Enforcing Configurable Trust in Client-side Software Stacks by Splitting Information Flow

Lenin Singaravelu[1], Bernhard Kauer[2], Alexander Boettcher[2], Hermann Härtig[2],
Calton Pu[1], Gueyoung Jung[1], Carsten Weinhold[2]

[1] Georgia Institute of Technology, [2] Technische Universität Dresden

{lenin, calton, gueyoung.jung}@cc.gatech.edu, {kauer, boettcher, haertig, cw183155}@os.inf.tu-dresden.de

ABSTRACT

Current client-server applications such as online banking employ the same client-side software stack to handle information with differing security and functionality requirements, thereby increasing the size and complexity of software that needs to be trusted. While the high complexity of existing software is a significant hindrance to testing and analysis, existing software and interfaces are too widely used to be entirely abandoned. We present a proxy-based approach called FlowGuard to address the problem of large and complex client-side software stacks. FlowGuard's proxy employs mappings from sensitiveness of information to trustworthiness of software stacks to demultiplex incoming messages amongst multiple client-side software stacks. One of these stacks is a fully-functional legacy software stack and another is a small and simple stack designed to handle sensitive information. In contrast to previous approaches, FlowGuard not only reduces the complexity of software handling sensitive information but also minimizes modifications to legacy software stacks. By allowing users and service providers to define the mappings, FlowGuard also provides flexibility in determining functionality-security tradeoffs.

We demonstrate the feasibility of our approach by implementing a FlowGuard, called BLAC, for https-based applications. BLAC relies on text patterns to identify sensitive information in HTTP responses and redirects such responses to a small and simple TrustedViewer, with an unmodified legacy software stack handling the rest of the responses. We developed a prototype implementation that works with a prominent bank's online banking site. Our evaluation shows that BLAC reduces size and complexity of software that needs to be trusted by an order of magnitude, with a manageable overhead of few tens of milliseconds per HTTP response.

1. INTRODUCTION

Security-sensitive client-server applications such as online banking and electronic commerce are increasingly used by a large number of people. These applications involve exchange of information with differing security and functionality requirements, e.g., passwords, addresses, account status, images, scripts, etc. Service providers realize this and use security mechanisms such as SSL, dual-factor authentication, Transaction Authorization Numbers and keypads to enter PIN to protect exchange of highly sensitive information. However, these applications use the same client-side software stack to handle all categories of information.

Employing the same software stack to handle information with differing security and functionality requirements has two implications: First, security-sensitive functionality residing in the same protection domain as non-sensitive functionality. Therefore, vulnerabilities in non-sensitive functionality can be used to compromise security-sensitive functionality, e.g., a buffer overflow in image processing subsystem of the Internet Explorer browser can be used to install malicious extensions [4]. Secondly, co-locating security-sensitive functionality with non-sensitive functionality also increases the size and complexity of software that needs to be trusted (greater than 2 million LOC for the Mozilla Firefox browser), which in turn makes analysis and testing of such software harder.

One approach to mitigating this problem is by constructing application-specific Trusted Computing Bases (TCBs) [22]. While this approach eliminates unnecessary software programs from the TCB, it does not address the large size and complexity of software in the TCB and at the application level. AppCore [44] extends this approach by refactoring existing software into small and simple components which are then used in security-sensitive applications. Doing so reduces the complexity of the software components that have to be trusted (Trusted Components), thereby, simplifying the analysis and testing of Trusted Components. However, this approach requires considerable efforts in redesigning or refactoring of current software stacks.

Alternatively, one can employ the Proxos system [46], which allows application programs to configure their trust in operating systems by partitioning the system call interface into trusted and untrusted parts. The trusted part of the interface is implemented by a trustworthy OS while the rest of the interface is implemented by an untrusted commodity OS. By limiting the flow of sensitive information to the application and to the trusted portion of interface, Proxos provides configurable trust in systems software. However, middleware and application-level software too suffer from multiple critical security vulnerabilities, threatening the flow of sensitive information.

This paper proposes FlowGuard, a system architecture that combines the desirable features of both approaches: smaller and simpler Trusted Components for handling security-sensitive information and the reuse of existing software and interfaces for handling non-sensitive information. The first contribution of this paper is the FlowGuard architecture, which consists of a proxy server and multiple client-side software stacks of varying functionality and trustworthiness executing on a microkernel or a virtual machine monitor (VMM). The proxy server intercepts information flow from the server and determines the sensitiveness of data items in the flow. Based on its classification, the proxy server demultiplexes information flow from the server amongst the various client-side software stacks. FlowGuard supports software stacks of varying complexity and functionality. At one extreme is a fully-functional, lightly-modified, legacy software stack and at the other extreme is a small and simple but functionally constrained stack meant to perform a small set of operations. Based on software engineering studies, we expect the small and functionality constrained stack to be more amenable to exhaustive testing, resulting in software with fewer vulnerabilities [34][43].

The second contribution of this paper is a concrete instance of the FlowGuard architecture for https-based applications (called BLAC). BLAC consists of an https proxy executing on top of a microkernel and two client-side software stacks: a small and functionally limited application called the TrustedViewer, and a legacy software stack running a lightly-modified, fully-functional browser. The https proxy employs text patterns to identify HTTP responses containing sensitive information and directs these responses to the TrustedViewer. By allowing users to specify text patterns, BLAC enforces the functionality-security tradeoffs preferred by the user. In our prototype implementation, we show that the TrustedViewer is two orders of magnitude smaller than the Mozilla Firefox browser and the complete trusted software stack is an order of magnitude smaller than a legacy software stack. Even with an unoptimized implementation, we show that the performance overhead of BLAC is in the range of few tens of milliseconds per HTTP response, and well within user expectations. Moreover, since the https proxy implements the HTTP and the SSL protocols without any modifications, it can be used with legacy service providers. We successfully tested our prototype with the online banking site of a major international bank.

The rest of the paper is organized as follows: Section 2 discusses the challenges and opportunities in designing client-side computing bases for client-server applications. Based on the discussion, we also present a list of requirements for an effective solution. Section 3 presents the FlowGuard architecture and discusses the components of FlowGuard in detail. Section 4 describes BLAC, an instance of the FlowGuard architecture for https-based applications and Section 5 evaluates BLAC. We discuss the general applicability of FlowGuard in Section 6. Section 7 discusses the related work and Section 8 concludes the paper.

2. MOTIVATION

In a client-server application, client-side software typically consists of systems software (OS, VMM), middleware and application level software. For a given client-server application, we define its *client-side software stack* (also referred to as *software stack* for brevity) as set of all software components on the client-side that is needed satisfy the functionality and security requirements of the application. We define *Trusted Components* as a subset of the software stack that can directly or indirectly compromise the flow of sensitive information. Trusted Components include any application-level software and middleware that have access to sensitive data and the TCB for the application.

One should note that TCB or Trusted Components do not by themselves imply trustworthiness. These software components are trusted in the sense that they *must be trusted* to protect the security properties of information flow associated with the application. Only by using secure components do we transform these trusted components into trustworthy components.

2.1 Challenges

There are two key challenges to building trustworthy client-side software stacks. First, current software stacks are large and complex, hindering analysis and testing. This results in Trusted Components with multiple critical security vulnerabilities. For example, the Mozilla Firefox browser is a popular client-side software program that supports various information exchange protocols such as HTTP, and FTP. This browser contains about 2 MLOC (§ 5.1) and suffers from multiple security vulnerabilities including arbitrary code execution, information leak vulnerabilities and security system bypass [4][5]. Application-level software typically run on large commodity operating systems, e.g., 380 KLOC for a typical Linux kernel configuration and 11 MLOC for the complete Windows XP operating system and support software [13]. In addition, these software programs also rely on several support software (middleware) that run at the same or higher privilege levels than the client application, e.g., X server on Linux runs with superuser privileges. The operating system kernels and middleware suffer from multiple vulnerabilities, including arbitrary code execution, security system bypass and information leaks [6][7].

The second challenge is the popularity and ubiquity of current software and interfaces. For example, web browsers are increasingly used in many online applications including mail, spreadsheet and document editing. Users are very familiar with the user-interface (UI) and the functionality provided by the browser. Limiting the functionality or modifying the UI will reduce the appeal of the modified software. Similarly interfaces at various levels of the software stack are too widely used to modified or abandoned outright, e.g., API

between OS kernels and application-level software can be enhanced with information flow primitives [25], but this approach will not be applicable to the large number of existing software. Information exchange protocols between service provider and client, e.g., HTTP protocol, too are well entrenched that solution requiring modifications to the protocol will have limited appeal.

Previous efforts have addressed the first challenge, large and complex software stacks, by proposing application-specific TCBs (e.g., Terra [17]). While this reduces the size and complexity of the TCB, it does not address complexity of middleware and application-level software. AppCores [44] extends this approach to all layers of the software stack by reducing complexity of Trusted Components of application-level software. However, reducing the complexity of Trusted Components requires modifications such as curtailing the functionality of the components and narrowing the interface exported by the components.

The Proxos approach attempts to work around this problem by allowing software developers to specify trust in one of the interfaces – the OS-application (system call) interface – by splitting the interface into a trusted and untrusted part. The trusted part of the interface is implemented by a trusted OS, whereas the untrusted part is implemented by a commodity operating system. By doing so, Proxos allows the reuse of current application-level software and systems software as far as possible, while at the same time minimizing modifications to the application-level software. However, large and complex client software such as browsers and middleware such as the display manager still need to be completely trusted.

2.2 Opportunities

Client-server also applications possess many properties that allow us to reduce complexity *and* reuse existing software and interfaces as far as possible. First, information flow in a session of interaction in client-server application contains data items with differing security and functionality requirements. As a concrete example, consider a session in an online banking application, where the user logs in, checks her account, modifies her account and logs off. One can argue that account status information is less sensitive than user authorization for account modifications, e.g., transfer of funds, ordering cheques or change of personal information. Some banks already recognize this difference in sensitiveness of information and ask the user for an authorization token, e.g., a one-time Transaction Authorization Number (TAN), for account modifications to be approved [1]. From a functionality point of view, account status information also requires a richer presentation format, e.g., spreadsheets and graphs, whereas account modification information typically shows up in a tabular format on a *confirm* page. Sensitive input consists of human input in the form of a sequence of keystrokes or mouse clicks.

We can leverage this difference in security and functionality requirements and construct small and simple components to handle security-sensitive information and handling the rest of the information in legacy components. Doing so allows us to reuse existing interfaces as far as possible.

Another opportunity arises from the functionality and security tradeoffs preferred by the user and the service provider. For example, the user might be willing to use a legacy computing base on her computer system to handle sensitive information, but prefers to use a small and reduced functionality software stack to operate on all sensitive information when operating on a public-access machine. Similarly, during a virus outbreak, the service provider might ask users to interact using a smaller and simpler software stack more extensively than otherwise. By providing the user (and the service provider, if necessary) with freedom in choosing the software stack, we can ensure that security requirements are balanced with the functionality requirements.

2.3 Requirements

Based on our discussion in the previous sub-sections, we identify the following requirements for an effective solution:

R1. Reduce the complexity of the Trusted Components.
R2. Reuse existing software and interfaces as far as possible.
R3. Allow users and service providers to configure functionality-security tradeoffs.
R4. Ensure that malicious software cannot spoof Trusted Components, thereby minimizing the risk that the user or the service provider is tricked into revealing security-sensitive data.
R5. Minimize performance overheads.

3. FLOWGUARD

3.1 FlowGuard Architecture

The FlowGuard architecture takes a data-centric approach to securing the flow of information. The user and the service provider are allowed to classify data items in a session of information flow based on their security requirements. They also specify the trustworthiness of the different software stacks available on the client system. Based on this information, FlowGuard redirects the flow of information to appropriate software stack.

We assume that the user and service provider deem some software stacks more trustworthy than others. This difference could arise because of differences in software size and complexity, or because the trusted stack is instantiated from a trustworthy read-only image, or because the trusted stack uses high assurance components, e.g., hardened OS kernels. Evaluating and comparing the trustworthiness of software stacks is outside the scope of this paper.

Figure 1 presents an overview of the FlowGuard architecture. At the base of FlowGuard is a Trusted Platform, which

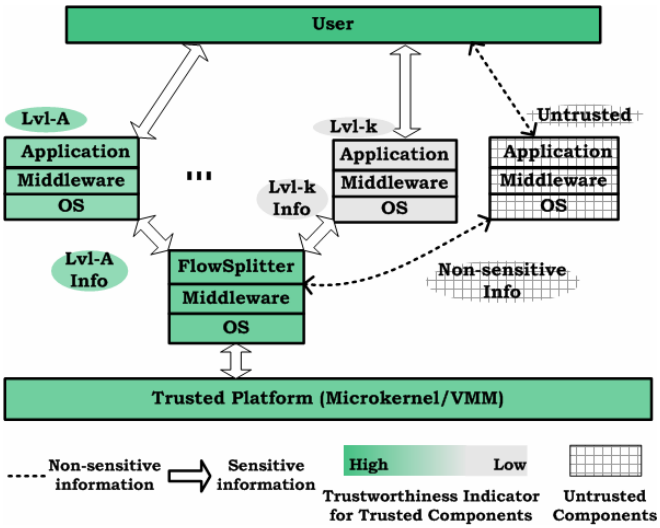

Figure 1. Overview of FlowGuard: Instead of passing on sensitive data directly to a legacy software stack, or to a small and simple Trusted Components, data is first directed towards a FlowSplitter. The FlowSplitter directs data to software stacks based on the sensitiveness of information and the trustworthiness of the software stack.

can be a VMMs, a microkernel or a hardened operating system kernel [13][30]. On top of this Trusted Platform is a group of software stacks of differing functionality and trustworthiness. Each of the software stacks contains an application-level program and all necessary middleware and systems software to interact with other application-level software, or hardware or with the service provider. The user of the system acts as a source of input. Alternative input sources such as disks and other IO devices also fit in FlowGuard as long as the software interface of a device (e.g., a file server for a disk) does not send sensitive input to software stacks that are not trustworthy enough to receive the input.

The main component of FlowGuard is a configurable FlowSplitter that functions as a proxy server. The FlowSplitter multiplexes information flows from multiple software stacks to the service provider. It also demultiplexes a single flow from the service provider amongst multiple software stacks. Demultiplexing is determined by configuration information provided by the user or the service provider. The FlowSplitter is discussed in detail in Section 3.1.1.

The rest of FlowGuard consists of multiple configurations of software stacks that are, from a security perspective, layered on top of the Trusted Platform. The composition of these stacks is discussed in Section 3.1.2

Hardware Requirements of FlowGuard. FlowGuard relies on Trusted Computing (TC) hardware support, as specified by the Trusted Computing Group [9], to function correctly. Specifically, it requires authenticated booting to ensure that the correct and unmodified versions of software are loaded. This information is displayed to the user via a trust

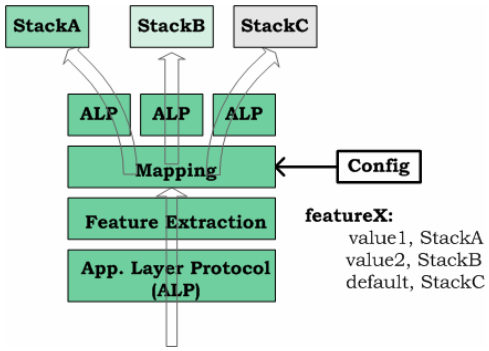

Figure 2. Incoming Message Processing in the FlowSplitter.

indicator, e.g., portion of screen or LED lights. In case the service provider wants to ascertain the configuration of the client-side software stack, FlowGuard also requires support for remote attestation. In addition, sealed storage is required to protect the confidentiality of the cryptographic keys that are foundations for the security of the rest of the system.

3.1.1 FlowSplitter

The FlowSplitter has two functions: First, it has to act as a proxy server for the client application. Second, it has to demultiplex information flow from the service provider amongst multiple client applications (and their software stacks) depending on the sensitiveness of information being exchanged. As with every proxy server, the FlowSplitter has to implement the application-layer protocol used for communication by the service provider, e.g., HTTP over SSL, POP3.

In addition to the regular proxy functionality, the FlowSplitter has to demultiplex data items in a single information flow amongst multiple software stacks. The recipient software stack is determined by the sensitiveness of incoming information and the trustworthiness of the stack, each of which is in turn dependent on user or service provider preferences.

Figure 2 illustrates the processing of an incoming message in the FlowSplitter. Since the FlowSplitter has to work with legacy service providers, we cannot assume that incoming flows are annotated with security classifications. Instead, the FlowSplitter has to extract features from the response and map them to security classifications. The security classifications are in turn mapped on to client-side software stacks based on the trustworthiness of the stacks. These mappings can be provided either by the user or the service provider. The details of the mapping vary from application to application. We discuss one such mapping in Section 4.3.2 for https-based applications.

Since the FlowGuard architecture entails the use of multiple software stacks in a single session, transfer of client-side state between the stacks is essential to proper functioning of the system. For purposes of discussion, we classify clients into two categories: stateless clients and stateful clients. In the case of stateless clients, switching between stacks is trivial. On the other hand, stateful clients complicate switching

of software stacks as the newly activated application-level software might need the client state of the previously used application-level software to carry out subsequent processing, e.g., cookie information in many https-based applications. For stateful clients, we assume that the client-side state is externally visible to the FlowSplitter *or* the application software provides interfaces to export and import state. When switching between stacks, the FlowSplitter extracts state information from the old stack and passes it on to the new stack.

In online banking, for example, client-side state consists of a cookie that is visible in every outgoing message. The FlowSplitter can easily capture this state information and pass it on to the new stack. The application software in the new stack has to be modified to import this state information. In certain other applications, e.g., a remote desktop session, application software maintains a lot of state that is not externally visible. In such cases, either the application software has to be modified to export and import state or the server has to support storage and retrieval of client-side state. Note that the second scenario is not far-fetched as VNC implementations (e.g., TightVNC[8]) allow client-side state to be stored at the server and restored at a later time.

The FlowSplitter should also export a control interface that allows client-side software stacks or the service provider to configure its behavior.

3.1.2 Client-side Software Stacks

Client-side software stacks contain application-level software and the execution environment for the software. This includes any middleware or systems-level software necessary to satisfy the security and functionality requirements of the application-level software. One should note that the Trusted Platform is a component of all software stacks as it is needed to satisfy security (isolation) and functionality (access to hardware) requirements.

The simplest stack consists of standalone application that uses the Trusted Platform as its execution environment. However, application-level software typically requires higher-level abstractions to access devices, e.g., files, sockets. Hence, we need more complex stacks containing (para)virtualized operating systems running the necessary middleware and the application-level software, e.g., a web browser running on top of X server and paravirtualized Linux.

Client-side software stacks may have to be modified in three ways for use with FlowGuard: First, application-level software has to be programmed to use the proxy server instead of directly contacting the service provider. This is a easily accomplished as most application-level software provide mechanisms to specify the use of a proxy.

Secondly, the application-level software may have to be modified to export and import client state to and from the FlowSplitter. Thirdly, if the software stack wants the ability to configure the FlowSplitter, then the stack needs an additional component that can communicate with the control interface of the FlowSplitter. Note that the third modification is applicable only to trustworthy software stacks.

3.2 Properties of FlowGuard

In this section, we discuss how FlowGuard satisfies the requirements mentioned in Section 2.3.

The first requirement specifies reducing the complexity Trusted Components, i.e., components of the software stack which handles sensitive information. FlowGuard relies on two assumptions to reduce the complexity of these components. First, it assumes that the proxy server does not have to completely understand the information exchange protocol to determine sensitiveness of information. Secondly, it assumes that the functionality requirements of the Trusted Components are lesser than that of a legacy software stack. Taken together, we expect that the resulting Trusted Components will be smaller and simpler than legacy software stacks.

FlowGuard uses legacy software stacks to handle non-sensitive information, thereby maintaining the UI and interface between various components of the system (e.g., system call interface). FlowGuard also uses a proxy server that implements an unmodified application layer protocol between the client and the server, thereby enabling client-side software to communicate with legacy service providers.

FlowGuard allows users (or service providers) to specify mappings between security-sensitive information and software stacks that will handle such information. Doing so provides users control over functionality-security tradeoffs and minimizes the degradation in functionality due to the use of functionally limited Trusted Components.

By using TC principles such as authenticated booting and remote attestation, FlowGuard reduces the possibility that a user or a service provider is tricked into communicating with malicious software that is masquerading as trustworthy software.

We expect that there will be performance degradation due to the introduction of a proxy server. We discuss the performance implications of the FlowGuard architecture for https-based applications later on in the paper.

4. BLAC: A FLOWGUARD FOR HTTPS-BASED APPLICATIONS

In this section we discuss a specific instance of the FlowGuard architecture for https-based applications. First we provide an overview of https-based applications and discuss the problems with Trusted Components in the current software stack. Next, we present BLAC, a FlowGuard for https-based applications. Finally, we discuss each of the components of the BLAC in detail.

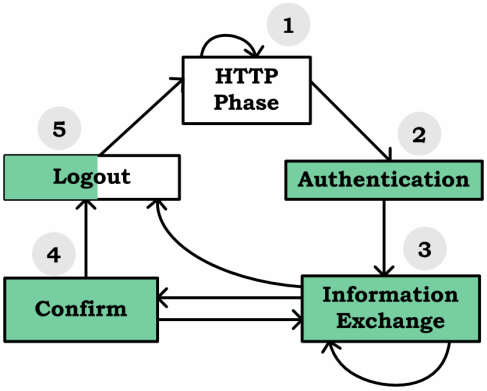

Figure 3. Stages in an https-based application; collectively referred to as a Session. Shaded boxes indicate communication over SSL. Partially shaded boxes indicate transition between SSL and plain-text modes. Numbers indicate sequence of operations.

4.1 Overview of https-based Applications

https is a URI (Uniform Resource Identifier) scheme that augments the HTTP protocol with SSL or TLS and is widely used in many applications such as online banking and electronic commerce. The user, typically a human, interacts with a software stack that consists of a web browser, a window manager and an operating system.

Figure 3 presents the stages in an interaction between a client and a server in a typical https-based application. We refer to the sequence of interactions as a session. Initially, the client and the server employ plain-text, HTTP-based communication. Therefore, information exchange is limited to non-sensitive information in this stage. This stage might be missing in servers that do not support the http URI. The client has to authenticate itself, typically using a login and a password, before it can access sensitive information. The authentication step is carried out using https URLs. The authentication information is encoded in a session identifier (the cookie), thereby enabling the user to access sensitive-data over multiple interactions without having to authenticate repeatedly. Upon successful authentication, the web server and the client exchange information over multiple interactions. At any point during the session, the user is allowed to logout, returning the interaction to unencrypted communication.

Each stage in a session follows the HTTP protocol. The client makes an HTTP request based on previous HTTP responses and/or user input. The server replies with an HTTP response. The client parses this response to generate more requests (e.g., for embedded images) or render it in a viewer. The client now waits for user input to generate a new request. In SSL enabled stages, requests and responses are transmitted over a secure channel (SSL or TLS). The client has to first establish an SSL connection with the web server. The SSL connection could be a new connection or the restoration of a previously suspended connection. Next, the client transmits an HTTP request over the encrypted channel and receives an HTTP response. The SSL connection is then suspended or closed. From HTTP/1.0 onwards, more than one HTTP request and response can be exchanged before an SSL connection is closed or suspended.

Information flow in https-based applications can be classified into three categories: *Non-sensitive*, *Low-sensitivity* and *High-sensitivity* information. Of the three, the first one is accessed using http URIs and the later two are available only through https URIs. This three level classification has justifications in the realm of many online applications such as online banking (described in Section 2.2) and electronic commerce. In electronic commerce, the user is asked to choose shipping address, select a payment method, enter a new payment method (e.g., bank account or credit card information) and confirm the purchase. Clearly, information on a new payment method is more sensitive than shipping address. Since confirming a purchase modifies account status and places a financial burden on the user, we also treat user's choice on confirmation as high-sensitivity data. We argue that the user and the service provider want to use a more trustworthy software stack to handle such high-sensitivity information.

Currently, all information, including non-sensitive information is handled by the same software stack. As mentioned previously (Section 2.1), current software stacks suffer from multiple security vulnerabilities. Using the same software stack to handle various types of information flows increases the stack's vulnerability in three ways: First, since browsers execute security-sensitive functionality in the same address space and protection domain as security-insensitive functionality, vulnerabilities in any part of the browser can be used to comprise security-sensitive functionality. Secondly, as the browser is designed to support multiple sessions, cross-site scripting [37] or cross-site request forgery attacks [15] become feasible. Finally, research has shown that time-shifted attacks are possible on the Mozilla browser and operating system kernels as they store data from security-sensitive sessions, long after the session has ended [17].

The requirements for an effective solution (Section 2.3) are applicable to BLAC. Particularly, we want a solution that reuses the web browser as far as possible given the ubiquity and the functionality of the browser. Similarly, the components of the https URI, the HTTP and SSL protocols, are widely used on the Internet. We cannot expect service providers to switch to a new or augmented protocol immediately. Therefore, BLAC has to confirm to existing protocols.

4.2 Design of BLAC

Figure 4 presents an overview of the BLAC architecture. It is an instance of the more general Figure 1. We use the L4 microkernel [30] as the Trusted Platform. An enhanced https proxy functions as the FlowSplitter. The https proxy uses the microkernel and the execution environment of the microkernel (called L4Env) as its execution environment. We use

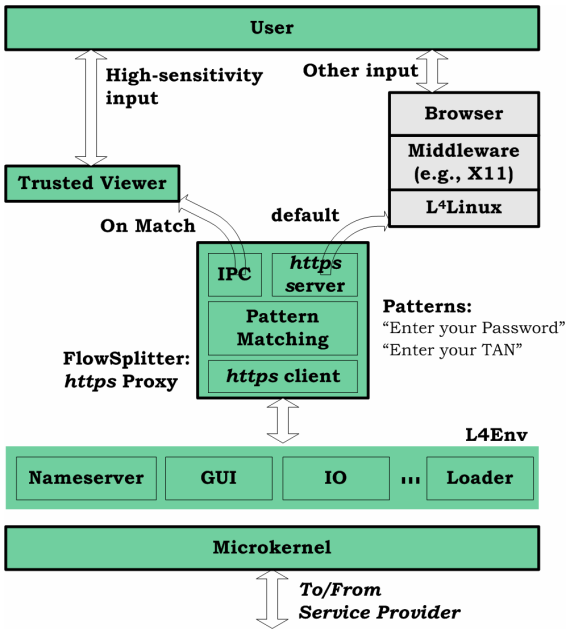

Figure 4. Architecture of BLAC. The https proxy functions as a FlowSplitter. High sensitivity information, as identified by the text patterns, is directed to a small trusted viewer and the rest of the information is directed to a legacy software stack.

two configurations of software stacks: one is an application-level program called the TrustedViewer which acts as a rudimentary browser. The TrustedViewer reuses the execution environment of the https proxy. The second software stack is a legacy software stack that consists of a paravirtualized commodity operating system with a window manager and a full-fledged browser.

We use the TrustedViewer to handle high-sensitivity information and the legacy stack to handle low-sensitivity and non-sensitive information. We feel that this is an appropriate tradeoff between security and functionality. However, our system is easily extensible to support low-sensitivity information processing in the TrustedViewer (Section 4.3.1) or the addition of a new stack to handle low-sensitivity information (Section 4.5).

The https proxy accepts text patterns for identifying sensitive data from the TrustedViewer. https messages (HTTP responses) that contain the text patterns are forwarded to the TrustedViewer and the rest of the messages are directed to the legacy software stack. The proxy also multiplexes connections from the TrustedViewer and the browser to the service provider. We discuss the design of the proxy in detail in Section 4.3.

The TrustedViewer consists of a simple HTML parser and a GUI. It accepts an HTTP request and response and waits for input from user. It then generates the corresponding HTTP request and transfers it back to the https proxy. The TrustedViewer is discussed in Section 4.4.

The legacy software stack, discussed in Section 4.5, consists of a paravirtualized operating system, middleware and a full-fledged browser. Except for modifying the https-proxy settings on the browser, we use an unmodified legacy software stack.

4.3 The *https* Proxy

The https proxy has three functions: multiplex and demultiplex requests and response amongst the multiple software stacks, identify high-sensitivity information in http responses, and finally, accept configuration information from the TrustedViewer.

Since we require our solution to work with legacy service providers, we cannot assume that incoming information will be tagged with sensitivity levels. Therefore, we have to infer sensitiveness of information based on HTTP requests or responses.

4.3.1 Inferring Sensitive Information in https

There are two sources of high-sensitivity information in https-based applications: the service provider and the user. Since the https proxy lies in between the service provider and the client-side software stacks, it can trap any high-sensitivity information originating from the service provider. However, the proxy cannot directly trap or control any high-sensitivity input originating from user. However, the proxy can be programmed to redirect all HTTP responses that request the user to enter high-sensitivity data to the TrustedViewer. To do so, we have to expand the notion of high-sensitivity responses to include even non-sensitive or low-sensitivity responses that lead to high-sensitivity user input.

It is easy for users and service providers to identify high-sensitivity information at the data item level, e.g., password, payment information or a TAN. However, the values for these sensitive data items varies considerably and we cannot expect the proxy to be able to parse every response to determine if a variable sequence of characters or numbers is high-sensitivity data or not. Ideally, we would like to look for static and unique content in HTTP responses to identify responses containing high-sensitivity data. We assume that the user or the service provider has identified *possibly* unique text patterns in HTTP responses that contain high-sensitivity data.

We employ a two stage filtering process on the client-side to accurately identify high-sensitivity responses: first, we use simple pattern matching in the https proxy to trap *all* HTTP responses that contain the text pattern. While this leads to false positives, i.e., a low-sensitivity response is treated as a high-sensitivity response, it is conservative because with proper keywords, we identify all high-sensitivity responses. Next, we employ more complex parsing based on expected HTML page structure in the TrustedViewer (Section 4.4) to identify the high-sensitivity responses among the trapped responses and redirect all false positives back to the untrusted software stack.

Note that in the worst case, if the user or the service provider does not provide any keywords, all responses will have to be trapped and parsed. If the parsing fails, the response is redirected to the legacy browser. While this will increase the response time for the user, it will prevent the flow of high-sensitivity information to legacy or untrusted computing bases.

As an example, while shopping on Amazon.com, user's choice on the confirmation page is high-sensitivity data. This high-sensitivity data is encoded in an HTTP request, which originates from the HTTP response that contains the confirm page. The confirm page contains the string `*alt="Place Your Order"*`. So we trap all responses that contain the above string. In the unlikely case that someone has a product with the same title, that particular product page will also be trapped. However, the confirm-page parser that resides in the TrustedViewer will fail to parse the product page and it will be redirected back to the legacy browser.

4.3.2 Demultiplexing HTTP Responses

Once high-sensitive HTTP responses have been identified, the https proxy must direct them to the TrustedViewer. To do so, the proxy must also transfer client-side state from the legacy client software to the TrustedViewer. This is easy in the case of https-based applications as client-side state is embedded in a cookie. Since the cookie is transmitted with every outgoing message, the proxy has to capture the outgoing message and transfer it to the TrustedViewer along with the high-sensitivity response.

Switching from the TrustedViewer to the legacy client software is easier because the TrustedViewer does not modify client-side state. Since the old cookie is still valid, the https proxy does not have to take any additional steps to maintain client-state coherency between the two software stacks.

4.3.3 Accepting Configurations from the TrustedViewer

The third task of the https proxy is to accept configuration information from the TrustedViewer. To do so, the https proxy exports a control interface that accepts configuration information from the TrustedViewer. As seen in Section 4.3.1, text patterns are the only configuration information needed by the https proxy.

4.4 Trusted Viewer

The TrustedViewer consists of three parts: a configuration file, a DOM tree builder that can be used to extract parser and a GUI to interact with users. The configuration file lists all text patterns that are associated with high-sensitivity information and will trigger the activation of the TrustedViewer. The configuration file varies for each service provider. At startup, the TrustedViewer registers these patterns with the https proxy. Upon receiving a HTTP response which contains any of the keywords, the https proxy forwards the HTTP response and the corresponding HTTP request to the TrustedViewer. Remember that the HTTP request contains the cookie that is necessary for all subsequent interactions.

The second part of the TrustedViewer is a simple parser. The parser contains two sets of configuration information: (a) XPaths [11] in the response HTML document pointing to data that must be extracted and (b) sanity check tuples. A sanity check tuple is of the form <XPath, SanityPattern>. This indicates that the parser must check for the occurrence of the SanityPattern in the given XPath. For example, the confirm page for electronic commerce transactions typically contains standard patterns such as "Shipping Information", "Order Total" in predetermined XPaths.

The parser builds a DOM tree of the HTML document and extracts content from all specified XPaths. At the same time, the parser also checks if the SanityPatterns specified in the sanity check tuple match the extracted content. On success, the parser returns information in a tabular format to the TrustedViewer. If either the extraction fails or SanityPattern matching fails, the parser sends an error message to the GUI component. For more robust parsing, HTML content extraction tools as described in [29] can be employed. If the page cannot be parsed by the TrustedViewer, the user is given an opportunity to cancel the interaction or forward the page to the legacy software stack.

The third part of the TrustedViewer is a GUI component that displays the high-sensitivity information extracted by the parser. The GUI allows users to enter text input and confirm or cancel an interaction. The GUI also contains an option to forward the page to the browser, if the user feels that the page was not properly parsed or if the user feels that the tradeoff between functionality and security has changed.

Adding a new page to the TrustedViewer is easily accomplished. We have developed a script that operates on templates of HTML files that have to be handled by the TrustedViewer. The user clicks on content that she deems sensitive and the script provides the XPath of the content. At this point, the user can also select text patterns to generate sanity check tuples. The user then provides the XPath values along with the text patterns for the page to the TrustedViewer. The TrustedViewer forwards the text patterns to the https parser and uses the XPath and sanity check tuple to parse the new page.

4.5 Client-side Software Stacks

The microkernel, its execution environment and the https proxy are common components of all client-side software stacks. The TrustedViewer along with the above mentioned components forms the client-side software stack that is used to handle high-sensitivity information. The web browser along with the X server, the paravirtualized Linux kernel (L^4Linux) and the common components form the other software stack. Except for modifying the https proxy setting of

the browser, we do not make any modifications to the legacy software stack.

Even though, BLAC uses only two software stack configurations, it does not preclude other client-side software stack configurations. For example, BLAC can handle two L4Linux stacks, one of which is trusted. The difference in trustworthiness could be due to the use of a browser which is executed in safe mode or the L4Linux instance is booted up from a fresh, read-only image. In order to replace the TrustedViewer with a trusted L4Linux stack, we have to make two modifications to the browser. First, as before, we have to change the https proxy settings of the browser and second, we have to modify the browser to *pull* state information from the https proxy. This is easily accomplished by installing a browser extension or a plugin that contacts the https proxy, retrieves cookie information and sets the cookie. If the user desires to use this trusted L4Linux stack to configure the https proxy, we also need to provide a configuration tool (a browser extension or a standalone program), that programs the https proxy with new configuration information. Note that the proxy configuration still remains the same. The new stack is now ready to handle security-sensitive information.

5. EVALUATION

We implemented a prototype of BLAC on top the L4 microkernel. The execution environment of the L4 microkernel, called L4Env, consists of a set of servers providing basic services such as naming, memory and IO management and display manager. Similar components are described in detail in [23]. The https proxy and the TrustedViewer are executed directly on top of the microkernel and L4Env. The browser, Mozilla Firefox, is executed in L4Linux, a paravirtualized operating system, along with the X Server. We modified the proxy settings of the browser, requiring it communicate with the https proxy instead of directly communicating with the service provider.

First, we present a qualitative evaluation of security properties of BLAC. As supporting evidence, we present quantitative analysis of the complexity reductions in BLAC. We show that the size and complexity of the Trusted Components is reduced by an order of magnitude in BLAC. Next, we analyze the performance of BLAC and show that the performance overheads are manageable. Then, we discuss the use of BLAC with real-world service providers and discuss the complications faced during our analysis. Finally, we evaluate BLAC along the lines of requirements mentioned in Section 2.3.

5.1 Security Properties

BLAC improves the security properties of client-side software stack in three ways: First, BLAC switches software stacks depending on the sensitiveness of information being handled. In current systems, the same software stack is used to handle different categories of information flows. Therefore, an attacker can exploit vulnerabilities in handling of non-sensitive information to either access sensitive information (e.g., cross-site scripting attacks [15][37]) or to compromise the software stack (e.g., exploit arbitrary code execution vulnerabilities in browsers to install malicious extensions [12]). Since BLAC separates the handling of different categories of information flow, these attacks will no longer succeed in compromising the flow of sensitive information.

Secondly, BLAC treats the browser and its execution environment (L4Linux and X server) as untrusted components. By preventing the flow of sensitive information to these components, BLAC ensures that vulnerabilities in these components cannot affect the confidentiality and integrity of sensitive information. However, the current BLAC architecture and implementation does not address availability issues. This leaves open the possibility of Denial of Service attacks, e.g., by crashing the network stack, the attacker can prevent the https proxy from communicating with the service provider. We are exploring the use of replication to improve availability properties of BLAC.

Thirdly, BLAC uses a small and simple application and software stack to handle high-sensitivity information. A small and simple application is expected to have fewer bugs and should be more amenable to exhaustive testing or formal verification. We use two well known software metrics: source lines of code (LOC) and McCabe's complexity metric to illustrate the large reductions in complexity in BLAC.

Table 1: Complexity Comparison of BLAC and a Linux-based software stack. The Linux kernel includes networking, file system, IO, memory management support. L4Env includes servers for bootstrapping the system and resource managers for memory and devices, IO manager, window manager and ABI support. The networking stack, which is executed as a separate service, does not have to be trusted in BLAC, resulting in savings of about 170 KLOC.

Component	BLAC			Linux		
	Composition	LOC	MCC	Composition	LOC	MCC
OS	**L4**	14,000	2,300	**Linux Kernel**	383,000	65,000
Middleware	**L4Env**	86,300	11,300	**X Server**	1,015,000	140,300
https Proxy	**https Proxy**	13,600	1,900	**-**	-	-
Application	**Trusted Viewer**	5,000	290	**Mozilla Firefox**	2,208,000	328,300
Total		**118,900**	**15,790**		**3,606,000**	**533,300**

Table 2. Dataset for Measuring Page Access Times. Traces were generated using the WebScarab proxy [3] and the Internet Explorer web browser. Caching was enabled in the browser and the cache was cleaned before generating the trace.

Dataset	Pages	Fragments	Size (KB)	Max. Frags/Page
Amazon	6	52	569	40
Bank-A	14	67	441	20
Bank-B	18	107	931	17
Bank-C	11	119	1433	59

Many software engineering studies have shown that the LOC metric is roughly correlated with the number of defects [43]. MCC is based on McCabe's definition [32] of a control flow complexity metric. It is measured per function and it gives the number of distinct execution paths in a given function. The MCC metric represents the minimum number of tests that need to be carried out on that function to verify control flow properties. Studies have also shown that the MCC metric correlates with the number of defects [34].

We compare the size and complexity of BLAC with a comparable Linux-based software stack. The Linux-based software stack is configured to provide the same functionality as provided by BLAC. Table 1 compares the software complexity metrics of the two approaches. At the application layer, we can see that the size and complexity of BLAC is two orders of magnitude smaller than that of the browser. Comparing the complexity of the full system, we see that BLAC is an order of magnitude smaller than a comparable Linux-based system.

This comparison might seem unfair, especially given that the Mozilla Firefox browser possesses much more functionality than the TrustedViewer. However, one should note that BLAC employs the TrustedViewer to operate on high-sensitivity data. The browser, with its varied functionality, is still available to the user to operate on low-sensitivity or non-sensitive data.

5.2 Performance

The performance of VMMs in general and the L4 microkernel and the paravirtualized L^{4}Linux operating system in specific is well documented [13][24]. Therefore, we focus on the overheads introduced by BLAC. Since the https proxy introduces a layer of indirection between the user and the service provider, we expect performance degradation. This section attempts to quantify the overheads introduced by the https proxy.

We used two identical machines, each with a 2.26 GHz, Pentium-4 processor and 512 MB of RAM. The machines were connected using a 100 Mbps switch. The standard deviation was less than 5% in most experiments and less than 10% in all our experiments. We used https traces from four online sites in our experiments: Bank-A, Bank-B, Bank-C and Amazon.com (Bank names changed for anonymity).

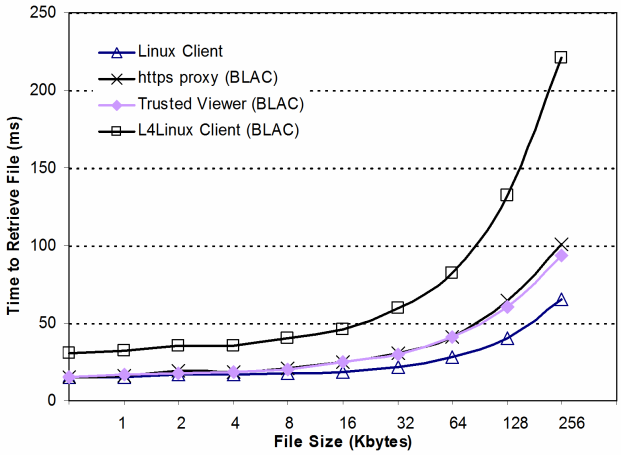

Figure 5. Comparison of File Access Times. Access time increases linearly in all cases.

Table 2 presents a profile of the four data sets. Pages represent the number of complete pages that were in the trace. Each page is the result of a user-initiated HTTPS request. Since a page consists of many fragments, each of which has to be requested separately by the client application, we also list the number of fragments in the trace and the maximum number of fragments per page. In all traces, the login request resulted in the maximum number of fragments requested.

In our experiments, the server side consists of a simple program that reads the traces, accepts HTTP requests over an SSL channel and transmits the appropriate HTTP response over the same SSL channel. We used three client-side configurations. One is a Linux client that connects to the server using SSL and generates HTTP requests. The client reads the complete response and discards it. The second configuration represents a legacy software stack configuration in BLAC. We use the same client as before; however, it runs on L^{4}Linux and connects to the https proxy. For this configuration, we also measured the performance of the proxy, when it contacts the service provider. This allows us to identify the overheads introduced by the proxy. The third client configuration consists of a TrustedViewer using IPC to contact the https proxy and request files from the service provider.

First, we measure the retrieval times for individual files. Figure 5 compares the performance of the various client configurations. Accessing a page in the L^{4}Linux client through the https proxy is about 2 to 2.5 times slower than the original Linux implementation. Regression analysis of data showed that overhead is represented by the equation

$ovhd = 0.535 * file\_size + 17.9$; and $r^2 = 0.998$.

where ovhd is the overhead in milliseconds and file_size is the size of the retrieved file in kilobytes. In absolute terms, the overhead per file is of the order of few tens of milliseconds.

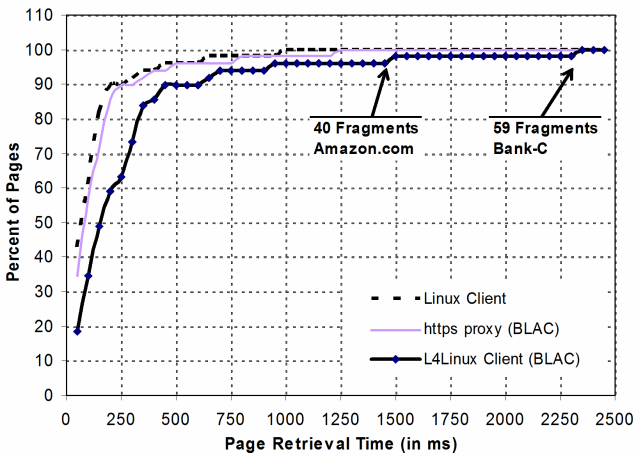

Figure 6. CDF of HTML Page Access Time based on HTML pages from our traces.

Since an HTML page is typically composed of multiple fragments, we also measure "page" access times for the dataset mentioned in Table 1. Figure 6 plots the CDF of page access times for the Linux and the L^{4}Linux client. While the L^{4}Linux client is slower than the Linux client, we see that the page retrieval time is less than 0.5 seconds in 90% of the cases. The pathological cases occur when a page has a large number of fragments, as illustrated in the figure. Even in the worst case, the page retrieval time is around 2.25 seconds. A survey by Jupiter Research and Akamai showed that about 75% of shoppers are willing to wait up to four seconds for a page to load [2]. So, even in the worst case, we are within the threshold for a majority of users.

We would like to note that the performance of our proxy can be improved in many ways. One source of improvement is to implement the HTTP/1.1 protocol in the https proxy that allows multiple requests to be sent over a single SSL connection. This amortizes the overhead of the CONNECT message and SSL connection establishment phase (14.5 ms in our current implementation). Techniques such as VMM-bypass-IO [31] can also be employed to reduce the overhead performing IO in virtual machine based systems.

5.3 Case Studies

Once again, we studied the four web sites mentioned in Table 2. High-sensitivity interactions accounted for two pages in all bank sites: login and confirm page. In Amazon.com, three pages were deemed highly sensitive: login page, confirm page and payment page as the user could enter a new payment method in the payment page. The payment page is a good example of a page where the user might want to transfer a page from the TrustedViewer to the legacy browser, in the case that she is not entering any high-sensitivity information. Compared to the total number of pages in the session, we can see that the TrustedViewer is sparingly used. We successfully tested our complete prototype using Bank-A's demonstration account.

In out analysis of these sites, we came across some cases which cannot be handled by our current implementation. First, if the browser chooses to enable compression, then we are not able to handle HTTP requests and responses. This problem can be solved by adding a compression library to the https proxy or by rejecting compressed requests from the browser.

The second problem is more critical as there are some web service provides who present the login page over an http URL. The login request is still transferred using the https URL. Since our proxy is an https proxy, we cannot currently capture such pages. Once again there are two solutions: one is to use the https URL instead of the http URL while retrieving the login page. This is not a problem as most sites allow http URL content to be served over https URLs. Alternatively, we could add an http proxy to the https proxy. While this will lead to increased overheads, we will be able to capture all requests for high-sensitivity data.

Another problem faced by the current version of BLAC is that SSL and HTTP do not provide support for remote attestation. This opens up the possibility of a malicious, untrusted browser generating an HTTP request that mimics a request generated by the TrustedViewer. For example, in electronic commerce sites such as Amazon.com, the user's choice on Confim/Cancel is high-sensitivity data and this data is encoded in the HTTP request. As there is no secret information involved in generating the request such as a TAN, the browser too can generate similar HTTP requests and modify the state of the user's account. We expect this problem to be alleviated with the use of remote attestation.

5.4 Revisiting Requirements

In Section 2.3, we mentioned five requirements of an effective solution. Of the five requirements, we evaluated complexity reductions and performance overheads in the previous sections. This section evaluates the design and implementation of BLAC with respect to the other three requirements.

R2. Reuse Legacy Software and Protocols. BLAC reuses unmodified (excluding proxy settings), legacy software to handle non-sensitive and low-sensitivity information flows. The https proxy in BLAC does not require any modifications to either SSL or the HTTP protocol. Hence, BLAC works with legacy service providers. However, if service providers need information on client-side software stack, then they must be modified to use remote attestation features.

R3. Flexibility in Tradeoffs. In our implementation, BLAC uses a very small Trusted Viewer, which is functionally limited, to operate on high-sensitivity data. However, BLAC provides the user with flexibility in determining the stack to use for processing various categories of information. If the user is confident about the trustworthiness of a legacy software stack, she can modify the https proxy settings to direct high-sensitivity information to the legacy software stack

handling non-sensitive and low-sensitivity information. Alternatively, the user can also create a new instance of a modified legacy software stack, as discussed in Section 4.5, to handle high-sensitivity data. In this case, the user retains most of the functionality but the gains in software complexity no longer apply.

R4. Protection against Spoofing Attacks. As mentioned in Section 3.1, we rely on Trusted Computing hardware support to determine the software stack in use. The execution environment of BLAC provides support for non-forgeable trust indicators, e.g., portion of screen or external LEDs, that communicates the trustworthiness of the software stack to a human user. We assume that the end-user is reasonably aware of trust indicators and will make use of the indicators to protect sensitive information. We do not address the human aspects of deception and spoofing here.

6. DISCUSSION

6.1 Applicability to other VMM-based approaches

We implemented BLAC on a microkernel platform. The FlowGuard approach can be applied on other platforms, including commodity operating systems. In any platform, the first requirement is that the FlowSplitter and each of the client-side software stacks need to be executed in different protection domains to isolate the flow of sensitive information. Second, each of the software stacks has to be able to communicate with the FlowSplitter.

Environments such as Xen [13], Asbestos [20], Linux and Microsoft Windows satisfy both above requirements. A Xen-based implementation, for example, would have the FlowSplitter and each of the client-side software stacks executing in a separate VM. They can communicate with each other using network stacks. In a Linux-based implementation, the FlowSplitter and each of the client-side software stacks (browsers or User Mode Linux instances) are executed as different users and they communicate using the network stack or more efficient IPC such as shared memory.

6.2 Applicability to other client-server applications

FlowGuard can be applied to other client-server applications beyond https. The design and implementation of FlowGuard contains two important aspects: (a) inferring sensitiveness of information and (b) transferring client-side state between the various software stacks. As described in previous section, for https-based applications we used a set of text patterns to identify high-sensitivity information and relied on externally visible cookie information to transfer state between software stacks. In this section, we briefly discuss potential designs that address the above aspects when using other protocols, including a POP3-based mail application and a remote terminal.

The FlowSplitter for a POP3-based application [33] would implement a POP3 proxy and handle incoming connections from multiple software stacks. Since authorization information (e.g., login and password) is sensitive, the end-user has to employ a trustworthy software stack for the authorization step. After this, the POP3 proxy will have to handle the RETR and the DELE commands from the mail clients. These commands are used to retrieve and delete mail respectively. Sensitive information for responses to the RETR commands can be easily identified by scanning mail headers or mail senders or even mail content. Sensitive mail is then redirected to trustworthy software stacks. However, the POP3 protocol uniquely numbers messages in the mailbox for every session and all commands are issued with the unique number as a reference. Hence, the FlowSplitter will have to forward dummy mail to stacks that are not trustworthy enough to receive the sensitive mail. Handling the DELE command is tricky, as malicious software stacks can issue arbitrary DELE commands and compromise the integrity of mail. To solve this problem, the FlowSplitter can maintain a per-session list of valid mail handled by each software stack and disregard all DELE commands on dummy mail.

If there are no other ways to distinguish security-sensitive information, the FlowSplitter may have to refactor the application level software to explicitly identify security-sensitive information. As an example, consider a FlowGuard for a command line shell (e.g., bash over SSH). Due to the generic nature of a shell, sensitive information comes in formats as varied as applications handled by the shell, e.g., input to *passwd* command, contents of a file, directory listings, etc. Consequently, the FlowSplitter for the shell would need to add the code to distinguish sensitive information for each application to be protected. One potential solution is to use shell commands to identify the sensitiveness of data processed by each shell command and redirect the response to the security-sensitive software stack. Since shells do not have client-side state, switching between software stacks can be easily accomplished.

6.3 Server-Side Support

One of the key requirements for FlowGuard (and BLAC) was support for legacy service providers. This complicates our design and implementation of FlowGuard, e.g., we have to indirectly infer the sensitiveness of information in the responses. With server-side support, we can simplify the Trusted Components in FlowGuard.

First, the service provider could label information in its responses with sensitivity levels. In HTTP responses, this means adding an additional field to the response header: Sensitivity= [*Non* | *Low* | *High*]. This step would reduce the need for parsing of responses in the FlowSplitter. However, to support user driven configurability, we would still have to rely on indirect methods for inferring sensitiveness of responses.

On a related note, if the service provider tags some responses as High-sensitivity, it could rely on a simple representation format (e.g., tabular format for high-sensitivity pages in BLAC) or provide template files for extracting relevant data from the responses. This would simplify the application-level Trusted Components.

Finally, the service provider can provide a mechanism to save and restore client-side state, ala VNC implementations. This would enable easy, albeit expensive, transfer of state between client-side software stacks.

7. RELATED WORK

As mentioned in Section 2.1, Proxos [46], Terra [22] and AppCores [44] are closely related to FlowGuard, in that they advocate the use of small and simple TCBs or application software. Efforts have also been directed towards building small operating system kernels [13], VMMs [30], TCBs [26][27][35] and system services [14][28][36][42]. FlowGuard reuses existing work on constructing small and simple TCBs and application-level software and middleware. FlowGuard combines this approach with a proxy-based system to reuse legacy software whenever possible.

Mandatory access control systems such as Asbestos [20], capability-based system such as EROS [41] and Authority Based Access Control systems such as Polaris [45] aim to control the flow of information or minimize the damage due to compromise of software handling sensitive information. Our work on FlowGuard is orthogonal to these approaches. In fact, by splitting the flow of information in a single session amongst multiple software stacks, FlowGuard allows for more rigorous enforcement of access control. For example, browsers have to be allowed to connect to the network, opening up avenues for information leaks. On the other hand, the TrustedViewer in BLAC is not expected to communicate over the network; it only needs to access the https proxy and a limited number of system services such as the display manager. Hence, the access rights of a TrustedViewer can be constrained to a greater extent than those of a full-fledged browser.

There is also considerable work on using static analysis to find vulnerabilities in software, e.g., [19][48]. Castro et al. [16] use static analysis to instrument loads and stores in programs to maintain the integrity of data flow. While these techniques detect vulnerabilities or protect against a major class of vulnerabilities (data corruption attacks), their scalability, especially to software as large and complex as a browser, is still an open question. FlowGuard attempts to reduce the size and complexity of software that needs to be trusted, enhancing the effectiveness of static analysis techniques. Language based information flow analysis tools [40] together with OS level mandatory access controls have been employed to provide fine-grained control over flow of information [25]. However, these techniques are not applicable to a large number of programs written in weakly-typed languages such as the C programming language. XFI [21] takes a novel approach by using an untrusted rewriter to instrument an extension to a software program with control flow guards and memory access guards. The guards are then verified with a small trusted verifier and the extension is now deemed safe for execution. Such techniques can be applied to protect software such as the browser from malicious extensions. However, the original software program can still be exploited if it contains vulnerabilities.

Lastly, given the large number of vulnerabilities in the browsers, there is a lot of work addressing vulnerabilities in the browser. Research efforts range from interface personalization to thwart spoofing attacks [47], password hashing using browser extensions to limit the damage done by leaking of passwords [39], rewriting scripts in HTML pages to prevent them from exploiting known vulnerabilities [38], and trust indicator extensions for browsers to prevent interface spoofing and phishing attacks [50]. All these systems assume that the browser can be trusted with high-sensitivity information. Given the large number of vulnerabilities (average of 2 per month [4][5]) and the types of vulnerabilities (arbitrary code execution, security-system bypass) in current browsers, attackers can exploit them to bypass the protection afforded by many of these systems.

The Tahoma architecture [18], on the other hand, assumes that the browser cannot be trusted. Tahoma proposes executing browser instances in separate VMs, similar to VMWare's Browser Appliance [10]. Additionally, Tahoma requires service providers to define a manifest, which is used to control behavior of the browser. Tahoma does not address the issue of a multitude of browser extensions (e.g., weather forecast extensions) that periodically talk to a site outside the service provider's manifest. FlowGuard also treats the browser as untrusted but it prevents the flow of sensitive information to the browser, instead of controlling the behavior of a browser instance that has access to sensitive information. Therefore, it does not have to curtail the browser's functionality.

8. CONCLUSION

Current client-server applications use the same client-side software stack to handle information with differing security and functionality requirements. This has resulted in large and complex software, with multiple security vulnerabilities. However, current software and interfaces are too widely used to be abandoned altogether. We presented a proxy-based approach called FlowGuard to address the problem of large and complex client-side software stacks. The main component of FlowGuard is a proxy that uses mappings from sensitiveness of information to trustworthiness of software stack to demultiplex responses from the service provider amongst multiple client-side software stacks. By employing a legacy software stack as the untrusted software stack and a small and simple trusted software stack allowed FlowGuard to reduce the complexity of Trusted Components

while at the same time allowing the reuse of the legacy software stack as much as possible. FlowGuard also provided mechanisms to allow users and service providers to determine the software stack the mappings, thereby providing flexibility in determining the appropriate functionality-security tradeoffs.

We demonstrated the feasibility of our approach by implementing a FlowGuard for https-based applications that was successful in interacting with a real-world bank's web site. Our evaluation showed that we were able to reduce complexity by over an order of magnitude, while limiting overheads to few tens of milliseconds per HTTP response.

We also showed that FlowGuards can be ported to other client-server applications and implementation platforms. We also discussed how FlowGuard can be further simplified with service provider support.

9. REFERENCES

[1] Anonymized.
[2] Jupiter Research. Retail Web Site Performance: Consumer Reaction to a Poor Online Shopping Experience. http://www.akamai.com/4seconds
[3] OWASP Web Scarab Project. http://www.owasp.org/index.php/Category:OWASP_WebScarab_Project
[4] Secunia. Vulnerability Report – Microsoft Internet Explorer 6. http://secunia.com/product/11/?task=advisories
[5] Secunia. Vulnerability Report – Mozilla Firefox 1.x. http://secunia.com/product/4227/?task=advisories
[6] Secunia. Vulnerability Report – X11 Windowing System (X11) 6.x. http://secunia.com/product/3913/?task=advisories
[7] Secunia. Vulnerability Report – Linux Kernel 2.4.x. http://secunia.com/product/763/?task=advisories
[8] TightVNC. http://www.tightvnc.com/download.html
[9] Trusted Computing Group. *TCG Main Specification v1.1b*, https://www.trustedcomputinggroup.org/
[10] VMware, Inc. Browser appliance virtual machine. http://www.vmware.com/vmtn/vm/browserapp.html
[11] W3C. XML Path Language. http://www.w3.org/TR/xpath
[12] J. Bambenek, SANS Institute. BHO scanning tool and New Scam Targets Bank Customers. http://isc.sans.org/diary.php?date=2004-06-29.
[13] P. Barham, et al. Xen and the Art of Virtualization. In *Proc. 19th ACM SOSP 2003*, NY, Oct. 2003.
[14] D. Brumley, D. X. Song. Privtrans: Automatically Partitioning Programs for Privilege Separation. In *Proc. USENIX Security Symposium,* San Diego, USA. Aug 9-13, 2004.
[15] J. Burns. Cross Site Reference Forgery: An introduction to a common web application weakness. http://isecpartners.com/documents/XSRF_Paper.pdf
[16] M. Castro, M. Costa and T. Harris, Securing Software by Enforcing Data-flow Integrity, In *OSDI 2006*, Seattle, Nov 2006.
[17] J. Chow, et al., Understanding Data Lifetime via Whole System Simulation. In *Proc. USENIX Security 2004*, pp 321-336, Aug. 2004.
[18] R.S. Cox, S.D. Gribble, H.M. Levy, and J.G. Hansen, A Safety-Oriented Platform for Web Applications, In *IEEE Symposium on Security & Privacy*, pp. 350-364, 2006.
[19] D. Engler, D. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *18th SOSP.* Banff, Canada, Oct. 2001.
[20] P. Efstathopoulos et al., Labels and Event Processes in the Asbestos Operating System. In *20th ACM SOSP*, Brighton, UK, October 2005.
[21] Ú. Erlingsson et al., XFI: Software Guards for System Address Spaces, In *OSDI 2006*, Seattle, Nov 2006.
[22] T. Garfinkel, et al. Terra: A virtual machine-based platform for trusted computing. In *Proc. of the 19th SOSP*, October 2003.
[23] H. Härtig, et al. The Nizza Secure-System Architecture. In *IEEE CollaborateCom 2005.* San Jose, Dec 2005.
[24] H. Härtig, M. Hohmuth, J. Liedtke, S. Schönberg, and J.Wolter. The performance of µ-kernel-based systems. In *Proc. 16th ACM SOSP*, pp 66–77, Oct. 1997.
[25] B. Hicks, S. Rueda, T. Jaeger, and P. McDaniel. Integration of SELinux and security-typed languages. In *Proc. of the 2007 Security-Enhanced Linux Workshop*, March 2007.
[26] Hohmuth, M., M. Peter, H. Härtig, and J. Shapiro. "Reducing TCB size by using untrusted components – small kernels versus virtual machine monitors", in *Proc. of the 11th ACM SIGOPS European Workshop*, Leuven, Belgium, 2004.
[27] T. Jaeger, R. Sailer, and X. Zhang, Analyzing Integrity Protection in the SELinux Example Policy, in *12th USENIX Security Symposium*, Washington D.C. USA, Aug. 2003.
[28] D. Kilpatrick, Privman: A Library for Partitioning Applications. In *USENIX Annual Technical Conference, FREENIX Track 2003*, pp 273-284. San Antonio USA, July 2003.
[29] A.H.F. Laender, B.A. Ribeiro-Neto, A.S. da Silva and J.S. Teixeira, "A brief survey of web data extraction tools", In *SIGMOD Rec.*, 32(2), pp 84-93, 2002.
[30] J. Liedtke, On Micro-Kernel Construction, In *15th ACM SOSP*, Copper Mountain Resort, Colorado, USA. Dec. 1995.
[31] J. Liu, et al. High Performance VMM-Bypass I/O in Virtual Machines. In *USENIX ATC* 2006, Boston, MA, May 2006.
[32] T.J. McCabe, A Complexity Measure, *IEEE Transactions on Software Engineering*, SE-2 No. 4, pp. 308-320, Dec. 1976.
[33] J. Myers, M. Rose, Post Office Protocol - Version 3. *RFC 1939.* May 1996.
[34] N. Nagappan, T. Ball and A. Zeller, Mining Metrics to Predict Component Failures, In *ICSE 2006*, Shanghai, Nov. 2006
[35] B. Pfitzmann, J. Riordan, C. Stüble, M. Waidner and A. Weber. The PERSEUS System Architecture. *Research Report. IBM Research Division.* RZ 3335. Sept. 2001.
[36] N. Provos, M. Friedl, and P. Honeyman. Preventing privilege escalation. In *12th USENIX Security Symposium*, Washington D.C, Aug. 2003.
[37] J. Rafail, Cross-Site Scripting Vulnerabilities. http://www.cert.org/archive/pdf/cross_site_scripting.pdf
[38] C. Reis, J.Dunagan, H.J. Wang, O. Dubrovsky, and S. Esmeir, BrowserShield: Vulnerability-Driven Filtering of Dynamic HTML, In *OSDI 2006*, Seattle, WA
[39] B. Ross, et al., Stronger Password Authentication Using Browser Extensions. In *14th Usenix Security*, Baltimore, 2005.
[40] Andrei Sabelfeld and Andrew C. Myers, Language-Based Information-Flow Security. In *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, Jan. 2003.
[41] J. S. Shapiro, J. M. Smith, and D. J. Farber. EROS: A Fast Capability System. In *Proc. 17th ACM SOSP*. Charleston, SC, USA. Dec. 1999.
[42] J. S. Shapiro et al., Design of the EROS Trusted Window System, In *Proc. USENIX Security Symposium*, San Diego, 2004

[43] V. Y. Shen, T. Yu, S. M. Thebaut, and L. R. Paulsen, Identifying Error-prone Software -- An Empirical Study, In *IEEE TOSE*, Vol. SE-11, pp. 317--323, April 1985.
[44] L. Singaravelu, C. Pu, H. Haertig, C. Helmuth, Reducing TCB Complexity for Security-Sensitive Applications:Three Case Studies, In *Proc. First Eurosys*, Leuven, Belgium, April 2006.
[45] M. Stiegler, et al., Polaris: virus-safe computing for Windows XP, In *CACM*, 49(9), pp 83-88, 2006
[46] R. Ta-Min, L. Litty and D. Lie. Splitting Interfaces: Making Trust Between Applications and Operating Systems Configurable. In *Proc. of the 7th USENIX OSDI*, pp. 279-292. Nov. 2006.
[47] J. D. Tygar and A. Whitten. WWW electronic commerce and Java Trojan horses. In *Proc. of the 2nd USENIX Workshop on Electronic Commerce*, Nov. 1996, pp. 243-250.
[48] D. Wagner, et al.. A first step towards automated detection of buffer overrun vulnerabilities. In *NDSS*, 2000.
[49] D. Wheeler.SLOCCount.http://www.dwheeler.com/sloccount
[50] Y. Zhang, et al. Phinding Phish: Evaluating Anti-Phishing Tools. In *14th NDSS*, San Diego, CA, 2007.