to support Starlight at a frame granularity in JavaScript. These represent an admittedly small subset of the widely used web technologies, but demonstrate the practicality and benefits of unifying web security across varied subsystems. The Native Client implementation, in particular, offers not just security but increased functionality as mobile code gains access to previously prohibited devices and storage, subject to Starlight's labels. Our audio conferencing application is a good example of a new class of untrusted web applications requiring no special privileges beyond those implicitly offered by the user through the user interface.

Starlight's contributions include a DIFC-based security architecture for the web, an implementation in three different subsystems, a mechanism for inferring security policies from user actions, and, finally, an enhanced API to provide web applications with greater access to storage, networking, and devices. In part because Starlight's design prioritized uniform security over backwards compatibility, we realize our own implementations are unlikely to be adopted by browser vendors. Nonetheless, by demonstrating what is possible, we hope our work can inform the web security discussion and influence the security of still evolving newer technologies such as Native Client.

2 Motivating example

As a driving example, we consider the case of building a browser-based voice over IP (VoIP) application. Figure 1 illustrates the different components of this application in the context of today's web-architecture. The setup

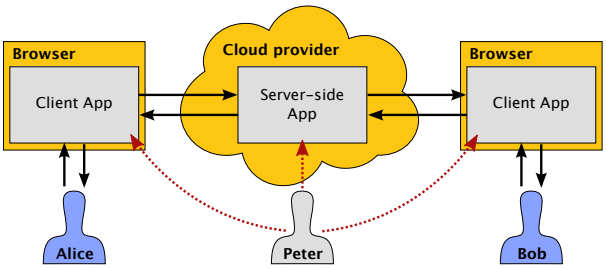

Figure 1: Existing framework of a web-based VoIP application. Alice and Bob use the VoIP app, authored by Peter. In the setup, the users must trust the whole application stack (shaded), including the app author.

consists of three parties [40]: the end-users, the application author, and cloud provider (e.g., Amazon EC2); to launch an application, users visit the website that is provided by the application author and hosted by the cloud provider.

In existing applications, such as Google Chat [6] or Twilio Client [11], users initiate calls through an HTML/JavaScript interface provided by the application author (who may also be the cloud provider, in this case). However, since neither JavaScript nor HTML have access to the user's microphone, these applications rely on browser extensions (Google Chat), or plugins, such as Flash (Twilio). Consequently, the security implications of using these applications are complex. In the case of a browser extensions, the user gives the extension access equivalent to a standalone program. Flash, on the other hand, prompts the user to give the website access to the microphone device. In both cases, once the consent has been granted, the user has no guarantees about (and, in fact, has no way of even specifying) where the captured voice will end up. Moreover, in most cases, user intentions are clear—only the end-user participants in the call should receive the audio stream—yet, lacks a means of tracking and enforcing correct information propagation.

Figure 1, highlights the core components that must be trusted when using an existing Flash-based VoIP application. Specifically, in addition to trusting the browser and the Flash implementation, the user must also trust:

1. the HTML/JavaScript client code to initiate a call to the correct recipient,
2. the Flash application to communicate audio only to the VoIP server,
3. the VoIP server not to leak the audio stream to unintended recipients or to store it to disk,
4. and the cloud platform hosting the application.

Using the existing framework, users must essentially trust the entire application stack, despite being limited to

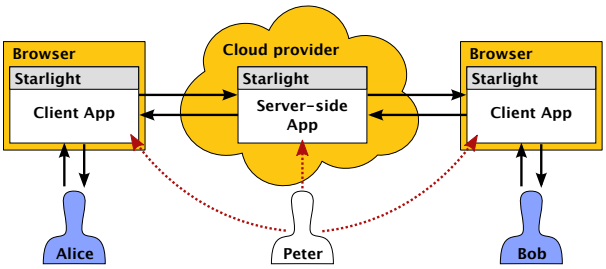

Figure 2: Proposed framework of a web-based VoIP application. Alice and Bob use the VoIP app, authored by Peter. In our setup, the users need not place any trust in the app components.

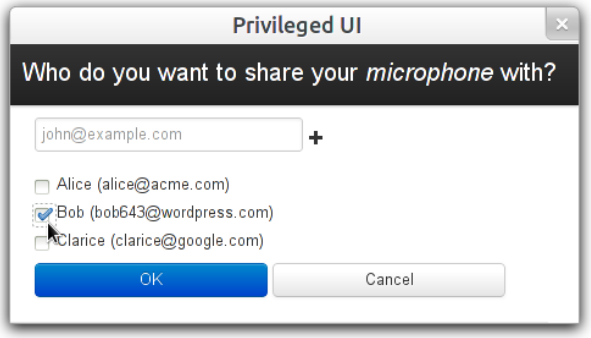

Figure 3: A user interface example that captures the user's intent and implicitly sets the security policy.

expressing very coarse-grained intentions (in this case, whether to allow a part of the stack to access the microphone). The goal of Starlight instead is shown in Figure 2, namely eliminating the trust placed in components (1-3): application code, both in the browser and on the server need not be trusted to enforce security. Section 7.1 details how this secure audio chat is realized in Starlight.

3 Starlight Overview

Given our goal to eliminate trust in application code, we define a threat model and present a high-level system architecture that allows us to safely transport information across the network, like the microphone audio as described in our motivating example.

3.1 Threat model

Adversary: Starlight assumes the *web attacker* [14] threat model where an attacker controls both the client-side and server-side application components.

Assumptions: On the client-side, we assume that Starlight's UI integration grants only the privileges appropriate for certain user actions and is not otherwise exploitable.

On the server-side, we assume that cloud providers run Starlight. Previous work on remote attestation such the trusted platform module [36], and CloudVisor [50] may obviate this assumption.

For both the client-side and server-side, it is necessary for trusted code to be implemented correctly. Specifically, it must not be possible for applications to break out of their sandbox and IFC labels must be properly enforced.

Security guarantees: Starlight guarantees that confidentiality and integrity of user data are preserved, end-to-end, according to user-specified policies. Starlight, associates policies with data and enforces the requirement that applications respect these policies on both the client and the server. Referring back to the VoIP example, the client application cannot record microphone information and exfiltrate it to an arbitrary server, store the information on the server-side, or exfiltrate it from the server to a party other than the user-specified destination.

Without explicit trusted user actions, Starlight applications are *harmless*. Starlight guarantees that privileges can only be granted by privileged UI widgets and, like capabilities, cannot be forged or treated as data. It is only by *exercising* these privileges that sensitive user data can be downgraded (declassified), or modified.

Limitations: Starlight applications gain privileges as a direct result of user actions. Consequently, social-engineering and phishing attacks can still be carried out. For instance, a malicious application may instruct the user to input their password in place of a file name, downgrade information to be uploaded to a malicious server, or even download an executable. Such attacks are outside the scope of this work, and we do not address them.

We also do not directly address covert channels in this work. However, Starlight's support for discretionary access control using clearance mitigates these kinds of attacks by preventing applications from accessing secret data.

3.2 System architecture

Starlight requires the following components to implement information flow control end-to-end with policies derived from user actions:

User action model. Starlight sets security policies based on user actions performed in the application. Figure 3 shows how an audio chat application

might ask the user whom to call. The UI widget is privileged in that the Starlight runtime can monitor the user's selection and set a security policy accordingly. In this example it would let the microphone audio stream flow only to the selected user, Bob.

Application sandbox. Starlight depends on sandboxing to restrict an application's access to secret information sources and to enforce IFC constraints. A sandbox must be used in each location that runs application software, both on the server and in the browser as shown in Figure 2.

Global namespace and network protocol. Starlight must support the distributed nature of web applications. This requires a networking protocol that allows transmission of labeled data. It also requires a global namespace to uniquely identify principals on the Internet. For example there must be a unique way to identify "Bob" on the Internet to send a microphone audio stream to him.

4 Information flow control

In a traditional access control system, permissions are associated with abstractions such as files. For instance, a file's ACL might state that user A or B can read its contents. The ACL restricts who can read the file, but once the file is read, the ACL becomes irrelevant; the process can write the contents to a publicly readable file or transmit it over the network.

Information flow control (IFC), by contrast, associates policies directly with data, regardless of the abstractions used to contain the data. Conceptually, every bit has a *label* stating who may observe or modify it. If a process reads a file labeled L, the memory into which the contents is read must also be labeled L (or something more restrictive than L). If the contents is subsequently written to a new file, the new file must also have a label as restrictive as L. When data are combined, their labels are also combined to reflect all restrictions on all sources. For instance, combining contents readable by A or B with contents readable by B or C results in contents readable by only B, as only B could read both sources.

Decentralized information flow control (DIFC) relaxes IFC by introducing *privileges* that allow software to bypass certain label restrictions. It is decentralized because different privileges allow different software to bypass different restrictions. For example, software with user A's privileges might be allowed to take data labeled for A or B and make it public, while it could not do the same given data labeled for B or C.

Starlight adopts a type of label called DC labels [22]. DC labels express privileges in terms of *principals*, which in Starlight include public keys and URL origins. (For example, a network endpoint has the privileges of a public key when it can produce an appropriate signature and certificate chain.) A DC label specifies which sets of principals can export (i.e., disseminate) data and which can modify it. These sets are expressed as boolean formulas over principals, which we typically write S ("secrecy") for the export sets and I ("integrity") for the modify sets. Hence, a DC label is written $\langle S, I\rangle$. We note that each label has a unique representation in conjunctive normal form. By analogy with military security, we sometimes refer to a disjunctive clause as a *category*, making each label component a conjunction of categories.

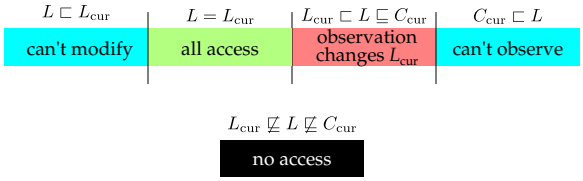

Figure 4: Access permissions to data labeled L. The red region requires mandatory access control.

An important question is when a label L_2 is as restrictive as another L_1, a partial order we write $L_1 \sqsubseteq L_2$ (pronounced "L_1 *can flow to* L_2"). For DC labels, $\sqsubseteq$ is defined using logical implication on the component formulas. If $L_1 = \langle S_1, I_1\rangle$ and $L_2 = \langle S_2, I_2\rangle$, then $L_1 \sqsubseteq L2$ iff $S_2 \implies S_1$ and $I_1 \implies I_2$. In other words, anyone who can export data labeled L_2 can also export data labeled L_1, and anyone who can modify data labeled L_1 can also modify data labeled L_2. Without privilege, data labeled L_2 can depend on data labeled L_1 only when $L_1 \sqsubseteq L_2$.

Executing code is always associated with two labels, the current label, L_{cur}, and current clearance, C_{cur}. Code may change the value of L_{cur} to any value L_{new} in the range $L_{\text{cur}} \sqsubseteq L_{\text{new}} \sqsubseteq C_{\text{cur}}$. In other words, the value of L_{cur} is monotonically increasing and bounded by C_{cur}. L_{cur} and C_{cur} precisely determine the current level of access allowed to data. Data labeled L can be observed only when $L \sqsubseteq L_{\text{cur}}$ and modified only when $L_{\text{cur}} \sqsubseteq L$. Figure 4 summarizes this relationship.

Executing code may also have access to privileges, which like label components are expressed as boolean formulas over principals. Typical privileges might be the conjunction principals who have issued certificates affirming trust in a particular server. Exercising privileges p for an operation causes Starlight to enforce an alternate, more permissive can-flow-to relation $\sqsubseteq_p$, defined as $\langle S_1, I_1\rangle \sqsubseteq_p \langle S_2, I_2\rangle$ iff $S_2 \wedge p \implies S_1$ and $I_1 \wedge p \implies I_2$. For example $\langle [A \vee B], []\rangle \sqsubseteq_A \langle [], []\rangle$, since A is allowed to export data with $S = [A \vee B]$. Note the empty boolean formula $[]$ is an abbreviation for **True**.

Starlight assigns the public network an empty label,

$L_\emptyset = \langle [], [] \rangle$. If code could always talk to the network, then code would always have $L_{\text{cur}} = L_\emptyset$ and IFC would boil down to verifying $p \implies S$ on reads and $p \implies I$ on writes of data labeled $\langle S, I \rangle$. A more interesting case arises when code lacking export permission nonetheless wishes to read data with a non-trivial label. If the target label is below C_{cur}, the code can still read the data by raising L_{cur}, but in the process gives up the ability to write to the network. This ability to trade write for read permissions is a form of mandatory access control differentiating Starlight from most existing web technologies.

Most mobile code does not have direct access to the network in Starlight. Nonetheless, code often needs to communicate across the network to other code running with a comparable label on a different machine. To allow labeled network communication, Starlight provides an exporter service that effectively exchanges labels for encryption. Exporters have the privileges required to send and receive labeled data from the network, but do so only under two conditions: First, over the network exporters only communicate with other exporters who cryptographically prove they have sufficient privileges to send or receive the labeled messages in question. Second, exporters ensure that on the local side, plaintext messages are appropriately labeled as requested by or asserted to the remote exporter. Exporters were originally introduced by [49]; we refer readers to that work for a more detailed explanation.

5 System design

Starlight provides built-in privileged UI widgets, which are used to endorse certain user actions. In addition, policy modules, which are shipped with the browser or manually installed by users, provide a means of granting applications privileges based on those endorsements.

Starlight enforces IFC using NaCl and BFlow [46] on the client-side, and the Haskell Labeled IO library [41] on the server-side. Finally, our client-server network communication respects labels by using a DStar-like protocol that leverages WebFinger for globally naming principals. We now present these core design components in detail.

5.1 User action model

The user action model is responsible for inferring policy from user actions. It consists of a fixed-set of privileged UI widgets and a collection of policy modules. We decouple the user interface from the privilege-granting code. This allows the flexibility to develop policy modules independently of UI code. Moreover, it reduces the attack surface of policy decision-making code. Specifically, the user action model contains:

1. *Privileged UI* widgets are used to perform common actions such as accessing a microphone or initiating a network connection to a mail server, the action of which produces an endorsement.

2. *Policy modules* are small, trusted applications that translate endorsements to application privileges.

The privileged UI consists of a set of widgets which applications can launch in order to interact with users, and ultimately, convey the required privileges necessary carry out the user's intent. Starlight provides the following widgets: textbox, button, hyperlink, icon, file chooser, address book (the endorsement represents a contact selection), and input device, including microphone and webcam.

User action endorsements provided by the privileged UI are specified in the form of integrity categories as added to the application process' label. For example, if the user types in "mail.google.com" in a textbox widget titled "Enter SMTP server address", the process would obtain the integrity category "textbox://Enter SMTP server address/mail.google.com". Since applications do not own the"textbox://" principal, or any other principal used by the privileged UI widgets, applications cannot forge such integrity categories.

As mentioned, applications can invoke policy modules with user action endorsements. The policy modules inspect endorsements and may grant the application privileges. For example, a policy module invoked with the "textbox://Enter SMTP server address/mail.google.com" endorsement may grant the application the privilege to connect to mail.google.com on TCP port 25. In addition to granting the application this privilege (e.g., by adding integrity categories to the application's label), the policy module may additionally remove certain protection mechanisms (e.g., DoS rate limiter). We note that the policy modules will likely be designed defensively to accept endorsements. For example, the SMTP policy module will expect endorsements prefixed with "textbox://Enter SMTP server address". This ensures that if the user is asked to input a hostname by a deceiving textbox caption (e.g., "Enter smtp.evil.com here"), the application cannot use the received endorsement ("textbox://Enter smtp.evil.com here/evil.com") to gain privileges from the SMTP policy module.

One challenge is that an untrusted application could try to impersonate a privileged UI widget. Although this could not be used to escalate privileges, the attacker may still be able to conduct a phishing attack and coerce the user into divulging secret information. One possible mitigation strategy might be to make trusted UI widgets visually distinct, for example by darkening the screen around it, including regions of the framebuffer not controlled by the application.

5.2 Application sandbox

Starlight uses sandboxing in the browser and on the server-end to restrict applications to APIs that respect IFC. For example, our sandbox guarantees that after an application has read a sensitive user-data, it cannot connect to arbitrary remote hosts.

5.2.1 Client-side sandbox

All system calls (to Starlight) and IO communications are subject to information flow control restrictions. We track and enforce IFC using DC labels. Every application owns a set of principals:

- The HTTP origin of the application
- Application-specific public key
- Vendor-specific public key

Ownership of the HTTP origin principal allows applications served from the same domain to share information with each other. Specifically, to share information with another application from the same domain, an application can simply label data such that every secrecy clause of the DC label contains a disjunction of the origin principal. In general, the disjunction property of DC labels provides a means for sharing information between applications, and allows for the implementation of very rich applications, such as mashups.

In Starlight, every client-side web application is signed with an application and vendor key. By labeling data with the principal of the application key hash the application can safeguard sensitive information. Conversely, labeling data with the vendor-specific key hash allows applications written by the same vendor to share information, in a manner that is more fine-grained than using the HTTP origin.

Enriching the web with APIs: We extend client-side web functionality by providing secure file access and arbitrarily complex network communication. Our filesystem is implemented as a file store, in a directory dedicated to the web browser, where a label is associated with every file. Using the labeled filesystem, authors can build complex web applications that can share files on the client-side, permitting IFC restrictions. (We note that since the file store is on the user's machine, the user can specify per-application quotas.)

As with filesystem access, arbitrary network communication is allowed as long as it obeys IFC; this is ensured by requiring that all communication be handled by a local DStar-like exporter [49]. A consequence of providing arbitrary network access is that Starlight needs to protect against malicious applications that leverage user bandwidth to harass remote servers (e.g., by sending spam or participating in DDoS attacks). To enforce this, we require that an application obtain an endorsement via a privileged UI, certifying that the user wishes to establish a specific network connection or listen on a socket.

To allow for the functionality provided by cross-origin resource sharing, a client application may connect to a remote server without a user endorsement if the remote host explicitly affirms the connection. This is determined by the Starlight runtime via a simple UDP protocol on a port 1392. The runtime sends a hash of the application and a connection request (protocol and port number) to the target server, allowing the application to proceed with the connection only if the that remote server is in agreement. Such requests are rate-limited to make DoS attacks via the authorization protocol ineffective.

5.2.2 Server-side sandbox

The server-side sandbox requires a subset of the client-side functionality. Specifically, on the server-end Starlight need not address issues related to DoS attacks, API extensions, or user interaction. Therefore, existing dynamic IFC systems, such as Flume [32], HiStar [48], and the Haskell Labeled IO library [41], can be used to sandbox server applications. In Section 6.1 we describe our server-side implementation.

5.3 Global namespace & network protocol

Since policies, in the form of labels, must be transported across the network between Starlight's client-side and sever-side components we rely on a protocol, similar to DStar [49]. We require that the protocol not leak information in establishing a connection with a remote entity, verify the trustworthiness of a remote application, encrypt messages as to preserve confidentiality, and encode labels in a network-meaningful way.

The latter requirement plays a crucial role in Starlight. To encode labels in a network-meaningful way, Starlight requires a mechanism for globally naming principals (i.e, users) on the Internet. This is important because, for example, a user using a UI widget to specify that her audio stream should be shared with her friend "Bob" is a local designation of "Bob"—we require a unique, global way to name "Bob". To this end, we rely on WebFinger, which allows for associating arbitrary data with unique names, which themselves are akin to e-mail addresses. In Starlight, we follow DStar's convention that each principal owns a public key, which corresponds to the principal's identity. A key difference however is that we cannot expect lay web users to refer to each other using public keys. This is why we need a user friendly naming scheme

like E-mail addresses and we use WebFinger to learn the corresponding public keys.

WebFinger provides a means for our privileged UI widgets to refer to principal public keys in a friendly way. Specifically, in the VoIP example the application may refer to 'Bob' using his WebFinger name 'bob643@wordpress.com'.

Users store their public key and exporter information for each application with their WebFinger provider. (Much as users trust their e-mail providers, we assume that users trusts their WebFinger providers.) The information stored with the WebFinger provider includes the user's public key and, for each application the user has installed, a URL with access information for the exporter and the exporter's public key. However, these URLs do not point directly to an exporter. Rather, they return the IP and port number used to connect the exporter. This level of indirection allows providers to offer load balancing, by, for example, returning different IPs and ports.

Like users, exporters are authenticated using public key. Finally, we note that all information of a user's WebFinger entry is signed with their public key, therefore the WebFinger server need not be trusted beyond the first lookup (assuming the public key is cached locally).

6 Implementation

We implemented Starlight for Firefox to allow BFlow integration. We use an older (NPAPI) version of NaCl since the latest (Peppr2) version only supports Chrome. Our implementation consists of a 6,341 line patch to Native Client and a 143 line patch to BFlow. Our Haskell application server and extensions to LIO are 1,235 lines.

We added APIs for setting and retrieving labels, registering and invoking policy modules, and adding trusted UI widgets. Our trusted UI widgets are implemented as a GTK popup running in a separate process which Javascript and CSS cannot affect, thus ensuring they are tamper-proof and isolated. A cleaner implementation would modify the browser's rendering engine to display the widgets inline within the page, while still making them tamper proof.

We also added POSIX calls for filesystem and networking support, like `open`, `read`, `close`, `socket`, `bind`, *etc.* The filesystem's root is in a private directory reserved for NaCl applications only. We maintain two separate directory trees in the filesystem: one where files with the same label are stored in the same directory, and another "shadow" filesystem where files are symbolic links but the directory structure is user-defined. This organization allows us to store label information without while preserving a root directory that looks, to the user, like a traditional filesystem, without modifying the OS filesystem. Audio is supported by exposing a `/dev/dsp`-like file. Labeled networking uses tcpcrypt [15] for encryption and we wrote our own exporter and WebFinger implementations.

All the I/O operations we added to NaCl required adding label checks. We also had to ensure that the APIs we offer did not leak information—for example, two processes can collude via the `bind` system call when trying to attach to a port and convey information if the *bind* call succeeds or fails. NaCl uses IMC for its IPC and NPAPI to communicate with Javascript so we also added label checks to both, and carry over label information when speaking to Javascript.

BFlow does not support disjunctive labels, so we need to translate disjunctive labels into traditional ones. To do this we hash the string representation of each disjunctive category in Starlight and use that as a BFlow tag. To convert from BFlow tags to disjunctive labels, we use a category with no disjunction and use the string representation of the BFlow tag. This way tags flowing from BFlow servers to NaCl and back are preserved and remain compatible end-to-end. Prior to each NPAPI call, the Starlight runtime invokes BFlow's reference monitor (via a socket) to taint the BFlow protection zones with any secrecy the Starlight module holds, and taints the Starlight module with any BFlow tags the protection zones hold. Our changes to BFlow are minimal since NaCl deals with integration logic, such as label conversion.

6.1 Haskell application server

Although the NaCl-based implementation can directly be used to implement server-side components, we use a language-level IFC system to get fine-grained labeling.

We built an application server platform using Labeled IO [41], a Haskell IFC library. Unlike NaCl, where the labeling granularity is at the process-level, LIO provides a means for enforcing IFC at a very fine-granularity. Namely, LIO provides a means of associating a label with a value. For example, a value of type `DCLabeled Integer` is a DC labeled integer. As in NaCl, observing sensitive data raises the current label, precluding the thread from writing to public entities.

An especially attractive property of the LIO library when implementing web-server applications, is the ability to execute computations that operate on differently labeled (and potentially secret) data without raising the current label. To achieve this, LIO provides a function named `toLabeled`, which is a generalization of constrained buffers [40]. With `toLabeled`, applications can manipulate user-sensitive data, without having to raise the current label (but also without having the ability to observe the computation result). The result of a `toLabeled` computation is a labeled value, which may

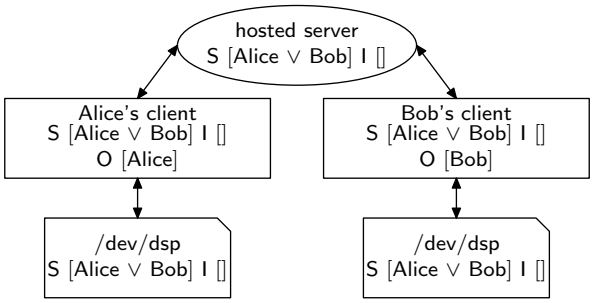

Figure 5: Labeling in the audio conferencing system. S denotes secrecy, I integrity, and O ownership.

be unlabeled at later point (it is at this point where the current label is raised). Of course, it is possible for an application to never unlabel such values, thus remaining "untainted", and simply forward the labeled values to end-users that have the privilege to observe them.

We extend LIO in a number of ways to allow execution of untrusted server-side applications. Specifically, we add a covert-channel safe subset of Haskell lightweight threads, support for labeled mutable locations, `MVar` [29], labeled (unbounded FIFO) channels, and a trusted exporter which exposes a safe, untrusted API to server-side applications.

7 Applications

We built a number of applications for Starlight, including the audio chat service used as an example in Section 2, a desktop search tool that searches both the web and local files, a SIP phone, a peer-to-peer video player, a text editor, and extensions to NaCl's Quake video game demo to support networked games and saving of game state. To port existing UNIX console applications we developed a terminal for Starlight. We ported SDL terminal [10] and Freetype [5] to create an xterm-like environment and ported `ash` to it for a shell. Both our text editor and desktop search use this terminal.

7.1 Audio chat

Our audio chat application send user audio streams to a server; the server then mixes the audio and sends it back to the users. This demonstrates a basic conference call system within Starlight. It relies on Starlight's end-to-end nature, utilizing both client-side and server-side components and shows how a client-specified security policy is enforced across a distributed web application. It also demonstrates the use of disjunctive labels in supporting collaboration between multiple users.

The Starlight NaCl extension provides the extra functionality needed in a web browser to program an audio application and set security policies. The remote user's public key (principal name) and the server to contact for a call is determined via WebFinger. Next, the exporter is used to ensure that this server is trusted and that it speaks for the remote user being called. The exporter then allows labeled data to flow across the Internet.

The audio chat server, running on the Haskell application server, handles connections and audio multiplexing. The application server ensures that client labels are respected. Specifically, the audio chat server cannot disclose the audio stream elsewhere on the network or save it unlabeled to disk.

The chat application presents an address book to the user upon startup. This interface is generated by Starlight as a part of the trusted UI. It includes a widget that indicates that the microphone will be shared, thus showing the user the link (and flow) between input device and remote peers. The user then selects which peers can receive data from the microphone device. In response, the microphone is labeled by the runtime as a disjunction of all the peers selected by the user. This enables the application to connect to the exporters and send the labeled microphone stream. A diagram of this labeling scheme can be seen in Figure 5.

7.2 Desktop and web search

Our search application demonstrates an application that accesses the local NaCl filesystem, talks to a trusted hosted provider to store an index for distributed searching, and communicates to untrusted network hosts to perform a websearch. All this is done without disclosing the user's private files. We implemented this by porting `idutils` [7] to Starlight, developing a frontend wrapper for it to obtain the search root directory and search terms.

We perform the web search first, while the search application is still "untainted" and holds no secrets (as it has not read local files yet). Only after all web search requests are fulfilled do we perform the local search, after which, (unlabeled) network access will be denied. For a new search, a fresh, instance of the application is created and search starts over from the beginning. Desktop search can also upload the client's index to a server running Starlight to support server-side search while still guaranteeing that the user's file contents will not be disclosed. Figure 6 shows the components and labeling of our search application. `Desksearch` is allowed to collaborate with third-party plugins or applications to extract metadata from files for better search results.

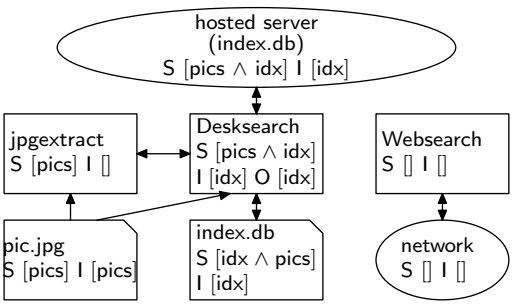

Figure 6: Labeling in the search application.

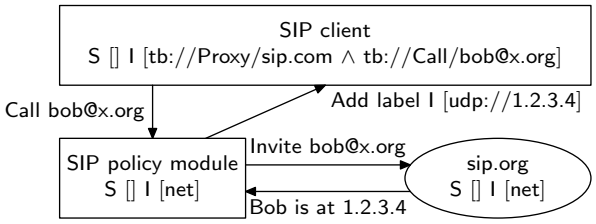

Figure 7: Labeling in the SIP application.

7.3 SIP phone

We ported `pjsip` [9] to Starlight and developed a SIP phone web application to demonstrate how Starlight can interact with already deployed servers and to show the intended use of policy modules. SIP works by contacting a configured local proxy to resolve the address of the person being called, and then connecting to that address directly. We use a trusted UI textbox to configure the SIP server and the application obtains networking privileges to the proxy via a networking policy module. The policy module converts endorsements for textboxes with caption "Enter server IP and port" to privileges for network connections to that IP and port, and is generic enough to be used by many networking applications. For outgoing calls, an additional policy module is needed to grant permission to contact the SIP server of the person being called. This takes two endorsements: one for the local SIP proxy (to prove which proxy the user wishes to use) and one for the SIP address being called. The policy module will contact the SIP proxy, resolve the address of the person being called and allow the application to connect to that address. This policy module is generic to SIP and can be used by any SIP client. The policy module can be avoided if the SIP proxy happens to be the origin server and is configured to route all SIP traffic. This is a mode often used for SIP to work in the presence of NATs. Figure 7 shows the labeling of our SIP application. With the two endorsements of the SIP proxy and the destination SIP address, the application uses the SIP policy module to obtain privileges for the IP address of the person being called and then performs the call.

The UI of our SIP client is implemented in Javascript. When receiving a call, the caller ID is displayed in a Javascript textbox, which thanks to BFlow, we can ensure it will not be disclosed on the network.

7.4 Peer-to-peer video player

We wrote a peer-to-peer video player that uses BitTorrent to share and stream videos on demand, showing the flexibility of Starlight's client-side. We ported `libbt` [1] and `ffmpeg` [3] to do this. Users can choose which folder to share via a file chooser through the trusted UI. This endorsement is used by a generic policy module to give read permission to all files in the selected subtree.

Allowing networking is more challenging. BitTorrent talks to a tracker and learns the IPs of hosts participating in the torrent. We implemented a BitTorrent policy module that given an endorsement for a torrent file, contacts a tracker, determines which IPs are participating in the torrent and grants the application privileges to talk to those IPs. The endorsement is proof that a user wishes to participate in a torrent. In our case this is a hyperlink to a torrent file. We assume the user trusts the tracker and authorizes connections when clicking on a torrent link (which displays the full URL of the tracker and torrent). This is similar to what users already do today when opening a torrent file from a web browser which then launches a torrent application. Our BitTorrent policy module is generic for the BitTorrent protocol—it can be used by any BitTorrent client for peer-to-peer sharing, or in fact be used to implement a video player as we did.

Figure 8 shows the labeling in our video player. Given an endorsement that the user clicked on a torrent file, the BitTorrent policy module contacts the tracker and allows the application to connect to the IPs and ports listed by the tracker.

7.5 Other: text editor, networked Quake

We ported the `ed` [2] text editor to Starlight. This application demonstrates a simple way the information flow security framework of NaCl can be used purely on the client-side to protect user's data from other NaCl applications, and how application writers can set their own security policies. By default all the files it creates are accessible only to itself, that is, other applications will require clearance from `ed` even for reading the files. We extended `ed` to recognize when files are saved in a folder

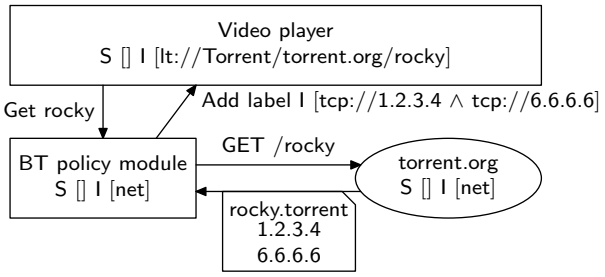

Figure 8: Labeling in the video player application.

named "shared". In this case `ed` will automatically grant clearance to applications wishing to read any of those files. Note that applications will still be unable to leak any information they learn from these files to the network. Another directory, "public", exists in which `ed` saves files unlabeled, allowing them to be read freely and sent over the network.

We also modified NaCl's Quake demo to support loading and saving games, and added support for peer-to-peer networked games. Quake's UI already asked players for the IP address of the server. We converted that to a trusted UI widget so the application gets an endorsement for the user typing in an IP address and port number. We use the same networking policy module used in our SIP phone for proxy connection. Once again we do not need to ask the user for additional privileges—we can infer intent from actions that need to be performed anyway (like configuring server addresses).

8 Evaluation

We evaluate Starlight in three ways. First, we argue for the security of both the design and implementation of Starlight. Next, we show that the Starlight API is flexible enough to allow porting of existing applications with minimal effort. Finally, we present performance benchmarks and discuss the impact of IFC on our measurements.

8.1 Security

In using existing security design such as exporters (from DStar) for networking and DC labels for our label format, we inherit their associated security properties by construction. Below, we discuss Starlight-specific security and trust concerns.

On the client we increase the TCB of NaCl and thus its attack surface—the exporter and label checking code account for 1150 lines. Our implementation makes an effort to provide safe APIs that do not allow NaCl applications to leak information through covert channels. Covert channels are an issue in most information flow control systems, and Starlight attempts to address this issue with the use of clearance. Nevertheless, our implementation approach is not ideal, as much of our APIs are implemented in the trusted runtime. Previous IFC systems, such as HiStar, minimize the attack surface by limiting the trusted kernel interface to a small subset on top of which a more complex, but untrusted, API can be built. Although this architecture is viable in our environment, it conflicts with our desire to use the existing NaCl code base.

On the server side, we use LIO to guarantee that end-to-end confidentiality and integrity is preserved, and that clearance is respected when confining an application. The latter properly allows us to essentially "sandbox" different applications. We extend LIO with mutable locations and channels (for a total of 132 lines to TCB), however the IFC properties of these constructs closely match the existing mutable references and filesystem constructs, part of which have been proven sound [41]. Introducing threads in an IFC system would typically require much stronger guarantees than those of LIO. However, we avoid new security implications by limiting the observation power of untrustworthy application (e.g., disallow measuring CPU time). Moreover, threads are limited to communicating through channels and mutable locations, and different applications are restricted from communicating via the network or file-system. From the perspective of IFC soundness, different application threads can be viewed as separate processes.

In our architecture, as summarized by Figure 2, there are various notions of trust imposed on Starlight users. First, users must trust the correctness of the Starlight runtime locally and in the cloud. A malicious application could leverage a vulnerability in either the client or server to circumvent IFC restrictions and violate the system security policies. For example, if the server exporter does not properly verify that client exporters can speak for the data, a malicious client application can essentially forge any data and transmit it to other clients of the application server. As previously mentioned, the user must also trust the cloud provider to actually run Starlight. (A level of trust that might someday be reduced tamper-resistant hardware and attestation, though current datacenter hardware is not up to the task.)

In addition to trusting the correctness of Starlight, users must trust their WebFinger provider—specifically that the provider responds with the correct public key when queried. Of course, to remove this trust users can run private WebFinger servers. Furthermore, as the notion of identity in our system depends on WebFinger providers with which users may only have a transitive

Application	Total	Patch	Policy module
Editor	3,210	0	0
Desktop search	43,016	0	0
SIP phone	413,343	591	0 for NAT. 150 for peer-to-peer.
Video player	407,561	886	91
Quake	109,302	2,717	0

Table 1: Changes needed to port applications and lines of trusted code.

relationships, an implicit "trustworthiness is bestowed on them. For example, if Alice tells Bob that her WebFinger address is `alice@home.org`, Bob cannot but assume that `home.org` will provide Alice's correct public key. However, depending on the sensitivity of the communication, integrity of the WebFinger may only be of concern to one of the parties. Consider, for example, Bob transmitting data that is sensitive to Alice but not to him: in this case, only Alice must trust the integrity of her WebFinger provider.

The client-side Starlight design also relies on the notion of trusted UI widgets. These UI components include, e.g., a file browser and a dialog that can be used to give applications access to the filesystem, or input devices such as a microphone. As in capability systems [34], Starlight infers policies exclusively from user actions in the trusted UI. Thus, even if a malicious application presents the user with a crafted widget, it will not be able to escalate its privileges. Similarly, an application cannot access the user's microphone by "tricking" Starlight into supplying it privileges by claiming that user has approved the said action. Moreover, the trusted UI runs in a separate process, completely segregated and tamper proof from NaCl applications, a design approach similar to the powerbox [34, 42]. Of course, malicious applications can still employ social engineering to "trick" users into performing certain actions, such as installing malicious software. Mitigating such attacks is outside the scope of Starlight, and we refer the reader to [18] for methods addressing such issues.

8.2 Porting

Table 1 shows the amount of code that was required to port applications to Starlight, and the size of the policy module required by each application. All five applications we ported to Starlight from existing code required only very small patches. For example, adding networking and filesystem support to Google's NaCl port of Quake only required 453 lines of code (compared to 109,302 lines of code overall, and 2,264 lines of code in the original Google patch for NaCl). Policy modules were only required for the SIP phone and Video player applications, and even then the modules were only 91 and 150 lines of code, respectively. Such small additions to the trusted code base can feasibly be audited by hand.

Rewriting these applications in a language already supported by browsers—*e.g.,* Javascript—would require significantly more effort. Note that porting some of these applications to Javascript, or vanilla NaCl would have been impossible due to the lack of filesystem and network support. Therefore, not only does Startlight provide better security guarantees, it also enables features not possible in existing systems.

Benchmark	Linux	NaCl	Starlight no label	Starlight
`creat`	6	20	447	460
`open`	3	10	40	44
`unlink`	13	24	71	78
`read`	4	–	–	5
`write`	5	–	–	14
`ping` (net)	147	148	259	266
Desksearch (index)	921ms	–	–	713ms
Desksearch (find)	1.01ms	–	–	0.97ms

Table 2: Overhead (us) of Starlight I/O.

8.3 Performance

Compared to a running applications natively, Starlight introduces overhead in both the NaCl sandbox and from information flow control checks. Since the overhead of the NaCl sandbox is well understood [44], we use this as our baseline for evaluating Starlight, and examine the overhead introduced by the IFC checks.

We perform microbenchmarks on Starlight's I/O, measuring the operation time of common filesystem operations, and a simple local network TCP ping (measuring RTT). All benchmarks were performed on a 2006-era laptop (Thinkpad x60s), a dual core Intel Core Duo 1.6GHz with 1GB of RAM running Linux 2.6.33 and Firefox 3.6. Table 2 shows the results in Starlight, Starlight with no label checks, a version of NaCl that simply forwards the system calls, and native Linux.

The baseline NaCl costs 2–3x compared to Linux. Starlight introduces an overhead of up to 4x on top of NaCl in all microbenchmarks except `creat` which is an order of magnitude slower in Starlight. Nearly all of this overhead is incurred before adding label checks. We infer that the cause is Starlight's unoptimized implementation of the filesystem, which searches for a unique filename for creation by listing directory contents (instead of storing an index), and updating a symlink to point to that new file. This can be optimized pre-generating an index. Label checks are stored and compared as strings

Test	mplayer	Flowplayer	Starlight
CPU usage	5%	16%	18%
Max fps	303	–	66

Table 3: Peer to peer video player performance.

and could be optimized by a binary representation.

Regardless, end-to-end measurements of the Desksearch application on Linux and Starlight show that this overhead does not significantly impact performance. The bottom two rows of Table 2 show the times to perform an `index` and `find` using the Desksearch application on both Linux and Starlight. Starlight actually outperforms Linux in these cases. This is probably because glibc performs an `mmap` on each file open, while NaCl's libc does not. Nonetheless, the filesystem overhead incurred by Starlight is clearly insignificant in this case.

The `ping` benchmark shows the overhead of labeled networking. First, `tcpcrypt` adds overhead by encrypting and MACing the packet (a 111us delay), and second, Starlight needs to encode and decode label information, adding an extra 7us delay. Most application deployed on the Internet will suffer from a much larger propagation delay thus making the overhead of Starlight less significant.

We also measure the performance of our Starlight video player to show that the platform is fast enough for complex and computationally intensive applications. Table 3 shows performance results when playing a h.264 video on our Starlight video player, Flowplayer [4] (a Flash video player) running in Flashplayer 10 and mplayer [8]. We measure CPU usage (idle time) to determine whether resource consumption is similar to alternatives accepted today. While our video player is not optimized, it consumes significantly more CPU than mplayer, but about the same amount of CPU as the Flash player. mplayer is highly optimized and has direct graphics access—it is not forced to paint to the browser. We also examine the maximum framerate our video player was able to achieve. The particular video we played was encoded at 29fps and at full CPU usage we could have handled playing it at more than twice that speed. mplayer can handle five times the frame rate of the Starlight video player, but clearly Starlight is fast enough for video playback and leaves. Again, this difference is likely largely attributed to mplayer's high optimization and direct access to graphics. In face, new versions of NaCl already support direct screen access and even hardware acceleration so we expect that high frame rate when we port Starlight to the latest version of NaCl.

9 Related work

Starlight brings information flow control [23, 25, 35, 48] to web applications. Starlight integrates with BFlow [46] to support Javascript. BFlow optionally supports server-side IFC, for example, by confining CGI scripts using Flume [32]. However, BFlow assumes that web sites are monolithic entities with a single owner of hardware and web server software. Thus, BFlow lacks the security benefits that Starlight achieves through cooperation with cloud providers. Additionally, BFlow does not enforce security policies specified by the user on the client-side, rather the server specifies which information is secret.

Java, Flash, HTML5 and Silverlight each allow some degree of both filesystem and network access. However, they tend to be limited by, *e.g.,* the same origin policy, or too broad by, *e.g.,* permitting unmitigated access to the filesystem. Starlight allows full use of the network and filesystem but restricts access based on IFC policies set by the user, applications and servers. Also unique to Starlight is the ability to infer permissions based on existing user actions.

Starlight uses disjunctive category labels, previously used only in languages-level systems [35, 41]. Starlight adapts them for an OS-like environment and shows how disjunctive labels enable easy sharing of data, a requirement and good fit for many web applications. Starlight's label system is most similar to HiStar's [48], although the addition of disjunctions eliminates the need for other more complex sharing mechanisms like gates. Starlight's network support is based on that of DStar [49]. Unlike DStar, Starlight is geared for the Internet, providing a distributed naming mechanism and service for discovering public key of principals—Starlight goes beyond the LAN deployment scenario.

Starlight combines information flow control with the idea of granting privileges based on user actions. Previous work focused [30, 31, 42, 45] on the usability and design of security related user feedback, whereas we focus on deriving user intent and privilege. We contribute a two step process for applying the idea in a generic way. By separating endorsement and privileges, we objectively monitor user actions and defer to policy modules to interpret these user actions. This separation also localizes policy code allowing for simpler auditing. Abadi and Fournet [13] use past code executions, rather than user actions to grant privileges.

The endorsements and privileges granted to Starlight applications, though similar, differ from capabilities [17, 38]. Endorsements cannot be invoked but merely act as proof. The approach of translating endorsements to privileges is similar in nature to PolicyMaker [16] though the latter focuses on trust relationship rather than attempting to derive user intent from actions. However, our notion

of privileges (though not endorsements) are in form capabilities.

A series of work based on Jif [35], the most popular information flow control compiler for Java, addresses IFC in web applications. SIF (Servlet Information Flow) is a framework that essentially allows programmers to write their web applications as Servlets in Jif [20]. Swift [19], based on Jif/split [47, 51], compiles Jif-like code for web applications into JavaScript code running in the client and Java code running in the server by applying clever partitioning algorithms. Our approach differs from these works in several ways. Firstly, our web application model realizes the presence of three separate parties, including the application author, and thus accounts for the execution of untrustworthy code. Second, as demonstrated by using NaCl in the client-side and Haskell+LIO on the server-side, our model does not restrict applications to being implemented using a single development platform; conceptually, any dynamically enforced IFC system can be used in place of LIO or NaCl, with the sole requirement of requiring an implementation of the DStar protocol. Of course, the Jif-based systems can leverage the features of a statically enforced IFC system and thus prevent runtime failures and covert channel attacks.

MashupOS [26] allows sandboxing web applications though the user must decide a priori which components to trust whereas in Starlight all application components are untrusted. The main distinction is that Starlight uses IFC which is a form of mandatory, rather than discretionary, access control [21, 37]. Similarly, Tahoma [28] is also a discretionary access control system for building web applications, offering an all-or-nothing model in terms of who is allowed to access sensitive data. More recently, Waterken [33] has been used to build secure distributed web applications, however, using the object capability model, which cannot enforce DIFC as directly.

IBOS [43] restructures the operating system to be tailored specifically to web browsing, resulting in dramatic reductions to the TCB. While Starlight extends to server-side too, some of the ideas from IBOS could be applied to Starlight's client-side to limit compromise. Xax [24] like NaCl provides a sandbox for browser applications and it could be a good alternative sandbox framework for Starlight.

Logical attestation [40] allows specifying a security policy in logic and the system ensures that the policy is obeyed by all server-side components. Starlight includes client-side components and in addition the policy is generated based on user action without requiring any manual specification. Both in Logical attestation and CloudVisor [50] TPMs are used to eliminate trust from Cloud providers. It may be possible to use these systems or TPMs in server-side Starlight, too for the same benefit.

10 Conclusion

We presented Starlight, a clean-slate unified security architecture for the web. Starlight uses information flow control to ensure security where policies must be enforced across multiple components on both the client and server. While today each new web technology requires ad-hoc reasoning about security, Starlight provides an *objective* basis for thinking about web components. We argue that the web's inherent interactivity allows security policies to be seamlessly inferred from application usage rather than forcing cumbersome settings on the user. Furthermore, within our unified security architecture we can safely add rich web APIs such as filesystem, networking, and device access.

We built a number of applications on top of Starlight including an audio conferencing system, a SIP phone, a BitTorrent based video player, and networked Quake. None of these applications need to be trusted by users, illustrating the power and generality of the Starlight design. Most importantly, what prevents these applications from being built on today's web are precisely the security concerns that Starlight addresses.

References

[1] libbt. `http://libbt.sourceforge.net/`.

[2] Ed - A line-oriented text editor. `http://www.gnu.org/software/ed/`.

[3] FFmpeg. `http://www.ffmpeg.org/`.

[4] Flowplayer - Flash Video Player for the Web. `http://flowplayer.org/`.

[5] The FreeType Project. `http://www.freetype.org/`.

[6] Google Chat. `http://www.google.com/talk/`.

[7] ID Utils - GNU Project - Free Software Foundation (FSF). `http://www.gnu.org/software/idutils/`.

[8] MPlayer - The Movie Player. `http://www.mplayerhq.hu/`.

[9] Open Source Portable SIP Stack and Media Stack for Windows and Mac OS X from PJSIP.ORG. `http://www.pjsip.org/`.

[10] SDL-Terminal library — Get SDL-Terminal library at SourceForge.net. `http://sourceforge.net/projects/sdl-terminal/`.

[11] Twilio. http://www.twilio.com/api/client/.

[12] Adobe flash player 10.0.12 security. White paper, November 2008. URL \url{https://www.adobe.com/devnet/flashplayer/articles/flash_player10_security_wp.html}. Available online (52 pages).

[13] M. Abadi and C. Fournet. Access control based on execution history. In *IN PROCEEDINGS OF THE 10TH ANNUAL NETWORK AND DISTRIBUTED SYSTEM SECURITY SYMPOSIUM*, pages 107–121, 2003.

[14] D. Akhawe, A. Barth, P. Lam, J. Mitchell, and D. Song. Towards a formal foundation of web security. In *Computer Security Foundations Symposium (CSF), 2010 23rd IEEE*, pages 290–304. IEEE, 2010.

[15] Andrea Bittau, Michael Hamburg, Mark Handley, David Mazieres, and Dan Boneh. The case for ubiquitous transport-level encryption. In *USENIX Security 2010*, USENIX Security10. USENIX Association, 2010.

[16] M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized trust management. In *SP '96: Proceedings of the 1996 IEEE Symposium on Security and Privacy*, pages 164+, Washington, DC, USA, 1996. IEEE Computer Society. ISBN 0-8186-7417-2. URL http://portal.acm.org/citation.cfm?id=884248.

[17] A. C. Bomberger, W. S. Frantz, A. C. Hardy, N. Hardy, C. R. Landau, and J. S. Shapiro. The keykos nanokernel architecture. In *Proceedings of the Workshop on Micro-kernels and Other Kernel Architectures*, pages 95–112, Berkeley, CA, USA, 1992. USENIX Association. ISBN 1-880446-42-1.

[18] J. C. Burstoloni and R. Villamarin-Salomon. Improving security decisions with polymorphic and audited dialogs. In *Symposium On Usable Privacy and Security*, pages 76–85. ACM Press, July 2007.

[19] S. Chong, J. Liu, A. Myers, X. Qi, K. Vikram, L. Zheng, and X. Zheng. Secure web applications via automatic partitioning. *ACM SIGOPS Operating Systems Review*, 41(6):31–44, 2007.

[20] S. Chong, K. Vikram, and A. Myers. Sif: Enforcing confidentiality and integrity in web applications. In *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*, page 1. USENIX Association, 2007.

[21] D.Bell and L.LaPadula. Secure computer systems: Unified exposition and multics interpretation. In *Technical Report ESD-TR-75-306, MTR-2997, MITRE, Bedford, Mass*, 1975.

[22] Deian Stefan, David Mazières, John Mitchell, Alejandro Russo. Disjunction Category Labels. In *NordSec 2011*, LNCS. Springer, October 2011.

[23] D. E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19:236–243, May 1976. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/360051.360056. URL http://doi.acm.org/10.1145/360051.360056.

[24] J. R. Douceur, J. Elson, J. Howell, and J. R. Lorch. Leveraging legacy code to deploy desktop applications on the web. In *OSDI*, pages 339–354, 2008.

[25] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazières, F. Kaashoek, and R. Morris. Labels and event processes in the asbestos operating system. In *SOSP*, 2005.

[26] J. Howell, C. Jackson, H. J. Wang, and X. Fan. Mashupos: operating system abstractions for client mashups. In *HOTOS'07: Proceedings of the 11th USENIX workshop on Hot topics in operating systems*, pages 1–7, Berkeley, CA, USA, 2007. USENIX Association.

[27] L. Huang, E. Chen, A. Barth, E. Rescorla, and C. Jackson. Talking to yourself for fun and profit. In *Proceedings of the Web 2.0 Security & Privacy*, 2011.

[28] R. C. Jacob, R. S. Cox, J. G. Hansen, S. D. Gribble, and H. M. Levy. A safety-oriented platform for web applications. In *In IEEE Symposium on Security and Privacy*, pages 350–364, 2006.

[29] S. Jones, A. Gordon, and S. Finne. Concurrent haskell. In *ANNUAL SYMPOSIUM ON PRINCIPLES OF PROGRAMMING LANGUAGES*. Citeseer, 1996.

[30] A. Karp, M. Stiegler, and T. Close. Not one click for security? In *SOUPS '09: Proceedings of the 5th Symposium on Usable Privacy and Security*, pages 1–1, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-736-3. doi: http://doi.acm.org/10.1145/1572532.1572558.

[31] A. H. Karp and M. Stiegler. Making policy decisions disappear into the user's workflow. In *CHI EA '10: Proceedings of the 28th of the international conference extended abstracts on Human factors in*

computing systems, pages 3247–3252, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-930-5. doi: http://doi.acm.org/10.1145/1753846.1753966.

[32] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. Information flow control for standard os abstractions. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, SOSP '07, pages 321–334, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-591-5. doi: http://doi.acm.org/10.1145/1294261.1294293. URL `http://doi.acm.org/10.1145/1294261.1294293`.

[33] A. Mettler, D. Wagner, and T. Close. Joe-e: A security-oriented subset of java. In *Network and Distributed Systems Symposium. Internet Society*, 2010.

[34] M. Miller, B. Tulloh, and J. Shapiro. The structure of authority: Why security is not a separable concern. *Multiparadigm Programming in Mozart/Oz*, pages 2–20, 2005.

[35] A. Myers and B. Liskov. Protecting privacy using the decentralized label model. *ACM TOSEM*, 9(4): 410–442, 2000.

[36] R. Sailer, X. Zhang, T. Jaeger, and L. Van Doorn. Design and implementation of a TCG-based integrity measurement architecture. In *Proceedings of the 13th conference on USENIX Security Symposium-Volume 13*, pages 16–16. USENIX Association, 2004.

[37] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems, 1975.

[38] J. S. Shapiro, J. M. Smith, and D. J. Farber. Eros: a fast capability system. In *In Symposium on Operating Systems Principles*, pages 170–185, 1999.

[39] K. Singh, A. Moshchuck, H. Wang, and W. Lee. On the incoherencies in web browser access control policies. In *Proceedings of IEEE Symposium on Security and Privacy (Oakland)*. IEEE, 2010.

[40] E. Sirer, W. de Bruijn, P. Reynolds, A. Shieh, K. Walsh, D. Williams, and F. Schneider. Logical attestation: an authorization architecture for trustworthy computing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 249–264. ACM, 2011.

[41] D. Stefan, A. Russo, J. C. Mitchell, and D. Mazières. Flexible dynamic information flow control in Haskell. In *Haskell Symposium*, pages 95–106. ACM SIGPLAN, September 2011.

[42] M. Stiegler, A. Karp, K. Yee, T. Close, and M. Miller. Polaris: virus-safe computing for windows xp. *Communications of the ACM*, 49(9):83–88, 2006.

[43] S. Tang, H. Mai, and S. T. King. Trust and protection in the illinois browser operating system. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–8, Berkeley, CA, USA, 2010. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=1924943.1924945`.

[44] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native client: a sandbox for portable, untrusted x86 native code. *Commun. ACM*, 53:91–99, January 2010. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/1629175.1629203. URL `http://doi.acm.org/10.1145/1629175.1629203`.

[45] K.-P. Yee. User interaction design for secure systems. In *In Proceedings of the 4th International Conference on Information and Communications Security*, pages 278–290. Springer-Verlag, 2003.

[46] A. Yip, N. Narula, M. Krohn, and R. Morris. Privacy-preserving browser-side scripting with bflow. In *EuroSys '09: Proceedings of the 4th ACM European conference on Computer systems*, pages 233–246, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-482-9. doi: http://doi.acm.org/10.1145/1519065.1519091.

[47] S. Zdancewic, L. Zheng, N. Nystrom, and A. Myers. Untrusted hosts and confidentiality: Secure program partitioning. In *ACM SIGOPS Operating Systems Review*, volume 35, pages 1–14. ACM, 2001.

[48] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in HiStar. In *OSDI*, 2007.

[49] N. Zeldovich, S. Boyd-wickizer, and D. Mazires. Securing distributed systems with information flow control. In *In Proc. of the 5th NSDI*, pages 293–308, 2008.

[50] F. Zhang, J. Chen, H. Chen, and B. Zang. CloudVisor: retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 203–216. ACM, 2011.

[51] L. Zheng, S. Chong, A. C. Myers, and S. Zdancewic. Using replication and partitioning to build secure distributed systems. In *Proceedings of the 2003 IEEE Symposium on Security and Privacy*, SP '03, Washington, DC, USA, 2003. IEEE Computer Society.