Mending Broken Trust: Ensuring Privacy and Integrity Online

By

MATTHEW D. VAN GUNDY
B.S. (University of California, Santa Barbara) 2005
M.S. (University of California, Davis) 2011

DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

Computer Science

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

Hao Chen (Chair)

Felix Wu

Matt Franklin

Committee in Charge
2014

To Jessi. Your love and encouragement are my strength.

Contents

Matthew D. Van Gundy
June 2014
Computer Science

Mending Broken Trust: Ensuring Privacy and Integrity Online

Abstract

Common Internet protocols fail to meet users' reasonable security expectations in a number of subtle ways. In this work we address three major issues in online communication privacy and integrity: software integrity in web applications, secure multi-party instant messaging, and consistency in distributed protocols subject to Byzantine failures.

To protect users from malicious websites, modern web browsers enforce isolation between potentially-malicious code from different sources. Even with perfect isolation, a web server which unintentionally serves malicious code, known as Cross-Site Scripting (XSS), allows attackers to take full-control of the web application's client-side interface. Previous XSS defenses primarily targeted only the server-side or the client-side, leading to a semantic gap. To address this problem, we created Noncespaces, an end-to-end system that allows web servers to reliably identify untrusted content so that browsers can enforce flexible security policies, neutralizing XSS vulnerabilities.

Many other online communication mediums also suffer from confidentiality and integrity problems. Instant Messaging (IM), another popular method of communication on the Internet, mimics impromptu face-to-face conversation. However, nearly all IM protocols fail to provide either confidentiality, end-to-end origin authentication, or deniability. Off-the-Record Messaging provides a solution for two-party conversations, but it does not generalize to conversations of three or more parties. To provide secure IM for privacy-conscious users, we propose Multi-party Off-the-Record Messaging (mpOTR). mpOTR provides confidentiality, end-to-end origin authentication, and deniability for conversations between an arbitrary number of parties.

Though mpOTR improves security guarantees for multi-party IM, dishonest users may violate consistency between correct users undetected until the chat session ends. Any distributed system which seeks to ensure causal consistency and liveness in a Byzantine environment faces similar challenges. Most existing protocols only provide guarantees when Byzantine failures do not occur; or they sacrifice consistency, liveness, or both when too many Byzantine failures occur. Either alternative is a poor fit for peer-to-peer systems that require consistency and liveness but cannot bound the number of Byzantine failures. To address this issue, we propose OldBlue, a broadcast protocol which ensures causal consistency and liveness between connected correct processes even when an arbitrary number of Byzantine failures occur.

Acknowledgments and Thanks

I would like to thank my advisor, Professor Hao Chen, for his continual support on my academic journey. You always encouraged us to aim high and pursue our own path.

Yuan, Francis, Justin, Juan, Liang, Gabriel, Ben, Clint, and all of my other labmates in the Security Lab, your collaboration, critical feedback, and camaraderie were easily the most enriching of my graduate school experiences.

I believe that I am much richer for the wisdom and guidance of Professor Zhendong Su, who was my unofficial co-advisor in the beginning of my graduate career. I would also like to thank Sean Peisert, Jeff Rowe, and the other security lab faculty for selflessly offering time and advice to help move us past roadblocks and mature in the field.

On many a week, the highlight of my time on campus was Hapkido class with Dr. Rick Schubert and Jenny McCabe. Thank you for your generous devotion to your students. UC Davis is so fortunate to have you.

I'd like to thank my wife, Jessi, for putting up with far too many late nights and weekends filled with school work. Thank you for your patience and for prying me away from the keyboard from time to time in order to enjoy the world around us.

And last, but certainly not least, I would like to thank my parents, Sam and Sharon Van Gundy. You taught us to try our hardest and we always knew that you'd love us just as much regardless of the outcome. I owe no small measure of success to your unwavering support. Thank you for all the sacrifices you have made to give us every advantage in life.

Coauthor Acknowledgements

Chapter 2 describes work done in collaboration with my advisor, Hao Chen. Portions of this chapter can also be found in the following papers [109, 110]. I would also like to thank: Francis Hsu for his assistance with figures and proofreading and Zhendong Su and his research group for critical input during the early stages of this work.

Chapter 3 describes work done in collaboration with my primary co-author Berkant Ustaoğlu, Ian Goldberg, and Hao Chen. Portions of this chapter were published in [43]. I would like to thank Mark Gondree, Matt Franklin, Matt Bishop, Zhendong Su, and Phillip Rogaway for their feedback during the early stages of this research.

Chapter 4 describes work done in collaboration with Hao Chen.

Chapter 1

Introduction

In times past, it was relatively straightforward to adopt habits to protect personal privacy and property. For someone to rob you, profile you, or surveil you, they had to be physically proximate and specifically target you. Individuals could protect their privacy by being mindful of where they were, what they said, and to whom. Individuals could protect their property by moving to safe localities and being careful about where they traveled. It was unthinkable that people could be deprived of their property and privacy by someone in a far-removed geographic location.

The advances in computing power and connectivity that characterize the Internet Age have enabled communication and commerce to be conducted across vast distances with ease. Voice-Over-IP (VOIP), Instant Messaging (IM), and email replace in person conversation and traditional postal services by providing real-time long-distance conversation and nearly-instantaneous document delivery. The World Wide Web provides a ubiquitous platform where individuals can manage their finances, engage in commerce, read news from diverse sources, and communicate with others all over the world. It should come as no

surprise that the Internet has come to play a very important role in business and personal life in the modern world.

The gifts of the Internet Age are not without their drawbacks. Advances in computation and communication efficiency can also be harnessed by miscreants to extend their sphere of influence and capabilities. Our exposure to those who wish to abuse and defraud us is increasing as more aspects of our daily lives are conducted online. Automation and system monoculture allow large-scale security compromises in the form of Internet worms, viruses, and botnets. Large-scale data aggregation allows wholesale compromise of the financial information of millions of people [58, 44, 54] and facilitates widespread profiling, location tracking, and surveillance [4, 27, 79]. Modern computing systems are sufficiently complex and opaque to the average user that the online analogues to the intuition and audiovisual cues that have kept them safe in the physical world may not be obvious. This makes designing information systems that meet the reasonable security expectations of average users a nuanced and challenging task.

When a user visits a bank web site over a secure connection, they should not have to worry that visiting the page gives an attacker access to their account, causes confidential information on the page to be leaked, or that the content they receive has been modified by an external attacker. Yet Cross-Site Scripting, an incredibly common web application vulnerability, allows all of these things to occur. When users engage in group discussions online, they should not have to worry that it is being recorded by an outside party, that an attacker is impersonating a legitimate discussant, that later, a rogue conversation member will be able to prove their authorship of statements made in confidence to an outside party, or that they may be basing their understanding and actions on a different version of the story than other parties to the same conversation. Yet common communication protocols

currently in use fail to provide one or more of the above properties. In this work we seek to restore some of the trust that users should be able to have in their information systems by addressing each of the above issues.

In Chapter 2, we address the problem of Cross-Site Scripting vulnerabilities in web applications. The web has become a ubiquitous platform for user-centric Internet applications. However, the trustworthiness of web applications is being undermined by Cross-Site Scripting (XSS) vulnerabilities which allow an attacker to inject malicious code into pages served by a trusted web server. Web browsers will run the injected code with the same permissions as all other client-side code from the same server allowing the attacker to modify the content displayed to the user, read all content served to the user, and perform any action in the web application which the user is authorized to perform. Historically, web browsers are very forgiving. When presented with malformed content, web browsers use ad hoc error recovery algorithms to derive a "reasonable" parse of the content. Previous XSS defenses were primarily server-side-only or client-side-only leading to a semantic gap between client and server. Only the server is in a position to distinguish between trusted and untrusted content leaving client-side-only mechanisms to guess the trustworthiness of content provided by the server. Only the browser is in a position to know how it will ultimately parse content leaving server-side-only solutions to guess the way that various browsers will interpret the content they serve. To address these issues, we developed Noncespaces, an end-to-end system to neutralize XSS vulnerabilities. With Noncespaces the server automatically identifies untrusted content and annotates all content with a trust label before sending it to the browser. The browser uses a server-specified policy which governs the capabilities of content in each trust class to ensure that untrusted content cannot perform unauthorized actions. To demonstrate the applicability of Noncespaces to existing

web applications, we modified two popular web applications to use Noncespaces and performed an extensive security evaluation to validate the ability of Noncespaces to protect confidentiality and integrity on the web by preventing XSS attacks.

The web is just one of many online communication mediums that suffers from confidentiality and integrity violations. Instant Messaging (IM), another popular method of communication on the Internet, mimics impromptu face-to-face conversation. However, most popular IM protocols do little to protect against eavesdropping and impersonation. Typical solutions for secure electronic communication use encryption and digital signatures which provide three properties: confidentiality, authenticity[1], and non-repudiation. Confidentiality and authenticity are precisely the properties we desire to prevent eavesdropping and impersonation. However, non-repudiation refers to the receiver's ability to *prove* to an outside party that a specific user authored a given message. This is undesirable in many circumstances, it is precisely the property that whistle-blowers, dissidents, informants, and journalists often wish to avoid. Off-the-Record Messaging (OTR), presented by Borisov, Goldberg, and Brewer [13], provides confidentiality and authenticity for instant messages between two parties while avoiding non-repudiation by using message authentication codes. Message authentication codes give each party equal power — preventing cryptographic proofs that a given user, and not the other, authored a given message. However, it is not straightforward to extend OTR to the chatroom environment where three or more individuals participate in a group conversation. With three or more parties a message authentication code only ensures a message recipient that the sender was a member of the chatroom but it does not prevent one chatroom member from impersonating another.

Chapter 3 presents Multi-party Off-the-Record Messaging (mpOTR), a protocol which

[1] Authenticity encapsulates two integrity properties: origin authentication which prevents impersonation, and message authentication which prevents unauthorized modification of messages.

provides confidential, authenticated, and deniable instant messaging for online chatrooms of three or more users. Confidentiality with perfect-forward secrecy is achieved using standard cryptographic constructs which ensure that past conversations remain confidential even if participants' long-lived private keys are compromised at a later time. Authenticity is ensured by using per-session signature keys allowing the recipient of a message to detect attempts to impersonate or modify messages sent by an honest chatroom participant. Deniability is based on a user's ability to disassociate from the signing key pair used in a session. The ability to repudiate your willingness to participate in a conversation is very important; in some cases revealing who you are willing to speak to may be nearly as damaging as revealing the contents of the conversation. [45, 47, 38] mpOTR is carefully structured to allow an outsider, without knowledge of the private key of any chatroom participant, to forge a cryptographically valid mpOTR transcript. This provides a basis for plausible deniability: because anyone can forge a cryptographically valid transcript without knowledge of the participants' private keys, even a judge with access to the participants' private keys has no algorithmic justification for accepting any transcript as authentic.[2] mpOTR achieves a stronger deniability property than OTR insomuch that each OTR session results in a cryptographic proof, which can be verified by an outside judge, that a participant was willing to hold a conversation with some other user.

Dishonest mpOTR participants may fail to send a message to some participants or may send conflicting messages to disjoint subsets of participants, network or chat server failures (whether benign or malicious) may partition the chatroom for arbitrary lengths of time, and malicious participants may collude to deny service to the rest of the chatroom. Ensuring consensus — that all honest participants agree on what was said — in a multi-party protocol

[2] A forged transcript cannot fool an active participant of the chatroom, but, after the fact, a forgery it is indistinguishable from an authentic chat transcript containing the same plaintexts.

where any number of participants may be working to violate consensus for their own ends is a non-trivial task. Therefore, mpOTR only provides a binary confirmation of consensus at the end of each session. When a chat session ends participants learn whether or not all participants agreed on the contents of the chat transcript.

Obviously, waiting until the end of a chat session to detect violations of consensus is less than ideal. Ideally, to prevent misbehavior and meet users expectations, mpOTR should provide a number of additional consensus and availability properties. At each point in time during the conversation, honest participants should be able to determine a useful lower bound on consensus over the chat transcript up to that point. Messages should be delivered in causal order to ensure that every reply is only delivered after all of its antecedents. During a network partition, participants within a connected component of the chatroom should be able to continue communicating with one another. When a network partition is healed, participants on both sides should be able resynchronize and reach a consistent state. And, to keep dishonest participants from preventing honest participants from communicating through resource starvation, fair scheduling should be employed to ensure that honest participants continue to grant each other a fair share of network resources. These availability and consistency properties are not unique to mpOTR, they are the guarantees provided by any highly-available causally-ordered broadcast protocol that remains secure in an environment where an arbitrary number of Byzantine failures[3] may occur.

Consensus in a Byzantine environment is a long-studied problem typically solved using Byzantine Agreement [55, 57, 23] or Reliable Broadcast [21, 86, 6, 40, 46] protocols. However, all Byzantine Agreement, Consensus, and Reliable Broadcast protocols sacrifice

[3]In contrast to a benign (or halting) failure where failed processes simply stop communicating, under a Byzantine failure failed process may behave in an arbitrary and malicious manner.

either consistency, availability, or both during network partitions. This is not accidental. For example, in Byzantine Agreement protocols, a sender sends a message m and all correct processes must deliver the some message m′. If the sender was correct m′ must be equal m. Suppose there are only two correct processes $\hat{A}$ and $\hat{B}$ on opposite sides of a network partition. $\hat{A}$ sends a message m. For availability to be preserved, $\hat{A}$ must be able to deliver m. However, for Byzantine Agreement consistency to be preserved, $\hat{B}$ must also deliver m. But, due to the network partition, there is no way for m to be communicated to $\hat{B}$. Therefore, either availability or consistency must be sacrificed. This is a fundamental limitation for distributed systems which attempt to ensure strong consistency properties such as Byzantine Agreement and Consensus. The CAP Theorem [19, 42] states that, during a network partition, a distributed system must choose between maintaining availability or consistency[4]. It cannot preserve both properties simultaneously.

In Chapter 4, we present OldBlue, a broadcast protocol which maintains availability and causal consistency between correct connected participants during network partitions. In contrast to previous Causal Broadcast protocols which tolerate only benign failures [68, 3, 82, 40, 11, 84], OldBlue can tolerate an arbitrary number of Byzantine failures. OldBlue provides an incremental guarantee of consensus. When an honest participant $\hat{A}$ delivers a message m sent by another honest participant $\hat{B}$, OldBlue ensures that $\hat{A}$ and $\hat{B}$ agree on all messages sent or received by $\hat{B}$ before m was sent. Availability during network partitions allows correct connected participants to continue to send and receive messages with one another. Cooperative retransmission is used to allow participants to converge toward a consistent shared state when recovering from lost messages or network partitions. And,

[4]Specifically, [42] proves the impossibility of preserving both atomic consistency and availability. However, more recent work [61] proves the impossibility of maintaining availability and a number of weaker consistency properties during a network partition.

OldBlue employs fair scheduling and other mechanisms to prevent dishonest participants from causing starvation among honest participants. We provide formal definitions of the properties that OldBlue provides, prove that OldBlue ensures those properties, and simulate OldBlue with groups of various size to better characterize its network performance.

Finally, Chapter 5 concludes and discusses directions for future work.

Chapter 2

Noncespaces: Using Randomization to Defeat Cross-Site Scripting Attacks

2.1 Introduction

In the years since its humble beginnings as a mechanism to publish and link static documents, the World Wide Web has become a ubiquitous platform for delivering rich Internet applications. On the web users can: conduct research, shop for retail goods, manage their finances, and communicate and collaborate with others through diverse venues from open forums to private messaging. Due to the great monetary and psychological value of these activities, web applications must provide their users with basic confidentiality and integrity guarantees. When a user reads a forum, they should not need to worry that posters to the form will be able to read their private email. When a user shops for a product, they should not have to worry that reviewers of the product will be given access to their bank or stock accounts. Modern web browsers are fundamentally multi-tenant — code from multiple

different web applications can run within the same browser session. In the absence of sufficient isolation, client-side code from malicious web sites could compromise the confidentiality and integrity of data from other sites.

To protect against malicious sites, web browsers adopt a share-nothing sandbox policy that prevents a web application from accessing client-side data belonging to other web applications. This mechanism, known as the Same Origin Policy [91], distinguishes web application code based on its source or *origin*[1]. When code from one web page requests to access code or data in another web page, the web browser checks to see if the requester and the target are both from the same origin. If the origins match, the access is permitted, if their origins differ, access is denied. Barring other vulnerabilities within the browser, if an attacker wishes to gain access to client-side state from another web application, he must find a way to cause his[2] code to have the same origin as the target web application.

Cross-Site Scripting (XSS) vulnerabilities are a class of server-side vulnerabilities that allow an attacker to launder malicious code through a trusted server. This effectively allows an attacker to forge the origin of the victim web application causing the victim web application to serve the attacker's code and giving it the same origin as all other code from that application. At this point, there is no further barrier between the attacker's code and the victim web application's client-side state. The attacker can run malicious code within the browser, impersonate the user to the victim web application, steal the user's private data and authentication credentials, or present forged content to the user. XSS vulnerabilities pose a serious threat to the security of modern web applications. Year after year, they top lists of the most dangerous and the most commonly reported vulnerabilities [31, 74].

[1]The origin of a web page is defined as the 3-tuple of the hostname, port, and protocol of the web server that served the page. Some browsers, e.g. some versions of Internet Explorer, define origins more broadly by ignoring the port number. [119, Chapter 9]

[2]When we refer to malicious actors in this document, we choose personal pronouns arbitrarily.

They are surprisingly easy to create and difficult to mitigate completely. Any web application that fails to properly sanitize user input before displaying it to other users will be vulnerable to XSS attacks.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
    "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="en">
<head>  <title>nile.com : ++Shopping</title>  </head>
<body>  <h1 id="title">{item_name}</h1>
  <p class='review'>{review.text}
     -- <a href='{review.contact}'>{review.author}</a>  </p>
</body>
</html>
```

Figure 2.1: Vulnerable Web Page Template. This template is used to render dynamic web pages. It is written in a Smarty-like language where the appearance of the token {x} instructs the template engine to replace the token by the value of the variable named "x".

Figure 2.1 shows an example web page template like those used by many web applications to render dynamic web pages. A sequence of the form {x} will be replaced at runtime by the value of the variable x. For instance, if an attacker can submit a review containing <script src='http://badguy.com/attack.js'/>, the template variable review.text will be replaced with this script tag. When a victim visits the page, the victim's web browser will download and execute http://badguy.com/attack.js with the same permissions as legitimate scripts. It has long been recognized that non-script elements pose a threat as well. [24, 105] For instance, an attacker could inject a fake login form and style it to obscure a legitimate login form. When the victim attempts to login, their credentials could be sent to a site of the attacker's choosing. Nevertheless, many existing XSS defenses focus on preventing the execution of untrusted scripts without addressing malicious non-script content.

To prevent XSS vulnerabilities, all the untrusted (user-contributed) content in a web page must be sanitized. However, proper sanitization is very challenging. The context in which untrusted data is interpreted determines the forms of sanitization that are appropriate. If sanitization is performed by the server, but the browser interprets the content in a way that the server did not intend, there are many ways for an attacker to take advantage of the discrepancy [89]. The Samy worm [92], one of the fastest spreading worms to date, used the ambiguity between server sanitization and client parsing to propagate. Alternatively, one could let the client sanitize untrusted content. However, without the server's help, the client cannot distinguish between trusted and untrusted content in a web page since both appear to originate from the trusted server. We can avoid ambiguity between the client and server by requiring the server to identify untrusted content and requiring the client to ensure that it is displayed safely.

However, challenges remain. After the server identifies untrusted content, it needs to tell the client the locations of the untrusted content in the document tree. If the untrusted content (without executing) could distort the structure of the document tree, it could evade sanitization. To achieve this, the untrusted content could contain node delimiters that split the original document node where untrusted content resides into multiple nodes. For example, if untrusted scripts are not permitted within `<p class='review'>` tags in the template of Figure 2.1, an attacker could submit a review of `</p> <script src='http://badguy.com/attack.js'/> <p>` to escape the review paragraph in order to cause his injected script to execute. This is known as a *Node-splitting attack* [48]. To defend against this attack without restricting the richness of user provided content, the server must take care to remove only those node delimiters which would introduce new trusted nodes.

Our system, *Noncespaces*, provides an end-to-end mechanism that allows a server to identify untrusted content, to reliably convey this information to the client, and that allows the client to enforce a security policy on the untrusted content. Noncespaces is inspired by Instruction Set Randomization [49, 7], which randomizes the processor's instruction set to identify and defeat injected malicious binary code. Noncespaces leverages the similarities between injected code in executable programs and injected content in web pages to apply Instruction Set Randomization-like techniques to prevent XSS vulnerabilities. Noncespaces randomizes (X)HTML tags and attributes to identify and defeat injected malicious web content. Randomization serves two purposes. First, it identifies untrusted content so that the client can use a policy to limit the capabilities of untrusted content. Second, it prevents the untrusted content from distorting the document tree. Since the randomized tags are not guessable by the attacker, he cannot embed proper delimiters in the untrusted content to split the containing node without causing parsing errors.

Various techniques can be used to distinguish between trusted and untrusted content on the server-side.

In order to efficiently distinguish between trusted and untrusted content on the server-side, Noncespaces takes advantage of a popular web application design pattern. In Model-View-Controller [20] frameworks, template languages are commonly used to generate (X)HTML output. As seen in the example of Figure 2.1, a template provides the basic overall structure of the (X)HTML page and explicit template variable expressions are used to insert dynamic data. Noncespaces modifies a popular template engine to automatically distinguish between trusted static content written by the web application developers and dynamic content that may be from an untrusted source. This allows Noncespaces to easily identify the bulk of trusted (X)HTML code in the web application and facilitates simple, ef-

fective client-side policies to defend against popular XSS attack vectors. Noncespaces uses a flexible yet simple language to specify common security policies. For more complex information flow policies, the Noncespaces policy language supports organizing trust classes into an arbitrary lattice allowing a wide range of expression. In contrast to most existing XSS defenses, Noncespaces allows prevention of both script and non-script XSS attacks. To ease policy development, Noncespaces provides a training mode that enables a policy writer to quickly create a policy allowing intended application behavior. To demonstrate the effectiveness and usability of Noncespaces we port a 155K SLOC blog application to work with Noncespaces, use the training mode to develop a policy for the application, and conduct an extensive security evaluation with the generated policy.

2.2 Threat Model

We restrict our attention to XSS attacks where the malicious content is unintentionally delivered to the victim user by a trusted server. Specifically, our solution addresses reflected (Type I) and stored (Type II) XSS attacks. In a reflected XSS attack, the victim visits a page controlled by the attacker. The attacker encodes malicious content into a link, web form, embedded frame, or JavaScript redirect that targets a trusted web site. When the victim's browser sends the attacker-controlled request to the targeted web application, the malicious content is *reflected* by the trusted web server back to the victim's web browser. In a stored XSS attack, the attacker causes malicious content to be *stored* directly on a trusted web server. Later, when the victim visits the trusted web server, the attacker's malicious content will be delivered to the victim's web browser. In both scenarios, because the malicious content was received from a trusted server, it will be granted full access to

all client-side data belonging to the trusted server.

The implementation of Noncespaces in this work does not attempt to address DOM-based (Type III) XSS attacks, where trusted client-side JavaScript permits the injection of untrusted content in violation the web application's security policy. However, the techniques used by Noncespaces could also be used in a client-side JavaScript implementation to prevent DOM-based XSS attacks.

Noncespaces does not seek to address Universal XSS Vulnerabilities [94], where a browser extension can be tricked into violating the browser's own security policies; or Cross-Site Request Forgery (CSRF), where a malicious web server causes the client to send requests to a trusted server that is not prepared to distinguish between requests initiated by other sites from requests initiated by the user.

Because the goal of Noncespaces is to defend against XSS attacks, we assume that the attacker's only means of attack is to submit malicious data to XSS-vulnerable web applications. Existing mechanisms can be used to ensure that an attacker cannot compromise the web server or browser directly via buffer overflow attacks, malware, etc.

2.3 Noncespaces

The goal of Noncespaces is to allow the client to safely display documents that contain both trusted content generated by a web application and untrusted content provided by untrusted users. To eliminate the client-server semantic gap and to adapt to differing security needs, the browser enforces a configurable security policy. The policy specifies the browser capabilities that each type of content can exercise. In this way, malicious content injected by an attacker is restricted to the capabilities allowed to untrusted content by the policy.

In order for the client to faithfully enforce a server-specified policy, the client must be able to unambiguously determine the trustworthiness of all content in a document. Therefore, the server must first classify content into discrete trust classes. The server then must communicate the content, trust classification, and policy to the client. Finally, the client can enforce the policy. This process is depicted in Figure 2.2.

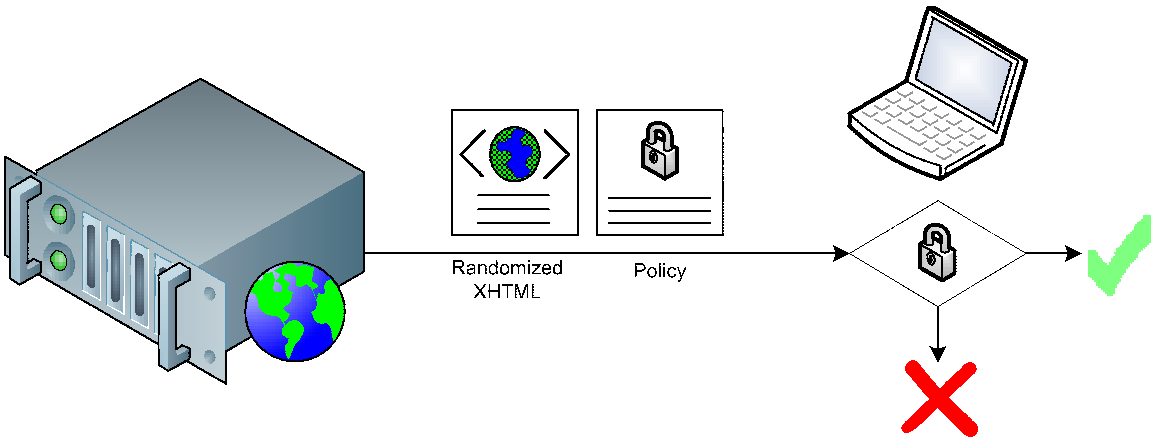

Figure 2.2: Noncespaces Overview. The server delivers an (X)HTML document annotated with trust class information and a policy to the client. The client accepts the document only if it satisfies the policy.

Our architecture permits Noncespaces to defeat both reflected and stored XSS attacks. In both scenarios, untrusted user input is returned to a victim user—immediately in the case of a reflected XSS attack or at some later time in the case of a stored XSS attack. In either case, as long as the server's content classification is conservative, the server faithfully communicates its classifications to the client, and the client faithfully enforces the server-specified policy, untrusted content will be confined to the capabilities expressly permitted to it by the policy, neutralizing the effects of an attempted XSS attack.

We take a modular approach to classifying content. The server can use a variety of techniques to determine the trust classes of content, ranging from whitelisting known-good

code to annotating output based on program analysis or information flow tracking. We present an expedient approach in Section 2.4. In this section, we describe our mechanisms for communicating trust information and policy enforcement.

2.3.1 Communicating Trust Information

There are a variety of mechanisms that a server might use to indicate content trust information to clients. The server could use a designated attribute to indicate the trust class of an element. However, malicious content may contain elements which forge the attributes that designate trusted content. Alternatively, the server could indicate the trustworthiness of content by its location in the document, e.g. restricting the capabilities of all descendants of a specific document node — a sandbox node [36, 66, 48]. However, malicious content may contain tags that split its original enclosing node into multiple nodes so that malicious nodes are no longer descendants of the sandbox node. This is the node-splitting attack discussed in Section 2.1.

Another alternative, inspired by Instruction Set Randomization (ISR), is to greatly reduce the probability of a successful attempt to forge trust class information by preventing an attacker from being able name trusted content. ISR defends against binary code injection attacks by randomly perturbing the instruction set of an application. To inject code with predictable semantics into the application, the attacker must correctly guess the randomization used. This is very difficult if the number of possible randomizations is sufficiently large. The attacker is effectively prevented from injecting code. We propose a similar technique for (X)HTML. We associate a different randomization function with each content trust class. The names of all elements and attributes in a trust class are remapped according to the associated randomization function so that no injected content can correctly name

(X)HTML elements or attributes in other trust classes.

```
<!DOCTYPE html>
<r617html r617lang="en">
<r617head>  <r617title>nile.com : ++Shopping</r617title>  </r617head>
<r617body>  <r617h1 r617id="title">Useless Do-dad</r617h1>
  <r617p r617class='review'></p><script>attack()</script><p>
    -- <r617a href=''></r617a>  </r617p>
</r617body>
</r617html>
```

Figure 2.3: HTML Document Randomized by Noncespaces. A random prefix has been applied to trusted HTML content in a template like that of Figure 2.1. The rendered document contains a node-splitting attack injected by a malicious user.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
    "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<r617:html xmlns="http://www.w3.org/1999/xhtml" r617:lang="en"
    xmlns:r617="http://www.w3.org/1999/xhtml">
<r617:head>  <r617:title>nile.com : ++Shopping</r617:title>  </r617:head>
<r617:body>  <r617:h1 r617:id="title">Useless Do-dad</r617:h1>
  <r617:p r617:class='review'></p><script>attack()</script><p>
    -- <r617:a href=''></r617:a>  </r617:p>
</r617:body>
</r617:html>
```

Figure 2.4: XHTML Document Randomized by Noncespaces. A random namespace prefix has been applied to trusted XHTML content in the template of Figure 2.1. The rendered document contains a node-splitting attack injected by a malicious user.

For example, let the randomly chosen string `r617` denote trusted content. We can defeat XSS attacks against the document from Figure 2.1 by annotating it. For HTML documents, we can prefix trusted tags and attributes with our random identifier as shown in Figure 2.3. For XHTML documents, we can preserve the original XML semantics of the document

while annotating by using our random identifier as an XML namespace prefix[3] as shown in Figure 2.4.

As illustrated by the embedded node-splitting attack, the attacker cannot inject malicious content and cause it to be interpreted as trusted because he does not know the random prefix. He also cannot escape from the enclosing paragraph element, because he does not know the random prefix and therefore cannot embed a closing tag with this prefix. (In the HTML document, the `<script>` element is the child of an `<r617p>` element, not a `<p>` element. In the XHTML document, when a closing tag tries to close an open tag but the prefixes of the two tags do not match, the XML parser will fail with an error.[4])

To prevent an attacker from guessing (namespace) prefixes, we choose the prefixes uniformly at random every time a response is rendered — hence the term Noncespaces. Given a prefix space of appropriate size, knowing the random prefixes in one instance of the document does not help an attacker predict prefixes in future instances of the document.

There is an additional complication, however. Naïvely prohibiting all untrusted content will not work because most modern web applications are designed to accept some amount of rich content from users. Though we can use randomization to ensure integrity of trust class information, a policy that places appropriate constraints on rich content provided by untrusted users is still necessary for our solution to be useful in practice. Therefore,

[3]To permit mixing of XML languages in a single document, XML namespaces [17] can be used to designate elements and attributes by namespace URI and local name. The URI for the language is bound to a prefix using the `xmlns` attribute. The prefix is then used to qualify the local names of elements and attributes from that language. E.g. `<p:a xmlns:p='http://www.w3.org/1999/xhtml'>` designates an XHTML`<a>` element.

[4]A subtlety occurs when two different prefixes, say `a` and `b`, are associated with the same URI. In this case, is "`<a:foo></b:foo>`" valid? Even though `<a:foo>` and `<b:foo>` are semantically equivalent, XML matches opening and closing tags *lexically* [18]. Thus "`<a:foo></b:foo>`" is not well-formed regardless of how `a` and `b` are bound. All XHTML compliant browsers we have tested exhibited this behavior. This implies that Noncespaces needs to randomize only namespace prefixes, but not the URIs to which the prefixes are associated.

we provide a mechanism for the server to specify a policy for the client to enforce when rendering the document.

Noncespaces adds three HTTP protocol headers to each HTTP response:

- `X-Noncespaces-Version: [0-9]+\.[0-9]+` communicates the version of the Noncespaces policy and semantics that should be used, in case future changes are required.

- `X-Noncespaces-Policy:` *URI* denotes the URI of the policy for the current document. If the client does not have the policy in its cache, a compliant client must first retrieve the policy and validate the document before rendering.

- `X-Noncespaces-Context:` *TrustClass*=*Rand*(,*TrustClass*=*Rand*)*

  Both *TrustClass* and *Rand* reduce to *Name*. To prevent an attacker from guessing the namespace prefixes in an (X)HTML document, the server must use different randomized prefixes each time it serves the document. This header maps the random identifiers used in the (X)HTML document delivered by the response to the trust class names used in the policy. This allows clients to cache the policy because the server can provide the same policy file to all the requests for the (X)HTML document.

Compatibility with Caching

At first glance, response caching may appear to pose a problem for Noncespaces. Noncespaces requires that an attacker must not be able to predict the value of prefixes in future document instances. Local caches, caching proxies, and content delivery networks (CDNs) save a single response and may deliver it multiple times. Indeed, if it were possible to inject

malicious content into a response after learning the prefix values chosen when the document was rendered, an attacker could defeat our encoding mechanism. Also, if a cache delivers a Noncespaces-encoded document without its associated headers, clients will not be able to interpret the response correctly.

To prevent an attacker from injecting content after learning which prefixes were chosen, we ensure that HTTP/1.1 compliant caches will only respond with complete Noncespaces-encoded responses. Only content injected by an attacker at the time the document was rendered, before the attacker learns the prefix values, will be included in any single response. Even if that response is subsequently rendered multiple times, it does not provide additional opportunities for an attacker to inject content.

An HTTP/1.1 [39] compliant cache will not combine portions of multiple responses into a single response unless it can ensure that the entity's octet representation has not changed. Choosing new random prefixes every time a document is rendered ensures that the octet representation of the entity will differ, with high probability, in every response. This ensures that a cache will not provide additional opportunities to inject content by combining multiple responses.

HTTP/1.1 also requires caches to store all end-to-end headers with each cache entry and to include them in any response formed from that cache entry. This ensures that clients will receive the headers necessary to interpret each Noncespaces-encoded response whenever such a response is served by a cache.

2.3.2 Policy Specification

A Noncespaces policy specifies the browser capabilities that content in a given trust class can invoke. A grammar for our policy language is given in Figure 2.5. We designed

Policy	::=	((*Namespace* | *TrustClass* | *Order* | *Rule* | *Comment*) \n)*
Namespace	::=	`namespace` *Name* "? *URI* "? *Comment*?
TrustClass	::=	`trustclass` *Name Comment*?
Order	::=	`order` *Relations Comment*?
Relations	::=	*Name* < *Name*
	|	*Name* < *Name* , *Relations*
Rule	::=	*Decision XPathExpression*
Decision	::=	`allow` | `deny`
Comment	::=	# .*
Name	::=	[a-zA-Z0-9_.-]+
URI	::=	As per RFC 2396

Figure 2.5: Grammar for Noncespaces Client-side Policy Language

`boolean ns:trust-class(node, constraint)` where `constraint` is a string specifying a comparison operator (e.g. =, !=, >, <, >=, <=) and the name of a trust class. If the trust class of node `n` is `tc1`, `ns:trust-class(n, '>=tc2')` returns the value of `tc1 >= tc2`.

`boolean ns:isspecified(attr)` Returns `true` if `attr` was explicitly specified in the document (as opposed to a default attribute supplied by a parser).

`string ns:tolower(string)` Return argument converted to uppercase.

`string ns:toupper(string)` Return argument converted to lowercase.

Figure 2.6: XPath Functions Provided by Noncespaces

the policy language to be similar to a firewall configuration language. Comments begin with an `#` character and extend to the end of the line. A minimal policy consists of a sequence of `allow`/`deny` rules. Each rule applies a policy decision—allow or deny— to a set of document nodes matched using an XPath expression. We have employed the XPath language because it was specifically designed for querying content from hierarchical documents. This allows constraints to be placed on elements, attributes, values, and position of nodes in the document hierarchy. Noncespaces also provides utility functions for string

normalization and for matching nodes based on trust class or whether an attribute value has changed from the default specified by the language (Figure 2.6). For example, the XPath expression `//a` can be used to match all anchor elements descending from the document's root node (`//`). `//@href` will match all `href` attributes in the document. `namespace` declarations bind a namespace prefix to an XML namespace URI for use in XPath expressions.

The `trustclass` and `order` commands are used to define the hierarchy of trust classes. The optional `trustclass` command declares a trust class. A sequence of `order` commands encode the partial ordering between trust classes. This allows policy authors to specify any lattice relation over trust classes.

When checking that a document conforms to a policy, the client considers each rule in order and matches the XPath expression against the nodes in the document's Document Object Model. Policy decisions are made on a first-matched basis. When an allow rule matches a node, the client permits the node and will not consider the node again when evaluating subsequent rules. When a deny rule matches a node, the client determines that the document violates the policy and will not render the document. To provide a fail-safe default, if any nodes remain unmatched after evaluating all rules, we consider those nodes to be policy violations (i.e. all policies end with an implicit `deny //*|//@*`). In the event that a policy author wishes to override the default behavior in order to specify a blacklist policy, he can specify `allow //*|//@*` as the last rule to allow all nodes that have not been matched so far. Algorithm 1 gives the algorithm for checking a policy.

Example policies are provided in Figures 2.7 and 2.8. The policy in Figure 2.7 is a policy for XHTML documents that specifies two trust classes, trusted and untrusted. There are no restrictions on which tags and attributes can appear in trusted content. Only tags and

Algorithm 1: Document Validation Algorithm. This algorithm determines whether a document satisfies a Noncespaces policy.

Input : A document d and a policy p.
Output: TRUE if the document d satisfies the policy p; FALSE otherwise.

```
1  begin
2      for Element or attribute node n ∈ d do
3          n.checked = FALSE
4      for Rule r ∈ p.rules do
5          for Node n ∈ d.matchNodes(r.XPathExpression) do
6              if n.checked == FALSE then
7                  if r.action == ALLOW then
8                      n.checked = TRUE
9                  else
10                     return FALSE
11     for Element or attribute node n ∈ d do
12         if n.checked == FALSE then
13             return FALSE;
14     return TRUE;
15 end
```

attributes that correspond to BBCode[5] are allowed in untrusted content: stylistic markup, links to other HTTP resources, and images. Note that lines 17–18 only permit link and image tags to specify URLs for the (non-script) HTTP protocol and that all other attributes on link and image tags are denied by line 21.

Figure 2.8 demonstrates the language's flexibility for more fine-grained policies. Lines 1–2 declare that the namespace prefixes `x` and `m` used in the following patterns refer to XHTML and MathML content, respectively. Lines 4–7 declare multiple trust classes and line 8 defines the ordering between them. The policy provides four trust classes. In or-

[5]BBCode allows basic formatting and linking of non-active content: `https://en.wikipedia.org/wiki/BBCode`

```
1  # Restrict untrusted content to safe subset of XHTML
2  namespace  x   http://www.w3.org/1999/xhtml
3  # Declare trust classes
4  trustclass trusted
5  trustclass untrusted
6  order untrusted < trusted
7
8  # Policy for trusted content
9  allow //x:*[ns:trust-class(., "=trusted")]  # Allow all trusted elements
10 allow //@x:*[ns:trust-class(., "=trusted")] # Allow all trusted attributes
11
12 # Allow safe untrusted elements
13 allow //x:b  |  //x:i    |  //x:u  |  //x:s  |  //x:pre  |  //x:q
14 allow //x:a  |  //x:img  |  //x:blockquote
15
16 # Allow HTTP protocol in the <a href> and <img src> attributes
17 allow //x:a/@href[starts-with(., "http:")]
18 allow //x:img/@src[starts-with(., "http:")]
19
20 # Deny all remaining elements and attributes
21 deny //*  |  //@*
```

Figure 2.7: Example Noncespaces Policy. This policy restricts untrusted content to BBCode.

der from most to least trust they are: (1) `static`: static web application content, (2) `developer`: dynamic content written by developers, (3) `auth`: dynamic content written by authenticated users, and (4) `unauth`: dynamic content written by unauthenticated users. In this web application, the capabilities of each trust class are a superset of the capabilities of all less-trusted classes. Line 11 allows `static` elements and attributes from any XML namespace to appear in the document.

In this policy, we make the simplifying assumption that all default attributes added by the XML parser are safe. Therefore, line 12 allows all default attributes from any XML namespace. Lines 15–17 allow dynamic XHTML `<script>` tags and `href` attributes to

```
1  namespace  x  http://www.w3.org/1999/xhtml
2  namespace  m  http://www.w3.org/1998/Math/MathML
3
4  trustclass static             # static code
5  trustclass developer          # dynamic code written by developers
6  trustclass auth               # dynamic code by authenticated users
7  trustclass unauth             # dynamic code by unauthenticated users
8  order unauth < auth, auth < developer, developer < static
9
10 # Allow all static code from any namespace and default attributes
11 allow //*[ns:trust-class(., '=static')] | //@*[ns:trust-class(., '=static')]
12 allow //@*[not(ns:isspecified(.))]
13
14 # Allow developer authored scripts and javascript: links
15 allow //x:script[ns:trust-class(., '>=developer')]
16 allow //x:script/@src[ns:trust-class(., '>=developer')]
17 allow //@href[ns:trust-class(., '>=developer')]
18
19 # Allow authenticated users to provide http: links and images
20 allow //x:a[ns:trust-class(., '>=auth')]
21 allow //x:a/@href[starts-with(., 'http:') and ns:trust-class(., '>=auth')]
22 allow //x:img[ns:trust-class(., '>=auth')]
23 allow //x:img/@src[starts-with(., 'http:') and ns:trust-class(., '>=auth')]
24
25 # Allow presentation and MathML markup from unauthenticated users
26 allow //x:b | //x:u | //x:i | //x:em | //x:strong | //x:pre
27 allow //m:*
```

Figure 2.8: Example Multi-level Noncespaces Policy. This policy illustrates multiple levels of trust, multiple XML languages, and use of custom XPath functions provided by Noncespaces.

be created by developers. Lines 20–23 allow trust classes greater than or equal to `auth` to specify links and images that reference an absolute HTTP URL. Finally, lines 26–27 allow all trust classes to specify various text presentation markup and MathML content.

This policy illustrates several strengths of our policy language. By using `ns:isspecified()` on line 12 to allow all default attributes, we avoid having to explicitly

allow every default attribute that the XML parser might add to each element. Each of the rules beginning from line 20 illustrates how the ability to make comparisons between trust classes saves the policy writer from having to explicitly enumerate every trust class that is permitted to specify a particular element or attribute. Line 27 allows any trust class to specify any MathML element by qualifying a wildcard with the MathML namespace prefix (`m`). Also, because trust classes form a lattice, there is always a unique lowest trust class. In the event that a server-side bug causes some content to escape classification, the browser can automatically place that content into the lowest trust class. (Each of these items represents an improvement over the policy language originally presented in [109].)

Though our examples employ XML, XPath can be used with HTML documents as well. Even though HTML documents need not be well-formed, browsers translate HTML into a Document Object Model representation that can be used to service XPath queries. We prefer this policy mechanism to more complex ones like event-based policies or dynamic information flow tracking for several reasons. Because it is not Turing complete, it is easier to reason about the effects of a policy making incorrect policy specification less likely. Enforcing policy at the syntax (Document Object Model) level also has advantages. It makes implementation less intrusive, facilitating adoption across multiple browsers and making it possible to retrofit legacy browsers without requiring internal modifications by using our proxy implementation described in Section 2.4.2.

2.3.3 Client Enforcement

When receiving a Noncespaces encoded response from a server, the web browser must ensure that the document is well-formed and conforms to the policy before rendering it. This requires the browser to retrieve the policy from the web server if it doesn't already

have an unexpired copy in its cache. The overhead involved in policy retrieval should be minimal given that most web pages are assembled from multiple requests. We also expect it to be common for a single, seldom-changing policy to be used for each web application.

Client-side enforcement of the policy is necessary because it avoids possible semantic differences between the policy checker and the browser, which might lead the browser to interpret a document in a way that violates the policy even though the policy checker has verified the document.

2.4 Implementation

2.4.1 Server Implementation

Noncespaces requires the server to identify untrusted content in web pages. The server may choose any approach from whitelisting trusted content statically to determining untrusted content dynamically by program analysis or information flow tracking. In our prototype implementation, we choose an approach that leverages a popular web application development paradigm to conservatively classify content with low overhead. The Model-View-Controller [20] design pattern advocates separating presentation from business logic. Many modern web applications employ a template system that inserts the dynamic values which business logic computes into static templates that decide the presentation of the web page. Since web developers author templates, content in templates can be trusted. By contrast, dynamic values may, and often do, come from untrusted sources. We consider dynamic values to be untrusted. This approach requires that JavaScript be placed in templates to be recognized as trusted content. This requirement is reasonable because most scripts can be specified statically. Scripts may then use DOM interaction to query for dynamic inputs.

Treating all dynamic values as untrusted is safe, but it might be too conservative in some situations. Consider the following template used to toggle the visibility of a dynamic menu: `<a onclick='toggle("{id}")'>`. The `toggle(id)` function accepts a string parameter indicating the HTML id of the element to operate on. Because the value of the `onclick` attribute contains a template variable (`id`), it is treated as a dynamic (untrusted) value. If the policy denies all untrusted `onclick` attributes, the client will reject this document, even when the generated JavaScript code conforms to the developer's intentions. There are several straightforward solutions to this problem. Our use of a configurable policy allows a policy writer to explicitly whitelist safe, untrusted content through constraints on attribute values. For instance, the policy could allow untrusted `onclick` attributes which conform to the intended format: `toggle("[A-Za-z0-9]+")`. Another alternative is for certain untrusted content to be whitelisted within the web application after ensuring either proper sanitization or ensuring that it contains no malicious input by program analysis or information flow tracking. We would then consider an XHTML construct to be trusted if it is either static or on the whitelist.

NSmarty

To automatically annotate the content of web pages generated from templates, we modified the Smarty Template Engine [95], a popular template engine for the PHP language. The Smarty language is a Turing-complete template language that allows dynamic inclusion of other templates. A Smarty template consists of free-form text interspersed with template tags delimited by { and }. A template tag either prints a variable or invokes a function. To use Smarty, a PHP program invokes the Smarty template engine, passes a template (or templates) to the engine, and assigns values to the variables referenced in the template.

The template engine will then generate a document based on the template and the dynamic values provided.

For our prototype implementation, we apply randomization to XHTML documents. To randomize XML namespace prefixes in Smarty templates, we must be able to recognize static XHTML constructs in the template. Since the Smarty language allows Smarty tags to appear anywhere in a template, e.g. in element and attribute names, we must restrict the Smarty language to be able to determine all static XHTML elements and attributes. Hence, we specified a subset of the Smarty language, which we call *NSmarty*. NSmarty prohibits template tags from appearing in element names or attribute names. Through these modest restrictions, we ensure that we can correctly identify all the statically specified XHTML tags and attributes.

The Smarty template engine operates in two phases. The first time it encounters a template, it compiles the template into PHP code and caches it. On each request, the cached PHP code will run to render the output document. We provide a preprocessor that is invoked by Smarty before it compiles each template. Our preprocessor inserts PHP code that replaces static XML namespace prefixes with randomly generated prefixes each time the document is rendered. The process is depicted in Figure 2.9.

To preserve the semantics of the generated document, we map each static prefix to a random prefix bound to the same namespace URI as the static prefix (note that different prefixes may be bound to the same URI or the same prefix may be bound to different URIs at different points in the document). However, since the Smarty (and also our NSmarty) language is Turing-complete, it is infeasible to determine the scope of each static prefix at compile time. This implies that it is also infeasible to determine the URI that each static prefix represents. Therefore, we map each unique static prefix to a unique random prefix.

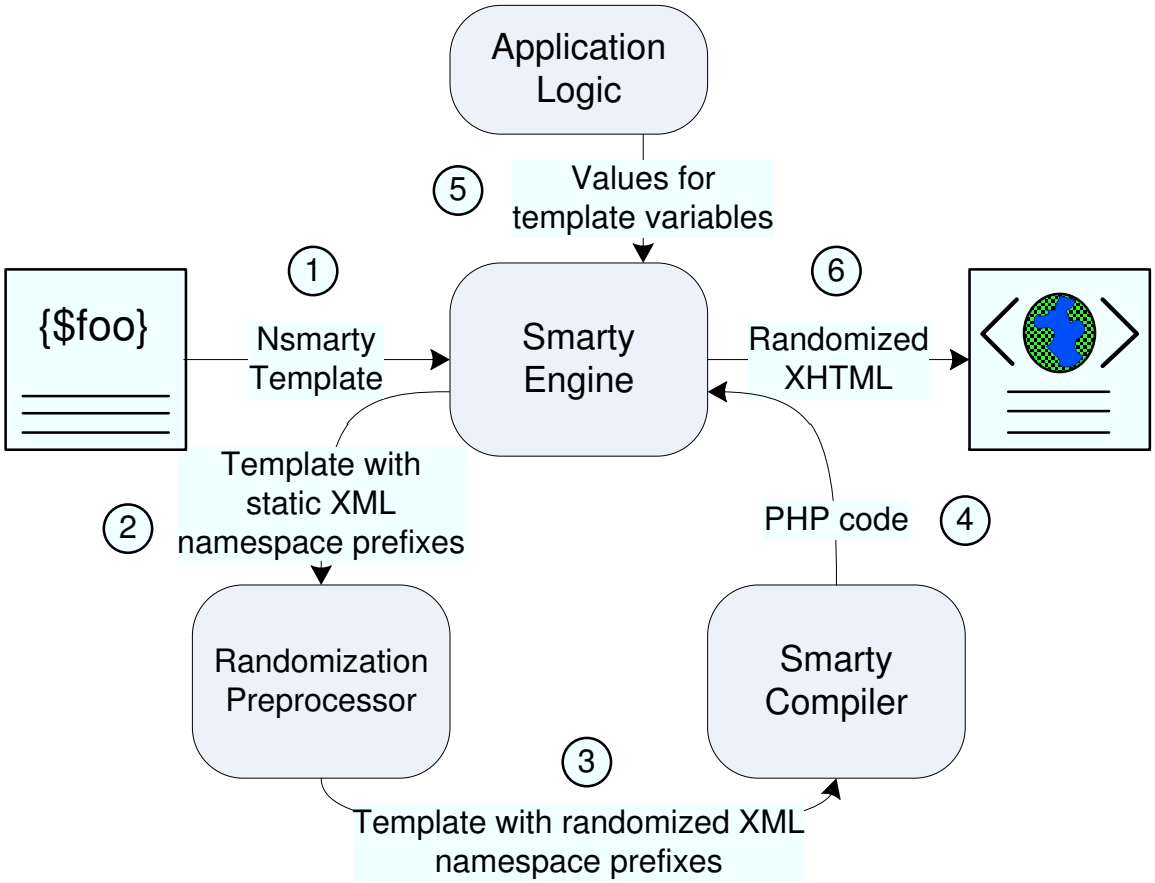

Figure 2.9: Implementing Noncespaces within Smarty. This figure illustrates the operation of our modified Smarty template engine.

If the original document without prefix randomization is well-formed and all XHTML appearing in dynamic content is well-formed, the new document with prefix randomization will also be well-formed and will be semantically equivalent to the original document. If dynamic content contains non-well-formed XHTML, our prefix randomization algorithm may create non-well-formed documents. We prevent this from occurring by verifying that each document remains well-formed after prefix randomization.

Figures 2.1 and 2.4 show an original XHTML template and the rendered document after prefix randomization, respectively. Algorithm 2 shows the pseudocode for prefix

Algorithm 2: XML Namespace Prefix Randomization Algorithm. This algorithm prepends randomly chosen prefixes to all static XML elements and attributes within a document.

```
   Input  : An XML document d
   Output: The document d after prefix randomization
 1 begin
 2     for Tag t ∈ d do
 3         for Attribute a ∈ t do
 4             if a is a namespace declaration then
 5                 if map[a.prefix] is not defined then
 6                     map[a.prefix] = random()
 7                 a.prefix = map[a.prefix]
 8             else if a.value is static (i.e. containing no template tag) then
 9                 a.prefix = map[a.prefix]
10
11         t.prefix = map[t.prefix]
12 end
```

randomization.

Because NSmarty takes advantages of XML namespaces, the server should serve Noncespaces documents with the `application/xhtml+xml` content type. Serving the document as XML provides other benefits, discussed below.

2.4.2 Client Implementation

The client validates each document against its policy before rendering to ensure safety. We implemented our policy validator as a client-side proxy that mediates communication between the browser and the server. Our proxy forwards requests from the web browser to the appropriate server. When it receives a Noncespaces encoded response from the server, the proxy attempts to validate the document against the specified policy. If the document conforms to the policy, the proxy forwards it to the client. If the document violates the

policy, fails to parse, or some other error occurs, such as the policy being malformed or inaccessible, the proxy returns an error document indicating the problem to the web browser.

We chose a proxy implementation to provide a rapid proof of concept and incremental deployability for any browser that supports the use of an HTTP proxy. Also, until adoption of Noncespaces is sufficiently widespread, a Noncespaces enabled server can protect incompatible clients by employing our proxy in a reverse proxy configuration. Using Noncespaces in this configuration is reminiscent of SWAP [116] and is subject to shortcomings we outline in Section 2.6.

Performing policy validation in a proxy, instead of within the browser, has several disadvantages. Using a proxy increases the response latency experienced at the browser. Also, because the policy validator does not have access to the browser's DOM, parsing differences between the validator and the browser may provide opportunities for attack. Our NSmarty implementation targets XHTML served as `application/xhtml+xml` to help mitigate this problem. The stricter parsing requirements of XML means that the proxy is less susceptible to parsing ambiguities than would be the case with HTML. We do not view requiring XHTML 1.0 compliance as a shortcoming of our prototype. Most modern browsers (with the notable exception of Microsoft Internet Explorer) are XHTML compliant. The restrictions imposed by XHTML are not onerous; they merely require documents to follow a simple, well-defined format. However, the prefix randomization technique that we have presented has one subtle incompatibility with XHTML that is easy to work around. While some browsers (such as Opera [80]) understand XHTML attributes that have been qualified with a prefix bound to the XHTML namespace, XHTML Modularization 1.1 [5] specifies that most XHTML attributes should not be qualified. For browsers that do not

support qualified attributes, we can use a client-side JavaScript stub to unqualify attributes randomized by Noncespaces after the document has been validated.

2.4.3 Policy Training Mode

In order to protect any web application effectively, Noncespaces requires a security policy. To create a whitelist policy manually, a policy developer must enumerate all outputs that should be permitted by the policy. This can be difficult for web applications with a significant amount of dynamic content. The developer can either attempt to statically infer all possible outputs from the application source or she can run the application in order to observe possible outputs. Achieving completeness is a challenge with either approach. The Turing-completeness of common template languages can make it impossible to statically determine all possible outputs. Likewise, output observed from running an application will not reflect any application features not exercised by the developer.

Even given a complete view of application output, the developer must write rules which accurately capture permitted outputs and then test the web application with the resulting policy to ensure that the policy is general enough to allow full use of the web application. Whenever a policy violation is encountered the developer must either modify the policy to allow the offending document structure or modify the web application to conform to the existing policy. Performing these actions by hand can be very tedious and time consuming.

To facilitate rapid policy development, we have implemented a training mode for our client-side proxy that helps the developer create a whitelist policy for their application by automating many of these steps. The developer provides a seed policy, an incomplete policy that they would like to serve as a basis for the policy generated by our training system. The developer then exercises web application functionality in a trusted environment

to provide a reasonably complete view of the web application output. Whenever the proxy encounters a document node that is not permitted under the current policy, it generates a rule to allow that node and adds it to the current policy. In this way, when the developer finishes exercising the full-range of application functionality, the proxy can return a policy allowing all of the content encountered. This combines, into a single activity, the separate steps of determining possible application outputs, writing rules to permit them, testing the policy, writing rules to remedy incompleteness, and subsequent re-testing of the policy.

Beginning training from a seed policy allows a policy to be incrementally updated after changes are made to the application by supplying the current policy as the seed policy and re-running the application test suite. An empty seed policy corresponds to a deny by default policy and rules will be generated to allow every document node encountered.

Exercising the application can be automated by running existing functionality and quality assurance tests in an integration testing environment. If automated tests aren't available, the developer can manually interact with the web application. Common test automation tools [90, 93] can be used to create an automated test suite from a one-time manual interaction.

When the proxy encounters content not permitted by the policy, it must derive policy rules to allow the content. This is safe because training occurs in a trusted environment. Therefore, the output will not contain any malicious content. When creating a rule to remedy a policy violation, our system must make a tradeoff between being too restrictive and too permissive. If our system attempted to find a maximally-restrictive set of rules allowing only observed behavior, the learned policy would prohibit legitimate outputs not seen during training. Instead our training mode generates simple rules allowing the node in question to appear anywhere in the document. The generated rules must be reviewed by the

developer to ensure that they conform to the intended security policy. We believe that this is an acceptable tradeoff—guidance from the developer is already necessary because our system cannot know the developer's intended security policy. In practice, we find that the training mode allows us to quickly derive a policy that permits large classes of innocuous XHTML content while highlighting security-sensitive content that was not present in the stub policy.

2.4.4 Deployment

It is easy to retrofit existing web applications with Noncespaces. The developer writes an appropriate policy and, when necessary, revises the Smarty templates such that they are also valid NSmarty templates.

If the developer wishes to enforce a static-dynamic policy, where all static content in the Smarty template is trusted and all dynamic content is untrusted, no further modification is necessary. Noncespaces will randomize all the static namespace prefixes. Because no namespace prefixes in the dynamic content will be randomized, they cannot invoke any capabilities reserved for trusted content.

2.5 Evaluation

To evaluate the effectiveness and overhead of Noncespaces we conducted several experiments. We evaluated the security of Noncespaces to ensure that it is able to prevent a wide variety of XSS attacks. We also measured the performance overhead imposed by Noncespaces from both the client's and server's points of view.

2.5.1 Security

TikiWiki Case Study

We tested Noncespaces against six XSS exploits targeting two vulnerable applications. The exploits were crafted to exhibit the various forms that an XSS attack may take [109]. The applications used in this evaluation were a version of TikiWiki [106] with a number of XSS vulnerabilities and Trustify, a custom web application that we developed to cover all the major XSS vectors.

We began by developing policies for each application. Because TikiWiki was developed before Noncespaces existed, it illustrates the applicability of Noncespaces to existing applications. We implemented a straightforward 37-rule, static-dynamic policy that allows unconstrained static content but restricts the capabilities of dynamic content to that of BBCode (similar to Figure 2.7). We also had to add exceptions for trusted content that TikiWiki generates dynamically by design, such as names and values of form elements, certain JavaScript links implementing collapsible menus, and custom style sheets based on user preferences.

For Trustify, our custom web application, we implemented a policy that does not take advantage of the static-dynamic model. Instead, the policy shown in Figure 2.10 takes advantage of Noncespaces's ability to thwart node splitting attacks to implement an ancestry-based sandbox policy similar to the noexecute policy described in BEEP [48]. This policy denies common script-invoking tags and attributes from any namespace (e.g., `<script>` and `onclick`) that are descendants of a `<div>` tag with the `class="sandbox"` attribute. (Note: the policy does not attempt to be exhaustive. It does not enumerate non-standard browser-specific tags and attributes.) To allow the rules to apply to elements and

attributes in any namespace we use the common XPath idiom of matching by each node's `local-name()`.

```
trustclass unclassified

# Blacklist of script-invoking XHTML 1.1 nodes
deny //*[local-name() = 'div' \
          and @*[local-name() = 'class' and . = 'sandbox']]\
       //*[local-name() = 'script']
deny //*[local-name() = 'div' \
          and @*[local-name() = 'class' and . = 'sandbox']]\
       //@*[   local-name() = 'onload'      or local-name() = 'onunload' \
            or local-name() = 'onclick'     or local-name() = 'ondblclick' \
            or local-name() = 'onmousedown' or local-name() = 'onmouseup' \
            or local-name() = 'onmouseover' or local-name() = 'onselect' \
            or local-name() = 'onmouseout'  or local-name() = 'onfocus' \
            or local-name() = 'onblur'      or local-name() = 'onkeypress' \
            or local-name() = 'onkeydown'   or local-name() = 'onkeyup' \
            or local-name() = 'onsubmit'    or local-name() = 'onreset' \
            or local-name() = 'onmousemove' or local-name() = 'onchange' \
            or (local-name() = 'href' \
                and starts-with(ns:tolower(normalize-space(.)), \
                                        "javascript:")) \
            or (local-name() = 'src' \
                and starts-with(ns:tolower(normalize-space(.)), \
                                        "javascript:"))]

# Allow everything else
allow //*
allow //@*
allow //namespace::*
```

Figure 2.10: Example Sandbox Policy. This ancestry-based sandbox policy prohibits potential script-invoking tags and attributes that are descendants of a `<div>` node with the `class="sandbox"` attribute.

For each of the exploits we first verified that each exploit succeeded without Noncespaces enabled. We then enabled Noncespaces and verified that all exploits were blocked

as policy violations.

LifeType Case Study

To gain more insight into the work involved in porting existing applications to Noncespaces, we ported LifeType, a popular blog application, to work with Noncespaces. LifeType is a mature, full-featured blog application consisting of 155K lines of PHP and XHTML code. Enabling Noncespaces required changes to only 180 lines of code. The majority of code changes occurred in LifeType's HTTP header handling. These changes were necessary because Noncespaces needs to include its own headers before any content is sent to the client.

We developed a static-dynamic policy for LifeType that attempts to restrict untrusted content to a minimal set of capabilities. Using our proxy's training mode, it took approximately 4 hours to exercise a significant portion of LifeType's functionality and to manually refine generated rules that were overly general. We then went through our functionality exercise again to ensure that we did not prohibit any legitimate behavior.

To test the effectiveness of our LifeType policy, we introduced XSS vulnerabilities into the application. We used the XSS Cheat Sheet [89] to craft 100 XSS exploits. We then tested each exploit in Opera 9.27[6] Before applying Noncespaces, 50 of the exploits were successful. The remaining 50 exploits were unsuccessful against Opera because they exploit functionality unique to some other browser (such as executing JavaScript by invoking the `mocha:` protocol scheme present in older Netscape versions). After we applied Noncespaces, Noncespaces blocked 98 of the 100 exploits as either policy violations or XHTML parsing errors. These results give us confidence in our policy's ability to recognize exploits

[6]We used Opera for our evaluation due to its native support for namespace qualified attributes.

	Total Exploits	Failed Exploits		Successful Exploits
		Blocked by Noncespaces	Incompatible with Browser	
Without Noncespaces	100	-	50	50
With Noncespaces	100	98	2	0

Table 2.1: Security Analysis Results. This table summarizes the results of our security analysis of Noncespaces-enabled LifeType using a policy developed with our new training mode.

while allowing intended behavior and in Noncespaces's ability to block exploits that target multiple browsers. Since Noncespaces processes exploitable web pages before the browser renders them, many exploits that would have been incompatible with the browser were blocked by Noncespaces before they reached the browser. Neither of the two exploits that were not blocked resulted in a successful XSS attack: one was rendered as text, the other as a comment. That neither exploit caused a policy violation does not indicate a limitation of our approach. Our browser-agnostic prototype proxy implementation targets XHTML compliant browsers, as discussed previously. Neither exploit was valid XHTML.

The latter exploit is an Internet Explorer conditional comment [72]. XHTML compliant browsers will render the comment as a comment and ignore its contents. However, Internet Explorer interprets the comment as HTML code if the specified conditions are met. This exploit illustrates how non-standards-compliant behavior can lead to security vulnerabilities and confirms our preference for eventual in-browser implementation. Only a Noncespaces-aware browser can ensure complete mediation—that all content interpreted as HTML code is checked for conformance to the policy. Table 2.1 summarizes the results.

2.5.2 Performance

Our performance evaluation seeks to measure the overhead of Noncespaces in terms of response latency and server throughput. Our test infrastructure consisted of the applications that we used for our security evaluation running in a VMware virtual machine with 512 MB RAM running Fedora Core 3, Apache 2.0.52, and mod_php 5.2.6. The virtual machine ran on an Intel Pentium 4 3.2 GHz machine with 1 GB RAM running Ubuntu 7.10. Our client machine was an Intel Pentium 4 2 GHz machine with 256 MB RAM running Ubuntu 8.10 Server. These results represent an upper bound on performance penalty as we have spent no effort optimizing our Noncespaces prototype. In each test we used `ab` [1] to retrieve an application page 1000 times. We varied the number of concurrent requests between 1, 5, 10, and 15, and the configuration of the client and server between the following:

- Baseline: measures original web application performance before applying Noncespaces.
- Randomization Only: measures impact of Noncespaces randomization on server without policy validation on client-side.
- Full Enforcement: measures the end-to-end impact of Noncespaces.

We ran three trials with each test configuration against both the TikiWiki and LifeType applications.[7] We report the mean, median, and standard deviation of results over all trials. The server virtual machine was rebooted between tests. The target page was prefetched once before the test to warm up the systems' caches to prevent any one-time costs (such

[7]We do not report performance results for Trustify. We developed Trustify for our security evaluation to exhibit all forms of XSS vulnerability vectors. It is not representative of realistic web application workloads.

as compiling the NSmarty templates) from skewing our results. Our results are shown in Figures 2.11 and 2.12.

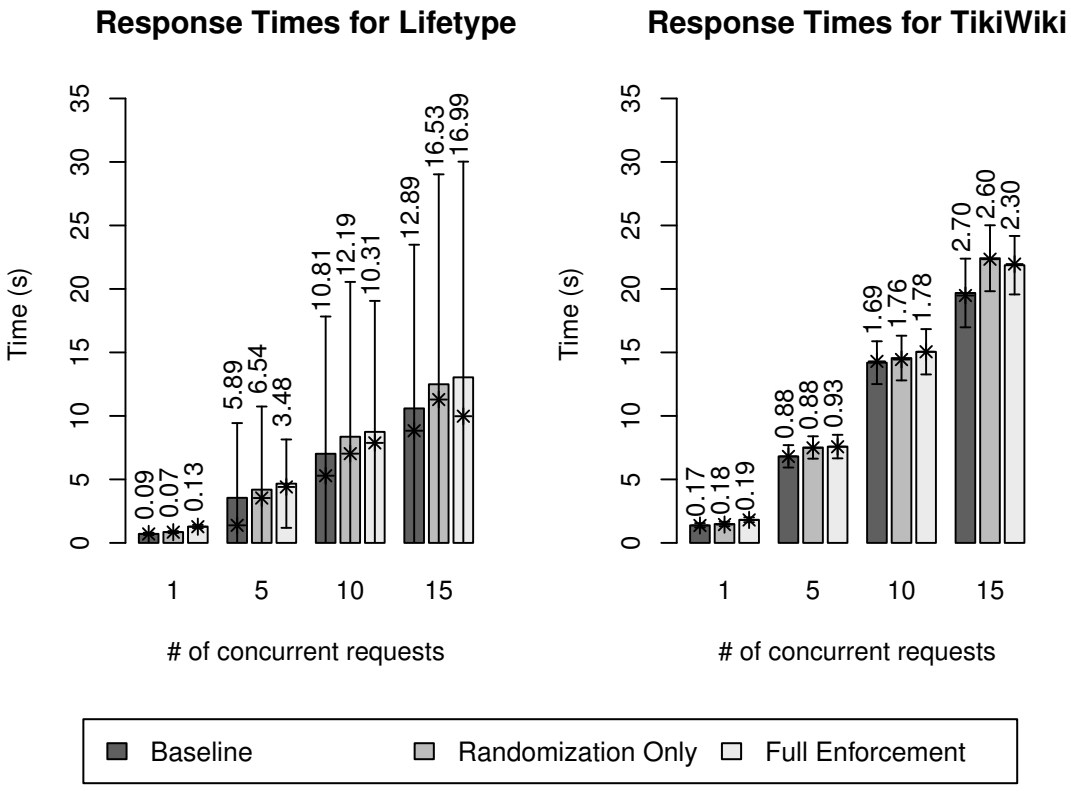

Figure 2.11: Response times for Noncespaces-enabled LifeType and TikiWiki applications. Response times are grouped by the number of concurrent requests. Bars depict the mean value, asterisks the median, and the vertical line segments show the mean plus or minus the standard deviation. The value of the standard deviation is shown by label above each line segment.

The graphs of response latency show that enabling Noncespaces randomization on the server increased response time by (at most) 14% for TikiWiki and 20% for LifeType. Enabling the policy checking proxy resulted in response times that were (at most) 32% higher than the baseline response time for TikiWiki and 80% higher for LifeType. Though the

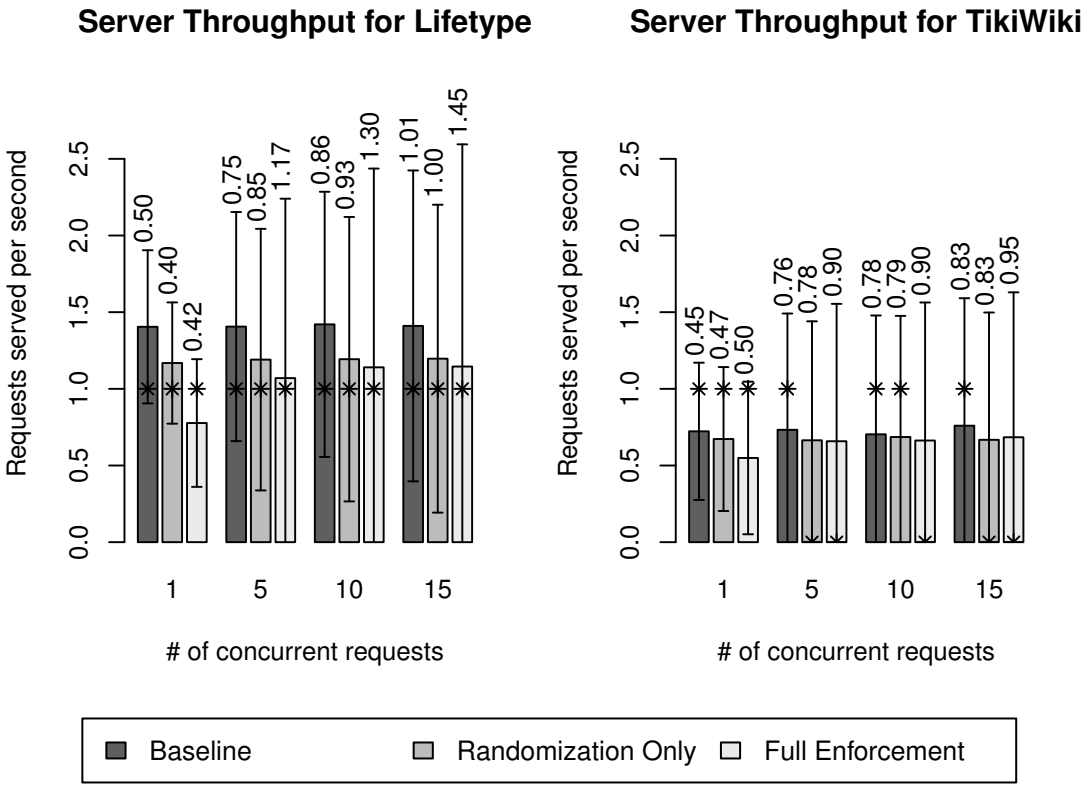

Figure 2.12: Throughput for Noncespaces-enabled LifeType and TikiWiki applications. Throughput rates are grouped by the number of concurrent requests. Bars depict the mean value, asterisks the median, and the vertical line segments show the mean plus or minus the standard deviation. The value of the standard deviation is shown by label above each line segment.

overhead may appear significant at first glance, during interactive use latency typically increased by no more than 0.6 seconds.

We also examine the effect of Noncespaces on server throughput. With randomization enabled throughput is reduced by about 10% for TikiWiki and 20% for LifeType. After enabling policy checking, the throughput of both TikiWiki and LifeType decreases by an additional 3% for higher numbers of concurrent requests. Because policy checking is per-

formed on the client side the effect of policy checking on server throughput is minimized when multiple clients make requests simultaneously.

2.6 Related Work

Given the high impact of XSS attacks, there is a significant amount of related work. Here we attempt to compare our work with a sampling of other relevant work.

Randomization Our work was inspired by Instruction Set Randomization (ISR) [49, 7] — a technique for defending against code injection attacks in executables by randomly remapping a computer's instruction set architecture. SQLrand [15] first employed randomization to defeat SQL Injection attacks. Noncespaces is an analogous approach that protects web applications from Cross-Site Scripting attacks. ISR and SQLrand prevent an attacker from injecting meaningful code and SQL keywords by forcing an attacker to guess a random mapping. Noncespaces also forces an attacker to guess a random mapping in order to inject trusted content. ISR and SQLrand consider static program code and SQL queries trustworthy. Similarly, our prototype considers all static (X)HTML template content to be trustworthy. Unlike ISR or SQLrand, Noncespaces must support web applications which permit rich user input. Therefore, Noncespaces expands the ISR approach by using a configurable policy to constrain the capabilities of untrusted content on the client side.

Preserving Document Structure Integrity Document Structure Integrity (DSI) [75] was developed independently and contemporaneously with our work. Each system has advantages over the other in different areas. Like Noncespaces, DSI uses randomized de-

limiters to allow a web browser to distinguish between trusted and untrusted content. In DSI, the server identifies untrusted content using a prototype taint tracking implementation for PHP. The browser enforces a simple policy that limits untrusted content to terminals in (X)HTML and JavaScript and to tags and attributes whitelisted on a per-page basis. DSI also augments the browser with information flow tracking in order to defeat DOM-based (Type III) XSS attacks. Noncespaces's policy language is more expressive than the policy language provided by DSI. DSI's policy language does not capture the position of whitelisted elements or provide an ability to constrain terminal values. Both of these capabilities are important for defeating injection of non-script elements. In many scenarios, non-code injection attacks can be just as dangerous as code injection attacks [25]. For instance, an attacker can steal login credentials by injecting a fake login form onto a bank's website. The whitelisting approach employed by Noncespaces is superior to the blacklisting approach employed by DSI (called "minimal-serialization") insomuch as it respects the Principle of Fail-Safe Defaults. If the classification mechanism is incomplete, Noncespaces would classify trusted data as untrusted and therefore might refuse to render a legitimate page, but DSI would classify untrusted data as trusted, resulting in security vulnerabilities.

Before Noncespaces and DSI, work on ensuring document structure integrity typically focused on ensuring that output documents were valid [51] and that web designers would not inadvertently print unsanitized output to the browser [41]. However, neither approach is sufficient to mitigate XSS attacks.

The Noncespaces and DSI approach of defeating XSS by ensuring document structure integrity has appeared in subsequent research. Blueprint [102] provides a DSI-like terminal confinement and no script policy mechanism for unmodified modern browsers. An application developer manually annotates all web application statements that output untrusted con-

tent. Untrusted content will then be transmitted to the browser where client-side JavaScript ensures that it cannot invoke the JavaScript interpreter. By contrast, Noncespaces does not require the developer to manually identify and modify untrusted output statements. Noncespaces integrates with the Smarty template engine, allowing it to intercept all untrusted template outputs automatically. Because Blueprint only encodes untrusted content, incomplete sanitization can result in successful XSS attacks. Noncespaces encodes trusted data to ensure that any failure of complete mediation will not result in a successful attack. Noncespaces also provides a significantly more flexible policy mechanism.

SWAP [116] takes a complementary approach on the server-side — it attempts to whitelist all trusted scripts. SWAP identifies all static scripts and replaces them with non-executable script identifiers. Before delivering the response to the browser, SWAP invokes a server-side script detector (consisting of a browser residing on the server) to determine if any scripts have been injected. If no scripts are detected, the script identifiers in the response are replaced with the original scripts and the response is delivered to the client. Both Blueprint and SWAP incur higher server-side overheads than Noncespaces because they perform all policy enforcement on the server, preventing clients from sharing the computational burden. Noncespaces's client-side enforcement also allows for the possibility of browser or user contributed policies. Also, like DSI, Blueprint and SWAP do not defend against non-script attacks.

Alhambra [101] is a pragmatic approach to ease deployment of a document structure integrity system. It is an entirely client-side mechanism that attempts to infer a document structure integrity policy from multiple web page visits. Client-side information flow tracking is also employed to prevent DOM-based XSS attacks and certain common script abuse patterns. Alhambra's learning capabilities can be compared to a more advanced rendition

of Noncespaces's training mode. Greater learning algorithm intelligence is necessary for Alhambra because, unlike the training environment in Noncespaces, the resulting policy is not reviewed by an expert before being employed to defend against attacks. Alhambra's client-side only approach imposes several additional challenges including the ability of a malicious attacker to mislead the policy learner and attempting to remain robust to legitimate website modifications.

Robertson and Vigna present another approach for ensuring the structural integrity of a document by defining a strongly-typed web application framework [87]. Instead of generating unstructured strings, applications output an Abstract Syntax Tree. The AST is then translated by a trusted renderer which ensures that all content incorporated into document terminals is correctly escaped. This prevents attacks which rely on violating the document's structural integrity. In addition to defeating attacks that violate structural integrity, Noncespaces goes one step further by protecting against attacks which leverage unsafe attribute values and vulnerabilities caused by web applications that permit an untrusted user to create security sensitive structural content.

Client-side Policy Enforcement Client-side policy enforcement mechanisms enforce a security policy in the browser to avoid the semantic gap between the client and server. BEEP [48] allows a server-specified JavaScript security handler to decide whether to permit or deny the execution of each script based on a programmable policy. The BEEP authors present two example policies: an ancestry-based sandbox policy, which prohibits scripts that are descendants of a sandbox node, and a whitelist policy, which allows a script to execute only if it is known-good. Mutation Event Transforms [37] extend the mechanism of BEEP to all operations that modify the DOM. Before each DOM modification, a JavaScript

callback is invoked that can allow, deny, or arbitrarily change the operation performed.

Noncespaces is similar to both of these approaches in that the server delivers a policy that the client enforces. Like BEEP, our policy language is able to express both ancestry-based sandbox and whitelist policies. Additionally, like Mutation Event Transforms, our policy language is also able to express policies which constrain non-script content of a web page. This is important because, as mentioned above, malicious non-script content can successfully exploit security vulnerabilities. It would have been possible to leverage Mutation Event Transforms as our client-side policy mechanism. However, giving policies the full power of JavaScript would make it hard to reason about a policy's effects and could also provide a new vector for bugs and vulnerabilities to be introduced. This is why our client-side policy language consists of simple rules that match nodes and attributes in the DOM and declare whether they should be allowed or denied. We also note that the main contributions of our work are a mechanism for reliably communicating trust information from server to client and leveraging properties of the web application to determine trustworthiness of content automatically. Neither BEEP nor Mutation Event Transforms addresses these issues.

Noncespaces is also closely related to Content Restrictions [66], Mozilla's Content Security Policy (CSP) [98], Script Keys [65], and Brendan Eich's proposed `<jail>` tag [36]. Content Restrictions allow the server to specify certain restrictions on the content that it delivers, such as: whether scripts may appear in the document body, header, only externally, or not at all; which hosts resources may be fetched from; which hosts scripts may be fetched from; etc. Mozilla's Content Security Policy is an implementation of Content Restrictions for Firefox with some additional features. Noncespaces client-side policies are able to specify most of the same restrictions as Content Restrictions and CSP. CSP

provides a few features outside the domain of Noncespaces. However, Noncespaces can prevent malicious content from attacking trusted origins or exfiltrating data to untrusted domains via forms and links, but CSP cannot. Content Restrictions and CSP also do not provide a mechanism for differentiating between server-trusted content executing a script in an approved location or injected content doing the same. Both Script Keys and Noncespaces provide a way to differentiate between the two scenarios.

Script Keys prohibits scripts from running unless they include a server-specified key in their source. The proposed `<jail>` tag does just the opposite: it prohibits active content in the document subtree below it and uses a nonce embedded in the opening and closing tags to prevent a node-splitting attack from closing a `<jail>` tag prematurely. In the limit, when the script key is changed on every page load, Script Keys behaves like Noncespaces — the attacker must guess the randomly generated key for each request to enable their script to run. To an extent, Noncespaces can be seen as the first implementation of Content Restrictions, Script Keys, and the `<jail>` tag. However, none of these other proposals provide a means to restrict non-script content with the same level of precision as Noncespaces. Noncespaces and CSP may be useful in conjunction with one another allowing specification of policy constraints at the most natural layer.

ConScript [71] enables client-side policy enforcement for JavaScript code by providing an Aspect Oriented Programming model for JavaScript. Noncespaces's client-side policy enforcement and ConScript are orthogonal. Noncespaces determines whether or not a tag or attribute should be added to the DOM. ConScript constrains the behavior of scripts after that point. ConScript also only effects the execution of scripts and thus does not help defend against non-script attacks.

Prohibiting Anti-Patterns Two main goals of XSS attacks are stealing the victim user's confidential information and invoking malicious operations on the user's behalf. Noxes provides a client-side web proxy to block URL requests by malicious content using manual and automatic rules [50]. Vogt et al. track the flow of sensitive information in the browser to prevent malicious content from leaking such information [112]. Both of these projects defeat only the first goal of XSS attacks. By contrast, Noncespaces can defeat both goals of XSS attacks because it prevents malicious content from being rendered. Internet Explorer 8's XSS filter [88] attempts to prevent reflected XSS attacks by disabling scripts that embed certain substrings seen in an outgoing request. This only prevents reflected XSS attacks while Noncespaces is also able to prevent stored XSS attacks. Attackers have also been able to leverage the semantic gap between the server and the client to cause IE 8's XSS filter to create new security vulnerabilities [77] illustrating the need for and end-to-end solution such as Noncespaces.

Leveraging Language Techniques SQLCheck [100] defines a sub-language of SQL that untrusted user input can safely express. In theory, the same approach can be employed to prevent XSS attacks; however, specifying an appropriate subset of (X)HTML is more difficult because web application security policies cut across multiple languages (including JavaScript, URIs, and CSS), can depend on position in the document hierarchy, and may depend on specific terminal values. Therefore, we believe that expressing a policy in terms of XPath expressions is more straightforward for web developers than modifying a unified grammar for web pages in order to restrict untrusted input to a suitably safe subset for each web application.

Static Analysis Numerous papers [114, 115, 117, 60] have employed static program analysis to detect XSS vulnerabilities. Static analysis approaches cannot be both sound and complete, forcing a choice between false positives or missed vulnerabilities. By favoring a dynamic analysis approach, Noncespaces avoids loss of precision due to round-trips to the browser and difficult to support PHP features. Our use of NSmarty to determine trust classifications is a conservative approximation; however, if greater precision is needed, our technique can be integrated with a full information flow tracking system.

Information Flow Tracking A number of different information flow (or taint) tracking solutions for web applications have appeared in the literature [78, 111, 118]. Unfortunately, none of the solutions for PHP have seen widespread use. This prompted our development of NSmarty to simply and conservatively approximate information flow by leveraging a common web application programming paradigm. The encoding and client-side policy enforcement mechanisms that Noncespaces provides can use a mature information flow tracking system as a content classifier, when one arises.

A preliminary version of this work was presented at NDSS 2009 [109]. Since then we have extended the policy language to support hierarchical trust classes and XML name-space-specific policy rules, which are important for handling real-world web applications. During the research for [109], we found that it could be difficult to create complete policies for large web applications. Therefore, we have implemented a training mode to facilitate policy development (Section 2.4.3). Finally, to demonstrate the effectiveness, usability, and backward compatibility of Noncespaces, we ported a large web application to Noncespaces and conducted a more extensive security evaluation using a policy developed with our new training mode (Section 2.5.1).

Chapter 3

Multi-party Off-the-Record Messaging

3.1 Motivation

In contrast to traditional thick-client applications, modern web application architectures typically require the web server to be fully trusted. The web server delivers the software which runs in a user's browser on-demand, giving users little ability to control changes in or verify the integrity of the client-side application. In situations where users require end-to-end security guarantees, that is, intermediate servers are not trusted, peer-to-peer or traditional client-server applications running specialized protocols are common. The Internet has popularized a novel means of communication, instant messaging (IM), where users can engage in interactive conversations across great distances. However, common IM protocols lack certain fundamental security properties that a physical private conversation can provide. Impersonation, eavesdropping and information copying are all trivial attacks over IM making it unsuitable for sensitive conversations.

Online communication systems commonly provide three properties: confidentiality,

authentication and non-repudiation. Confidentiality and authentication are expected traits of face-to-face conversations, but non-repudiation clashes with the expectations for private communication. Non-repudiation denotes a receiver's ability to *prove* to a third party, possibly a judge, that the sender has authored a message. Although desirable under many circumstances, non-repudiation is the very property journalists, dissidents or informants wish to avoid. [45, 47, 38]

Borisov, Goldberg and Brewer [13] argued that instant messaging should mimic casual conversations. Participants of a casual talk can deny their statements in front of outsiders, and can sometimes deny having taken part in the talk at all. The authors presented a mechanism called *Off-the-Record Messaging* (OTR) that allows two-party private conversations using typical IM protocols. OTR aims to provide confidentiality, authentication, repudiation and forward secrecy, while being relatively simple to employ.

Despite its good design, OTR has limitations, the most important of which is that it can serve only two users. Hence it is not suitable for online multi-party conversations, commonly enjoyed by casual users via Internet Relay Chat (IRC), by open-source software developers, and by businesses that cannot afford confidential meetings across vast distances [10, §2.3]. It is non-trivial to extend OTR to allow for multi-party conversations, as OTR uses cryptographic primitives designed for two parties. For example, OTR uses message authentication codes (MACs) to provide authenticity. While for two parties MACs can provide a deniable authentication mechanism, MACs do not provide origin authentication when used by more than two parties.

Bian, Seker and Topaloglu [10] proposed a method for extending OTR for group conversation. The crux of their solution is to designate one user as the "virtual server". While this may be feasible under certain circumstances, it deviates from the original OTR goal,

which is to mimic private conversations. In private group conversations there is no virtual server responsible for smooth meetings. Moreover, the server becomes an enticing target for malicious parties. Finally, the server has to be *assumed* honest, as a dishonest server could compromise both the confidentiality and the integrity of all messages sent during a chat session.

In this work, we present a multi-party off-the-record protocol (mpOTR), which provides confidentiality, authenticity and deniability for conversations among an arbitrary number of participants. Using our protocol, an ad hoc group of individuals can communicate interactively without the need for a central authority. We identify the important traits of multi-party authentication for users, for messages *and* for chatrooms that share users; that is, we take into account that two or more users may concurrently share more than one chatroom with different peers. When considering privacy properties, we allow malicious insiders and identify their goals. These multi-party chatroom properties present new challenges that were not addressed in previous work.

An OTR transcript reveals that a user at some point communicated with someone. Our mpOTR protocol carries deniability further by allowing the user to deny everything except, by virtue of being part of the system, that they were willing at some point to engage in a conversation. In fact, it is unclear how users can deny the latter at all because by using the Internet, they already indicate their intent and ability to engage with others. In this sense, mpOTR is closer than OTR to simulating deniability in private conversations in the physical world: anyone could take or have taken part in a private conversation, but that person can plausibly deny ever having done so.

3.1.1 Related work

While not the first to address security in instant messaging, Borisov, Goldberg, and Brewer [13] popularized its privacy aspects, partly due to their now-popular open-source plug-in. Subsequently, more research was devoted to IM; in fact, the original proposal was found to contain errors [33], which were repaired in a subsequent version of OTR.

On a high level there are two approaches to secure IM. Clients can establish their connections via a centralized server and rely on the server for security and authentication [64]. Alternatively, participants can use shared knowledge to authenticate each other [2]. OTR, which aims to simulate casual conversations, is closer to the second solution.

While there is a wide literature on IM (see [63, §2.1] for an extensive list), little research has focused on the *multi-party* privacy aspects of instant messaging. To our knowledge, before this work the only published work with the explicit goal of achieving group off-the-record conversations is the aforementioned result by Bian, Seker and Topaloglu [10]. It has traits of Mannan and Van Oorschot's work on two-party IM [64] in the sense that a designated user acts as a server. In some cases, e.g. the Navy [26], it may be easy to establish a superuser whom everyone trusts, but if the goal is a casual off-the-record chat or users are unwilling to trust each other, agreeing on a server user becomes problematic. We adopt the scenario where all users are equal.

3.1.2 Outline

In §3.2 we identify the relevant properties of private meetings and how they apply to IM. §3.3 describes the different players of our model for private communication; we focus on the different adversaries and their goals. §3.4 outlines our solution at a high level and shows

that we have achieved the goals of private conversations. Section §3.5 outlines directions for future work.

3.2 Private chatrooms

3.2.1 Confidentiality

In meetings a user $\hat{A}$ is willing to reveal information to chatroom members but not outsiders. Hence chatroom messages need to remain *hidden* from the wider community. In private physical communication, should a new party approach, the participants can "detect" the newcomer and take appropriate actions.

On the Internet eavesdropping cannot be detected as easily; however, there are ways to guard against casual eavesdroppers. Cryptographic algorithms can assure parties that observers looking at the transmitted conversation packets are left in dark about the communicated content. That is, the transcript gives an eavesdropper no additional knowledge above information about lengths of messages and traffic patterns, beyond what the eavesdropper could have deduced without having seen the encrypted messages.

3.2.2 Entity authentication

In a face-to-face meeting we identify peers via their appearances and physical attributes. By contrast, on the Internet a user proves to another user knowledge of some secret identifying information, a process known as *entity authentication*.

The basic goal of entity authentication is to provide evidence that a peer who presents public key $S_{\hat{B}}$ also holds the corresponding private key $s_{\hat{B}}$. For example, suppose Alice

provides a challenge to Bob. If Bob can compute a response that can only be computed by an entity possessing $s_{\hat{B}}$, then Bob successfully authenticates himself to Alice. This type of authentication is limited in the sense that Bob only shows knowledge of $s_{\hat{B}}$. If Bob wants to claim any further credentials like "classmate Bob", then Alice would need additional proofs. Two-party entity authentication has been studied in the setting of OTR by Alexander and Goldberg [2, §4 and §5]; their solution is suitable for pairwise authentication.

The entity authentication goal for mpOTR is to provide a consistent view of chatroom participants: each chat participant should have the same view of the chatroom membership. We achieve this goal by first requiring users to authenticate pairwise to each other. Then users exchange a short message about who they think will take part in the chat. Alternatively, a suitable n-party authentication primitive could be used to authenticate all users to each other simultaneously.

Authentication is challenging. In a centralized approach, if a malicious party successfully authenticates to the server, the security of the whole chatroom is compromised. The problem is more evident when the server itself is malicious. In our approach, parties do not rely on others to perform faithful authentication. All parties check to ensure that no party has been fooled. While we do not provide means to prevent malicious parties from joining a chat, users can leave a chat if they wish. In other words a malicious party may join a chat with a given set of honest participants only if all the honest participants approve his entrance.

3.2.3 Origin authentication

Each message has a well-defined source. The goal of origin authentication is to correctly identify the source. First of all a user must be assured that the message is sent from some-

one who legitimately can author messages in the chatroom. In OTR if Alice is assured that a valid OTR peer sent a message and that peer is not Alice herself, then she knows Bob sent the message and that only she and Bob know the message[1]. In mpOTR if both Bob and Charlie are chat participants, Alice should be able to distinguish messages authored by Bob from messages authored by Charlie. She should also be able to identify origins with respect to chatrooms: if Alice and Charlie are both members of chatrooms $\mathcal{C}_1$ and $\mathcal{C}_2$, then when Alice receives a message from Charlie in $\mathcal{C}_1$, no one should be able to fool her that the message was sent in $\mathcal{C}_2$. In this way Alice is aware of who else sees the message.

Message authentication should be non-repudiable among chat participants in order to allow honest users to relay messages between one another or to expose dishonest users who try to send different messages to different parties. Alice should have the ability to convince Bob, or any other chat member, that a message she accepted from Charlie indeed was sent by Charlie. A word of caution: transferability introduces a subtlety when combined with our deniability requirement. Alice's ability to convince Bob that Charlie authored a message must not allow her to convince Dave, who is not a chat participant, that Charlie authored the message.

3.2.4 Forward secrecy

The Internet is a public medium: when a typical user sends a data packet, the user has little (if any) idea how the packet will reach its destination. To be on the safe side we assume that the adversary has seen and recorded every transmitted packet for future use. The adversary's ability to see messages motivates encryption; his ability to record messages motivates forward secrecy. Forward secrecy implies that the leakage of static private keys

[1] We assume no "over-the-shoulder" attacks.

do not reveal the content of past communication. Users achieve forward secrecy by using ephemeral encryption and decryption keys that are securely erased after use and that cannot be recomputed even with the knowledge of static keys.

We separate encryption keys from static keys. Static keys are used to authenticate ephemeral data, which is used to derive short-lived encryption keys. This is a common approach to achieve forward secrecy. Note that this goal is unrelated to deniability: in forward secrecy the user does not aim to refute any message; in fact, the user may not even be aware of the malicious behavior. The goal of the adversary is reading the content of a message as opposed to associating a message with a user.

3.2.5 Deniability

A casual private meeting leaves no trace[2] after it is dissolved. By contrast, the electronic world typically retains partial information, such as logs for debugging, for future reference, and so on. This contradicts the "no trace" feature of private meetings. As mentioned in the forward secrecy discussion, entities involved in relaying messages may keep a communication record: participants do not and cannot control all copies of messages they send and hence cannot be assured that all copies were securely destroyed. Users can claim the traces are bogus, effectively denying authoring messages. But what is the meaning of "deny" in this context?

Not all deniability definitions are suitable for off-the-record communication. Consider for example the plaintext deniability notion proposed in [22], where the encryption scheme allows a ciphertext author to open the ciphertext into more than one plaintext: Alice wishes to communicate (possibly incriminating) message M_1; she chooses $M_2, \ldots, M_n$ non-

[2] If there were no logs, wiretapping, etc.

incriminating messages and then forms the ciphertext as follows: $C = DeniableEncrypt_K(M_1, M_2, \ldots, M_n)$. When challenged to decrypt, Alice can validly decrypt C to any of the alternate messages M_i that she chose when forming C. Even though Alice denies authoring the incriminating plaintext, she implicitly admits to authoring the ciphertext. However, who you speak to may be as incriminating as what you say. Alice might get into deep trouble with her mafia boss by merely admitting that she has spoken with law enforcement, regardless of what she said. She would be in a much better situation if she could claim that she never *authored* the ciphertext, instead of decrypting the ciphertext to an innocuous plaintext and thereby implicitly admitting authorship.

Contrary to the above example, suppose Alice has means of denying all her messages in front of everyone, by arguing that an entity different from herself faked messages coming from her[3]. That is, any message purportedly from Alice could have been authored by Mallory. In that case how could Bob and Charlie have a meaningful conversation with Alice? They have no assurances that messages appearing to come from Alice are indeed hers: Alice's messages can be denied even in front of Bob and Charlie. What we need is a "selective" deniability. We next discuss the selectiveness of deniability in the requirements for multi-party Off-the-Record messaging.

Repudiation

The *fundamental problem of deniability* (FPD) describes the inherent difficulty for a user Alice to repudiate a statement. Let Justin be a judge. Suppose Charlie and Dave come to Justin and accuse Alice of making a statement m. In the best-case scenario for Alice, Charlie and Dave will not be able to provide any evidence that Alice said m, apart from

[3]For example Alice can pick a symmetric encryption key κ encrypt her message with κ, encrypt κ with Bob's public key and send everything to Bob.

their word. If Alice denies saying m, whom should Justin trust: Charlie and Dave, or Alice? The voices are two to one against Alice, but it is possible that Charlie and Dave are trying to falsely implicate Alice. Justin must decide who to believe based on his evaluation of the trustworthiness of their testimony. Justin's evaluation may be influenced by many hard-to-quantify factors, such as perceived likelihood of the testimony, the number of witnesses in agreement, potential benefits and detriments to the witnesses, etc. Justin may even explicitly favor the testimony of certain witnesses, such as law enforcement officers. In the end, Justin must base his decision on weighing the testimonies rather than on physical evidence. In the limit, when n parties accuse Alice of saying m, Alice will have to convince Justin that the other n witnesses are colluding to frame her. We call this the fundamental problem of deniability.

We cannot solve the FPD. The best we can offer is to ensure that Charlie and Dave cannot present any evidence (consisting of an algorithmic proof) that Alice has said m, thereby reducing the question of Alice's authorship to the FPD. In the online world Charlie and Dave make their claim by presenting a communication transcript. Therefore, we provide Alice with means to argue that Charlie and Dave could have created the transcript without her involvement. As long as Charlie and Dave cannot present an algorithmic proof of Alice's authorship, she can plausibly deny m, so Justin has to rule based on the same factors (e.g., weighing the testimonies rather than on physical evidence) as in the physical world. By this means, we provide comparable levels of repudiation between on-line and face-to-face scenarios.

In §3.2.3 we alluded to the conflicting goals of message origin authentication and privacy, where the complete deniability example prevents origin authentication. To provide origin authentication, we need a special type of repudiation. Let us look closer at a private

communication among Alice, Charlie and Dave. In a face-to-face meeting Charlie and Dave hear what Alice says. This is origin authentication. After the meeting, however, Alice can deny her statements, because, barring recording devices, neither Charlie nor Dave has evidence that Alice made any specific statement. This is the type of repudiation that we aim for.

In contrast to the physical world, on the Internet Charlie can differentiate between Alice and Dave when the three of them are talking and can send them different messages. While it is impossible to guard against such behavior (either due to malicious intent or connection problems), we would like the proof of authorship that Charlie provides to Alice to also convince other chat participants — and no one else — of his authorship. That way, all parties are assured that (1) they can reach a transcript consensus even in the presence of malicious behavior, and (2) all statements within the chat can be denied in front of outside parties. This condition should hold even if Alice and Charlie share more than one chat concurrently or sequentially: all chats must be independent in the sense that if Alice and Charlie share chats $\mathcal{C}_1$ and $\mathcal{C}_2$ any authorship proof Charlie has in $\mathcal{C}_1$ is unacceptable in $\mathcal{C}_2$. In relation to the previous paragraph we note that such authorship proofs should also become invalid at the end of the meeting.

Forgeability

In some cases[4] it is valuable to deny not only having made a statement but also having participated in a meeting. In the physical world Alice can prove she was absent from a meeting by supplying an alibi. On the Internet, however, such an alibi is impossible as Alice can participate in multiple chatrooms simultaneously. Short of an alibi, the next

[4]Police informants, for example.

best denial is a design where transcripts allegedly involving Alice can be created without her participation. Although this mechanism would not allow Alice to prove that she was absent from the meeting, it prevents her accusers from proving that she was present at the meeting. A refinement is to design transcripts that can be extended to include users that did not participate, to exclude users who did participate, or both. Effectively, such transcripts will offer little, if any[5], evidence about who participated in it.

For example, suppose Mallory tries to convince Alice that Bob spoke with Dave and Eve by presenting a transcript with participants Bob, Dave, and Eve. Ideally, forgeability would allow Bob to argue that Mallory fabricated the transcript even though Mallory is an outsider with respect to the transcript.

Malleability

While, forgeability allows the creation of valid transcript "out of thin air" without the help of the purported participants, malleability allows the contents of authentic transcripts to be denied. Ideally, the transcript should be malleable in the sense that given a transcript $\mathbb{T}^1$ and a message m_1 that belongs to $\mathbb{T}^1$, it is possible to obtain a transcript $\mathbb{T}^2$, where message m_1 is substituted with message m_2. Along with forgeability this approach provides a strong case for users who wish to deny statements and involvement in chat meetings. For accusers, transcripts with this level of flexible modification provide little convincing evidence, even in the event of confidentiality breaches.

[5]If the plaintext is recovered, the writing style or statements made may reveal the author's identity.

3.2.6 Anonymity and pseudonymity

While in our current work anonymity is not the main goal, we desire that our solution preserves anonymity. This includes, but is not restricted to, not writing users' identities on the wire. While we do not explicitly address it, users may wish to use our protocol over a transport protocol that provides pseudonymity. If they do so, it would be unacceptable if our protocol deanonymizes users to adversaries on the network. We do, however, use anonymity-like techniques to achieve a subset of our deniability goals.

3.3 Threat model

3.3.1 Players

We will first introduce the different players and their relations with each other. The set of users, denoted by $\mathcal{U}$, is a collection of entities that are willing to participate in multi-party meetings. Honest parties, denoted by $\hat{A}, \hat{B}, \hat{C}, \ldots$ follow the specifications faithfully; these parties are referred to as Alice, Bob, Charlie, ... Dishonest parties deviate from the prescribed protocol. Each party $\hat{A}$ has an associated long-lived static public-private key pair ($S_{\hat{A}}$,$s_{\hat{A}}$). We assume that the associated public key for each party is known to all other parties. (These associations can be communicated via an out-of-band mechanism or through authentication protocols as in [2].) A subset $\mathcal{P}$ of users can come together and form a chatroom $\mathcal{C}$; each member of $\mathcal{P}$ is called a *participant* of $\mathcal{C}$. While honest users follow the protocol specifications, they may observe behavior that is not protocol compliant due to either network failures, intentional malicious behavior by other parties, or both.

In addition to users that take part in the conversation we have three types of adversaries:

(i) a confidentiality adversary, denoted by O; (ii) a consensus adversary, $\mathcal{T}$; and (iii) a privacy adversary, $\mathcal{M}$. The last player in the system, the judge $\mathcal{J}$, does not interact with users but only with adversaries, in particular with $\mathcal{M}$. We will see his purpose when discussing the adversaries' goals.

3.3.2 Goals

Honest users wish to have on-line chats that emulate face-to-face meetings. It is the presence of the adversaries that necessitates cryptographic measures to ensure confidentiality and privacy.

Confidentiality adversary

The goal of the confidentiality adversary is to learn information about plaintext messages that he is not entitled to. Let $T_{\mathcal{C}_1} = \left\{ \mathbb{T}^{\hat{X}}_{\mathcal{C}_1} \mid \hat{X} \in \mathcal{P} \right\}$ be a collection of transcripts resulting from a chat $\mathcal{C}_1$ with set of chat participants $\mathcal{P}$, such that no user in $\mathcal{P}$ revealed private[6] information to, or collaborated with, the confidentiality adversary O prior to the completion of $\mathcal{C}_1$. Suppose also that for each honest participant $\hat{A}$, who owns $\mathbb{T}^{\hat{A}}_{\mathcal{C}_1} \in T_{\mathcal{C}_1}$[7], and for each honest participant $\hat{B}$, who owns $\mathbb{T}^{\hat{B}}_{\mathcal{C}_1} \in T_{\mathcal{C}_1}$, $\hat{A}$ and $\hat{B}$ have consistent view of the messages and participants. We say that O is successful if O can learn partial information[8] about at least one message in some $\mathbb{T}^{\hat{A}}_{\mathcal{C}_1}$ without obtaining the message from a user $\hat{B}$ who owns $\mathbb{T}^{\hat{B}}_{\mathcal{C}_1}$.

A few remarks on O's goals are in order. The confidentiality adversary can control

[6] Either static private keys or $\mathcal{C}_1$-related information.

[7] That is, user $\hat{A}$ did take part in $\mathcal{C}_1$, and in particular $\hat{A} \in \mathcal{P}$.

[8] Partial information in the sense of traditional ciphertext indistinguishability. The adversary should not be able to distinguish between two unequal plaintexts that do not differ in length, author, or order.

communication channels and observe the actions of any number of users in $\mathcal{P}$, learn messages that they broadcast in other chatrooms, and start chatroom sessions with them via proxy users. All these actions can take place before, during or after $\mathcal{C}_1$. However, O is allowed neither to ask for static private information of any user in $\mathcal{P}$ before the completion of $\mathcal{C}_1$ nor to take part in $\mathcal{C}_1$ via a proxy user. The adversary may ask an honest user to send messages to $\mathcal{C}_1$, but should still be unable to decide if or when his request is honored. Essentially, O aims to impersonate an honest user during key agreement or to read messages in a chatroom that consists only of honest users. O's capabilities are similar to the standard notion of indistinguishability under chosen-plaintext attack for encryption schemes [8].

Consensus adversary

We first explain the meaning of consensus, which relates to what Alice thinks about her and Bob's view of past messages. We say that $\hat{A}$ reaches consensus on $\mathbb{T}^{\hat{A}}_{\mathcal{C}_1}$ with $\hat{B}$ if $\hat{A}$ believes that $\hat{B}$ admits having transcript $\mathbb{T}^{\hat{B}}_{\mathcal{C}_2}$ [9] such that:

1. $\mathcal{C}_1$ and $\mathcal{C}_2$ have the same set of participants;
2. $\mathcal{C}_1$ and $\mathcal{C}_2$ are the same chatroom instance;
3. $\mathbb{T}^{\hat{B}}_{\mathcal{C}_2}$ has the same set of messages as $\mathbb{T}^{\hat{A}}_{\mathcal{C}_1}$;
4. $\mathbb{T}^{\hat{B}}_{\mathcal{C}_2}$ and $\mathbb{T}^{\hat{A}}_{\mathcal{C}_1}$ agree on each message's origin.

At the end of the meeting (or at predefined intermediate stages) honest users attempt to reach consensus with each other about the current transcript. Our consensus definition allows the possibility that Alice reaches a consensus with Bob but Bob does not reach

[9] By admitting this transcript $\hat{B}$ admits taking part in $\mathcal{C}_2$.

consensus with Alice: for example if either Bob or Alice goes offline due to network failure before protocol completion. We also allow the application to interpret "same set of messages" appropriately for its setting. For instance, the importance of message delivery order may vary by application.

The goal of the consensus adversary $\mathcal{T}$ is to get an honest user Alice to reach consensus with another honest user Bob on a transcript $\mathbb{T}_{C}^{\hat{A}}$, while at least one consensus condition is violated; that is, $\mathcal{T}$ wins if (honest) Alice believes that (honest) Bob has a transcript matching hers (in the above sense), but in fact Bob does not have such a transcript. Note that while Alice and Bob are honest users there is no restriction on the remaining chat members — they may even be $\mathcal{T}$-controlled, which is an improvement over KleeQ [84] where all parties are assumed honest. Resilience against $\mathcal{T}$ implies that users cannot be forced to have different views of exchanged messages and no messages can be injected on behalf of honest users without being detected.

Our consensus definition captures both the standard notions of entity and origin authentication and the adversary's abilities to make conflicting statements to different participants in the same chat session (as described in §3.2.5) as well as to drop, duplicate, reorder, and replay messages from other chat sessions.

Privacy adversary

The goal of the privacy adversary $\mathcal{M}$ is to create a transcript $\mathbb{T}_{C_1}^{\hat{A}}$ to convince the judge $\mathcal{J}$ that $\hat{A}$ took part in C_1 and/or read and/or authored messages in $\mathbb{T}_{C_1}^{\hat{A}}$. The only restriction is that $\mathcal{J}$ is not directly involved in C_1. This is perhaps the hardest adversary to guard against as $\mathcal{M}$ has few restrictions: $\mathcal{M}$ can interact in advance with $\mathcal{J}$ before C_1 is established and, by taking part in C_1, can obtain consensus with respect to $\hat{A}$. Furthermore, the judge can

force $\hat{A}$ as well as all other participants to reveal their long-term secrets. If under such a powerful adversary and judge combination, Alice can still plausibly deny $\mathbb{T}^{\hat{A}}_{\mathcal{C}_1}$, then many of her privacy concerns can be assuaged. Our privacy requirement is stronger than the settings presented in [34, 35] because $\mathcal{J}$ must not be able to distinguish between Alice's transcripts and forgeries even if $\mathcal{J}$ gets Alice's long-term secrets.

3.3.3 Local views

We complete the section by saying that from an honest user's perspective it is unclear a priori whether an honestly behaving user has no malicious intent. Conversely, if a user observes deviation from the protocol the user cannot always distinguish a true malicious player from network instability. (Certain deviations, such as a participant making conflicting statements, can be identified, however.)

3.4 Solution design

The mpOTR protocol follows a straightforward construction. To ensure confidentiality among the participants $\mathcal{P}_1$ of a chatroom $\mathcal{C}_1$ the participants derive a shared encryption key gk_1. Messages sent to the chatroom are encrypted under gk_1 to ensure that only members of $\mathcal{P}_1$ can read them. To provide message authentication, each participant $\hat{A} \in \mathcal{P}_1$ generates an ephemeral signature keypair $(E_{\hat{A},1}, e_{\hat{A},1})$ to be used only in the current session. Each message sent by $\hat{A}$ will be signed under $\hat{A}$'s ephemeral signing key for the current session $e_{\hat{A},1}$. Participants exchange ephemeral public keys for the current session $E_{\hat{X},1}$ ($\hat{X} \in \mathcal{P}_1$) amongst themselves in a deniable fashion. At the end of the session, each participant publishes their ephemeral private key $e_{\hat{X},1}$ ($\hat{X} \in \mathcal{P}_1$) for the current session to allow third

parties to modify and extend the chatroom transcript.

The mpOTR protocol lifecycle consists of three phases: setup, communication, and shutdown. In the setup phase all chatroom participants negotiate any protocol parameters, derive a shared key, generate and exchange ephemeral signing keys, and explicitly authenticate all protocol parameters including the set of chatroom members and the binding between participants and their ephemeral signature keys. During the communication phase, participants can send confidential, authenticated, deniable messages to the chatroom. To end a chatroom session, the protocol enters the shutdown phase. In the shutdown phase, each participant determines if he has reached consensus with each other participant, after which participants publish their ephemeral private keys.

Waiting until the end of a chat session to detect violations of consensus is less than ideal. It would be preferable if, at each point in time during the conversation, honest participants could determine a useful lower bound on consensus over the chat transcript up to that point. However, ensuring incremental consensus in an asynchronous network protocol with an arbitrary number of malicious (Byzantine) participants is a non-trivial undertaking. For simplicity in this work, we presume that such a protocol (see Chapter 4) can be layered underneath mpOTR. To free mpOTR from dependence on any specific consensus-preserving protocol, the shutdown phase guarantees that any consensus violations not repaired by the lower layer will be detected by mpOTR.

3.4.1 Network communication

Our constructions assume the existence of the following network primitives, typically provided by application layer protocols, such as IM or IRC. To free our constructions from undue dependence on the underlying network layer, we limit ourselves to the following

primitives:

- $Broadcast(M)$ — sends message M over the broadcast channel where it can be $Receive()$'ed by all other participants. In the absence of a broadcast medium, like an IRC channel, $Broadcast()$ can be simulated by sending M directly to each other participant in $\mathcal{P}$.

- $Send(\hat{A}, M)$ — sends message M addressed explicitly to $\hat{A}$. The network may send M to $\hat{A}$ directly (point-to-point) or via broadcast (during broadcast, all the honest participants other than $\hat{A}$ ignore M).

- $Receive() \rightarrow (\hat{A}, M)$ — returns any waiting message M received by the party that invokes $Receive()$ along with M's alleged author $\hat{A}$.

- $Receive(\hat{A}) \rightarrow M$ — waits until a message is received from $\hat{A}$ and returns that message (M).

To simplify our protocols, we make the following assumptions. $Broadcast()$ and $Send()$ are non-blocking. If message M from party $\hat{A}$ arrives at $\hat{B}$ before $\hat{B}$ executes a $Receive()$ call, M is buffered at $\hat{B}$ and will be returned upon some subsequent invocation of $Receive()$ by $\hat{B}$. $Receive()$ calls block until a message is available. If the current instance of some party $\hat{A}$ has assigned a value to its session id (sid_i) variable, $Receive()$ will only return messages M that were sent from an instance of some party $\hat{B}$ that has set its session id to the same value (i.e. $Broadcast()$, $Send()$, and $Receive()$ multiplex on sid_i).

Recall that, with all network access, the adversary has control over message delivery and may modify or deliver messages at will. Thus, when $Receive()$ invoked by $\hat{B}$ returns

$(\hat{A}, M)$, $\hat{A}$ may have invoked either $Broadcast(M)$ or $Send(\hat{B}, M)$, or the adversary may have sent M under the identity of $\hat{A}$.

In the following discussion, we abuse notation in that a single value M may be replaced by a tuple $(x_1, x_2, \ldots)$. This indicates that the values $x_1, x_2, \ldots$ have been encoded into a single message using an unambiguous encoding scheme. Upon receiving such a message, if parsing fails, the protocol assigns the distinguished value $\perp$ to each of $x_1, x_2, \ldots$.

3.4.2 Setup phase

Algorithm 3: $Initiate(\mathcal{P}_i)$ — initiate a chatroom $\mathcal{C}_i$ among the participants $\mathcal{P}_i$ in the context of party $\hat{X}$. On successful completion, all participants hold a shared encryption key, ephemeral public signature keys for all other participants, and have authenticated all other participants and protocol parameters.

Input: chat participants $\mathcal{P}_i$
Output: an encryption key gk_i, session id sid_i, ephemeral public signature keys of all other participants $\{E_{\hat{Y},i} \mid \hat{Y} \in \mathcal{P}_i\}$

```
// Initialize variables
```
$sid_i \leftarrow \perp$, $Sent \leftarrow \emptyset$, $Received \leftarrow \emptyset$;
$consensus_{\hat{Y}} \leftarrow$ false for all $\hat{Y} \in \mathcal{P}_i$;
$sid_i \leftarrow SessionID(\mathcal{P}_i)$;
```
// Exchange ephemeral signature keys
```
$(result, R) \xleftarrow{\$} DSKE(sid_i, \mathcal{P}_i)$;
if $result =$ accept **then**
  foreach $(E, \hat{Y}) \in R$ **do** $E_{\hat{Y},i} \leftarrow E$;
else
  abort session initiation;
```
// Agree on shared encryption key
```
$gk_i \xleftarrow{\$} GKA(\mathcal{P}_i, R)$;
if $gk_i = \perp$ **then** abort session initiation;
$Attest()$;

The setup phase is responsible for deriving the shared encryption key gk_i for the chat-

room $\mathcal{C}_i$, performing entity authentication, facilitating exchange of ephemeral signing keys $E_{\hat{X},i}$ ($\hat{X} \in \mathcal{P}_i$), and ensuring forward secrecy and deniability. In the following, we assume that the participants have negotiated the participant set $\mathcal{P}_i$ for the chatroom instance $\mathcal{C}_i$ via an unspecified, unauthenticated means. Each participant in the protocol executes the $Initiate(\mathcal{P}_i)$ algorithm (Algorithm 3) with their view of $\mathcal{P}_i$. The $Initiate()$ procedure will only succeed if every other party in $\mathcal{P}_i$ completes its portion of the protocol correctly and has the same view of $\mathcal{P}_i$.

Algorithm 4: $SessionID(\mathcal{P}_i)$ — invoked in the context of party $\hat{X}$, the algorithm returns a unique (with high probability) chatroom identifier for the set $\mathcal{P}_i$ upon successful completion.

Input: chat participants $\mathcal{P}_i$
Output: session id sid_i
$x_{\hat{X}} \xleftarrow{\$} \{0,1\}^k$;
$Broadcast(x_{\hat{X}})$;
$Outstanding \leftarrow \mathcal{P}_i \setminus \{\hat{X}\}$;
while $Outstanding \neq \emptyset$ **do**
    $(\hat{Y}, x) \leftarrow Receive()$;
    if $\hat{Y} \in Outstanding$ **then**
        $x_{\hat{Y}} \leftarrow x$;
        $Outstanding \leftarrow Outstanding \setminus \{\hat{Y}\}$;
return $H(\mathcal{P}_i, x_{\hat{Y}_1}, x_{\hat{Y}_2}, \ldots)$ for all $\hat{Y}_j \in \mathcal{P}_i$ ordered lexically;

First, the participants calculate a globally unique session id sid_i for the current session (Algorithm 4). Each participant $\hat{X}$ chooses a random value $x_{\hat{X}}$ of suitable length k and broadcasts it to the other participants. Each participant calculates sid_i by hashing the participant set $\mathcal{P}_i$ with the random contributions of all other participants. Under the assumption that $H(\cdot)$ is a collision-resistant hash function, sid_i is globally unique with high probability as long as at least one participant behaves honestly. If the adversary has manipulated the random contributions (x), it will be detected during the $Attest()$ algorithm executed

at the end of $Initiate()$ when sid_i and any other unauthenticated parameters $params_i$ are authenticated.

$\hat{X}$ then enters into a deniable signature key exchange protocol with the other participants of $\mathcal{P}_i$ ($DSKE(sid_i, \mathcal{P}_i)$) to generate an ephemeral signature key pair $(E_{\hat{X},i}, e_{\hat{X},i})$ and to exchange ephemeral public keys with the other parties in $\mathcal{P}_i$. $\hat{X}$ will use $e_{\hat{X},i}$ to sign messages sent to the chatroom $\mathcal{C}_i$. $\hat{X}$ generates a new signing key pair in each session so that there is no transferable proof that he has signed any messages in the chat transcript. However, the other participants must know that $E_{\hat{X},i}$ will be $\hat{X}$'s public signature key for this session.

Next, $Initiate()$ invokes a group key agreement protocol that uses the set of participants $\mathcal{P}_i$ and their ephemeral signature keys to derive a fresh encryption key gk_i shared by all members of $\mathcal{P}_i$. If any stage of the group key agreement fails, $GKA()$ returns $\perp$ and $Initiate()$ aborts.

Algorithm 5: $Attest()$ — authenticate (previously) unauthenticated protocol parameters for the current session in the context of party $\hat{X}$.

Input: session id sid_i, chat participant set $\mathcal{P}_i$, negotiated protocol parameters $params_i$
Output: aborts protocol initiation on failure

```
M ← H(sid_i, params_i);
AuthSend(M);
Outstanding ← P_i \ {X̂};
while Outstanding ≠ ∅ do
    (Ŷ, M_Y) ← AuthReceive();
    if M_Y ≠ M then
        abort the session;
    else
        Outstanding ← Outstanding \ {Ŷ};
```

Finally, all participants execute the $Attest()$ algorithm (Algorithm 5) to ensure that they

agree on all lower-level protocol parameters that they may have negotiated before invoking *Initiate*(). Each participant takes a hash over all of these values and the session identifier, and uses the *AuthSend*() and *AuthReceive*() procedures (see §3.4.3) to transmit the hash value to all the other participants in a confidential, authenticated manner. Each participant then ensures that the value sent by all other participants matches their own. Upon successfully completing *Attest*(), the participants have fully initialized the chat session and can enter the communication phase.

When users wish to join or leave a chatroom, the protocol shuts down the current session and then calls *Initiate*() with the new set of participants to initialize a new chat session. We handle joins and leaves in this manner because we currently determine transcript consensus during the shutdown phase and must derive a new encryption key before a membership change can take place. Client software can shut down and initialize a new session behind the scenes so that users need only decide whether or not they accept the proposed membership change.

Deniable Signature Key Exchange (DSKE)

In our construction, we use a sub-protocol that we call Deniable Signature Key Exchange. Deniable Signature Key Exchange allows the participants in a session to exchange ephemeral signature keys with each other in a deniable fashion. A participant will use his ephemeral signature key to sign messages during one session. Because it is ephemeral (used only in one session), the private key can be published at the end of the session to permit transcript modification. Because the key exchange protocol is deniable, there is no transferable proof that any party has committed to use any given key.

Deniable Signature Key Exchange is an n-party interactive protocol operating over

common inputs: *sid* — a fresh session identifier, and $\mathcal{P}$ — the set of participants for the session identified by *sid*. When the protocol concludes, each participant outputs a termination condition (either accept or reject) and a set R relating the members of $\mathcal{P}$ to public signature keys (e.g. $R = \{(E_{\hat{A}}, \hat{A}), (E_{\hat{B}}, \hat{B}), \ldots\}$).

Two-party signature key exchange The goal of two party signature exchange (Algorithm 6) is to allow Alice and Bob to exchange signing key pairs $(E_{\hat{A}}, e_{\hat{A}})$ and $(E_{\hat{B}}, e_{\hat{B}})$, respectively, such that: (i) Alice is assured that Bob knows $e_{\hat{B}}$ corresponding to $E_{\hat{B}}$; (ii) Alice is assured that Bob, if honest, will not associate $E \neq E_{\hat{A}}$ with Alice; and (iii) Alice is assured that after completing the exchange Bob cannot prove to a third party Charlie (without Alice's consent) that Alice has associated herself with $E_{\hat{A}}$ and knows $e_{\hat{A}}$. The same conditions must hold for Bob with respect to Alice.

Algorithm 6: $AuthUser(sid, \hat{B}, E_{\hat{A}}, e_{\hat{A}})$ — obtain and associate $\hat{B}$ with a signing key pair, and send $\hat{B}$ one's own signing key $E_{\hat{A}}$.

Input: session id *sid*, peer identity $\hat{B}$, signature pair $(E_{\hat{A}}, e_{\hat{A}})$
Output: associate $\hat{B}$ with $E_{\hat{B}}$ or $\perp$
$k, k_m \leftarrow denAKE(\hat{A}, \hat{B})$;
$Send(\hat{B}, SymMacEnc_k^{k_m}(E_{\hat{A}}, sid, \hat{A}, \hat{B}))$;
$(E_{\hat{B}}, sid', \hat{B}', \hat{A}') \leftarrow SymMacDec_k^{k_m}(Receive(\hat{B}))$;
$Send(\hat{B}, SymMacEnc_k^{k_m}(Sign_{e_{\hat{A}}}(E_{\hat{B}}, sid, \hat{A}, \hat{B}))$;
$\sigma \leftarrow SymMacDec_k^{k_m}(Receive(\hat{B}))$;
if $(sid' = sid) \wedge (\hat{A}' = \hat{A}) \wedge (\hat{B}' = \hat{B})$
$\quad \wedge Verify_{E_{\hat{B}}}(\sigma, (E_{\hat{A}}, sid', \hat{B}, \hat{A}))$ **then**
  return $\hat{B}, E_{\hat{B}}$;
else
  return $\perp$;

The signature exchange proceeds as follows: first Alice and Bob run a deniable two-party key agreement protocol $denAKE(\hat{A}, \hat{B})$ to derive a shared secret. Using symmetric

key techniques they exchange signature keys that they intend to use in the subsequent chatroom. Finally, both users sign the ephemeral public key of their peer along with both Alice's and Bob's identities under their ephemeral keys for the current session.

Assume that $denAKE()$ is a secure, deniable authenticated key agreement protocol. Let $SymMacEnc_k^{k_m}()$ (resp. $SymMacDec_k^{k_m}()$) be an algorithm that encrypts (decrypts) and authenticates messages with the symmetric keys k and k_m, and let $Sign()$ be an existentially unforgeable signature scheme. The protocol $denAKE()$ provides keying material only to Bob and Alice. Hence, they are assured about each other's identity. Since Bob signs Alice's ephemeral public signature key she is assured that the signature that Bob generated is not a replay from other sessions and that Bob knows the corresponding ephemeral private key. Bob is assured that $E_{\hat{A}}$ is connected with Alice because he did not generate $E_{\hat{A}}$ and his peer has to know k and k_m to complete the protocol. Since $denAKE()$ is secure, the only party other than Bob that could have computed k and k_m is Alice. Likewise, Alice is assured that an honest Bob will not associate $E \neq E_{\hat{A}}$ with her because Bob will only associate an ephemeral key with Alice if Bob received it through a secure channel that only Bob and Alice share. The only proof that Bob has about communicating with Alice is the $denAKE()$ transcript. Since $denAKE()$ is deniable Alice can argue that any transcript between herself and Bob was created without her contribution; in other words, Bob's view cannot associate Alice to $E_{\hat{A}}$ unless Alice admits the association. Thus Algorithm 6 achieves the three conditions that we described.

We conclude by saying that that $E_{\hat{A}}$ and $E_{\hat{B}}$ are "pseudonyms" that Alice and Bob exchange. As long as the corresponding private keys are not leaked each one of them is assured about the identity behind the pseudonym and messages signed with the keys, but neither can prove to a third party the relation between the pseudonym and a real entity.

Furthermore, any party Mallory can create a fake pseudonym for Alice or Bob.

Multi-party signature key exchange We extend the two-party algorithm to the multi-party setting. In particular, given a set of participants $\mathcal{P}$, every pair of users in $\mathcal{P}$ runs Algorithm 6. For a given identifier *sid*, Alice uses the same key pair $(E_{\hat{A}}, e_{\hat{A}})$.

The next stage is for participants to assure each other of the consistency of the association table that they build. Let $(E_{\hat{A}}, \hat{A}), \ldots, (E_{\hat{X}}, \hat{X})$, be the association table built by Alice, lexicographically ordered on the signing keys. Each user computes a hash of that table, signs the hash with her ephemeral signing key and sends it to the rest of the participants[10]. As a result each participant is assured that the remaining members have the same view about the association table. Note that the exchange does not reveal anything about the table. The set of participants can collaborate to introduce "non-existent" users into the chatroom. In other words, if agreed, a set of users can create a transcript that allegedly involves an absent user Alice. Such a transcript can be indistinguishable from a transcript where Alice did take part.

Deniable AKE By a "secure" key agreement protocol we mean the standard indistinguishable from random key notion introduced by Bellare and Rogaway [9]. However, we are concerned with malicious insiders so protocols that meet models as introduced in [69] are more suitable for our needs, since they allow the adversary to adaptively introduce malicious parties to the system.

In contrast to secure key exchange, "deniable" key exchange has not been as widely studied. On one hand there is a formal definition, presented in [34, Definition 1], that relies on the fact that a receiver's view can be simulated. The authors prove the deniability

[10]This can be incorporated into $Attest()$

of SKEME [53] according to their definition. However, there are some pitfalls related to leaking static secrets and the deniability of SKEME. If the judge $\mathcal{J}$ has access to the static secrets of the alleged participants, $\mathcal{J}$ can distinguish between authentic and simulated transcripts. Therefore, SKEME does not meet our privacy requirement (§3.3.2).

On the other hand, Diffie-Hellman variants like MQV [59] provide plausible deniability as outlined in [14]. The shared key is derived only from public values, so a peer can plausibly argue that he did not take part in the key agreement. Additionally, implicitly authenticated protocols that meet the security definition of [69] appear to meet our privacy notion. This allows any such protocol to be used in settings where the participants may expose their long-lived secrets without sacrificing deniability.

As suggested in [14], one can achieve improved deniability via self-signed certificates that users authenticate. At the extreme it is possible for users *not* to have any static secrets but to authenticate each other via out-of-band means for every session. While such a solution is possible, its usability is questionable. We accept that users cannot convincingly deny their static secrets in order to achieve a less complicated protocol. The users can still deny taking part in any fixed chatroom and the content of messages that they sent.

Group Key Agreement

Assuming that users successfully run the signature exchange protocol, they can proceed to establish group keys. Given *sid* and an association table from *sid* users run a typical key group key agreement protocol to derive a shared secret key *gk* to ensure that they have a means for confidential communication. Note that when the group key agreement is based on the session-specific signature keys, Alice can deny knowing *gk* by arguing that she took no part in the protocol — recall there is no proof of her relation with $E_{\hat{A}}$.

Properties

Alice can plausibly argue that she did not take part in a chat because it is possible to create a protocol transcript that includes users who did not actually take part in the chat. This can happen if all participants collaborate to introduce such non-existent users. In the limit, this allows a single party to create a transcript involving any number of other non-cooperating parties. With an appropriate deniable signature key exchange, the forging party need not even be a member of $\mathcal{P}$. The issue of modifying existing messages in a transcript will be addressed in the shutdown phase.

3.4.3 Communication phase

During the communication phase, chat participants may exchange confidential messages with the assurance of origin authentication — that they have received messages unchanged from their purported authors. Given a chatroom instance $\mathcal{C}_1$ with participant set $\mathcal{P}_1$, we use the group key gk_1, ephemeral public keys of the participants $E_{\hat{X},1}$ ($\hat{X} \in \mathcal{P}_1$) and session id sid_1 for $\mathcal{C}_1$ in a standard Encrypt-then-Sign construction to provide authenticated encryption [8] for messages sent to the chatroom. $AuthSend()$ (Algorithm 7) and $AuthReceive()$ (Algorithm 8) give our construction.

Algorithm 7: $AuthSend(M)$ — broadcast message M authenticated under party $\hat{X}$'s ephemeral signing key to chatroom $\mathcal{C}_i$.

Input: message M, session id sid_i, shared chat encryption key gk_i, ephemeral private signing key $e_{\hat{X},i}$
Output: authenticated encryption of M is broadcast to chat channel
$Sent \leftarrow Sent \cup \{(\hat{X}, M)\}$;
$C \leftarrow Encrypt_{gk_i}(M)$, $\sigma \leftarrow Sign_{e_{\hat{X},i}}(\ (sid_i, C)\ )$;
$Broadcast(\ (sid_i, C, \sigma)\ )$;

Algorithm 8: *AuthReceive*() — attempt to receive an authenticated message from $\mathcal{C}_i$, return the sender and plaintext on success, sender and $\perp$ on failure.

Input: session id sid_i, shared chat encryption key gk_i, ephemeral public signature keys of other participants $\{E_{\hat{Y},i} \mid \hat{Y} \in \mathcal{P}_i\}$
Output: sender identity $\hat{Y}$ and plaintext message M, or $\perp$ on failure

$(\hat{Y}, (sid, C, \sigma)) \leftarrow Receive()$;
if $sid \neq sid_i \vee \neg Verify_{E_{\hat{Y},i}}(\sigma, (sid, C))$ **then**
  return $(\hat{Y}, \perp)$; `// Bad signature or session id`
$M \leftarrow Decrypt_{gk_i}(C)$; `// returns` $\perp$ `on failure`
if $M \neq \perp$ **then**
  $Received \leftarrow Received \cup \{(\hat{Y}, M)\}$;
return $(\hat{Y}, M)$;

When $\hat{A}$ sends a message to the chatroom, she first encrypts the message under the shared key of the chatroom gk_1 to ensure that only legitimate chat participants ($\mathcal{P}_1$) will be able to read it. Then, $\hat{A}$ signs the session id sid_1 and ciphertext using his ephemeral signing key $e_{\hat{A},1}$ and broadcasts the session id, ciphertext, and signature to the network allowing all recipients to verify that $\hat{A}$ has sent the ciphertext to $\mathcal{C}_1$ and that it has been received unmodified.

We assume that $Encrypt()$ and $Decrypt()$ constitute a secure encryption scheme indistinguishable under chosen plaintext attack (IND-CPA) [8], $GKA()$ is a secure group key agreement scheme [12], $DSKE()$ is secure as described in §3.4.2, $Sign()$ and $Verify()$ constitute an existentially unforgeable signature scheme, and session identifiers are globally unique. Under these assumptions, we can transform any confidentiality adversary O (§3.3.2) into a successful adversary against the encryption scheme, the group key agreement that derives the encryption key gk_i, or the deniable signature key exchange scheme that distributes the ephemeral signature keys that are used to authenticate messages sent during the group key agreement. Therefore, under the assumption that the above protocols

are secure, our full scheme is secure against any confidentiality adversary O.

Likewise, the security of $DSKE()$ and the signature scheme imply that the adversary cannot forge messages that are acceptable by $AuthReceive()$. Including the globally unique session id in the message to be signed prevents a message from one session from being replayed in another session. We can also achieve this by deriving a chatroom-specific MAC key from gk_i, which verifies that messages are designated for sid_i. While a consensus adversary $\mathcal{T}$ is unable to successfully forge messages, she can attempt to break consensus by dropping or duplicating messages or by sending different correctly authenticated messages from a corrupted participant to disjoint subsets of honest participants. E.g. $\mathcal{T}$ uses corrupted participant $\hat{C}$ to send M_1 to $\hat{X}$ and M_2 to $\hat{Y}$ where $M_1 \neq M_2$. We address these last three threats during the shutdown phase.

3.4.4 Shutdown phase

When the application determines that there are no outstanding in-flight messages between participants and that the chat session should be ended, it invokes the $Shutdown()$ algorithm (Algorithm 9). $Shutdown()$ is responsible for determining whether all participants have reached a consensus and for publishing the ephemeral signature key generated for the current session. All in-flight messages must have been delivered before invoking shutdown for two reasons: (i) in-flight messages will cause unnecessary failure to reach consensus; and (ii) publication of the ephemeral signature key would allow the adversary to modify any in-flight messages.

To establish consensus, the local party ($\hat{X}$) takes a digest over all the messages authored by $\hat{X}$ during the chat session. $\hat{X}$ then calculates the digests of $\hat{X}$'s transcripts of the messages received from each other party, combines these digests into a single digest, and sends

Algorithm 9: *Shutdown*() — called in the context of party $\hat{X}$ when the application determines that the session should be shut down. Determines if consensus has been reached with other participants and, if so, publishes $\hat{X}$'s ephemeral signing key.

Input: all sent messages *Sent*, all received messages *Received*, participant set $\mathcal{P}_i$, session id sid_i, ephemeral signing key $e_{\hat{X},i}$

Output: $consensus_{\hat{Y}}$ values indicating if consensus has been reached for each party $\hat{Y}$, publishes private ephemeral signing key for current session $e_{\hat{X},i}$

```
// Compute and publish digest of full chat transcript
```

Let $((\hat{X}, M_1^{\hat{X}}), (\hat{X}, M_2^{\hat{X}}), \ldots) = Sent$ in lexical order;
$h_{\hat{X}} \leftarrow H(M_1^{\hat{X}}, M_2^{\hat{X}}, \ldots)$;
foreach $\hat{Y} \in \mathcal{P}_i \setminus \{\hat{X}\}$ **do**
  Let $(M_1^{\hat{Y}}, M_2^{\hat{Y}}, \ldots) = \{M \mid (\hat{Y}, M) \in Received\}$ in lexical order;
  $h_{\hat{Y}} \leftarrow H(M_1^{\hat{Y}}, M_2^{\hat{Y}}, \ldots)$;
Let $(\hat{Y}_1, \hat{Y}_2, \ldots) = \mathcal{P}_i$ in lexical order;
$h \leftarrow H(h_{\hat{Y}_1}, h_{\hat{Y}_2}, \ldots)$;
AuthSend(("shutdown", h));

```
// Determine consensus
```

$Outstanding \leftarrow \mathcal{P}_i \setminus \{\hat{X}\}$;
while $Outstanding \neq \emptyset$ **do**
  $(\hat{Y}, (M, h')) \leftarrow AuthReceive()$;
  if $M =$ *"shutdown"* $\wedge\, \hat{Y} \in Outstanding$ **then**
    $consensus_{\hat{Y}} \leftarrow h = h'$;
    $Outstanding \leftarrow Outstanding \setminus \{\hat{Y}\}$;

```
// Verify that no participants will accept new messages
```

AuthSend("end");
$Outstanding \leftarrow \mathcal{P}_i \setminus \{\hat{X}\}$;
while $Outstanding \neq \emptyset$ **do**
  $(\hat{Y}, M) \leftarrow AuthReceive()$;
  if $M \neq$ *"end"* **then**
    return;
  else
    $Outstanding \leftarrow Outstanding \setminus \{\hat{Y}\}$;

```
// Publish ephemeral signing key associated with X̂'s pseudonym
```

Broadcast($(sid_i, E_{\hat{X},i}, e_{\hat{X},i})$);

it along with the distinguished message "shutdown" to all the other parties. $Shutdown()$ then collects the digests published by all the other participants to allow the local party ($\hat{X}$) to determine if he has reached consensus with each of the other parties on the session transcript.

To ensure that out-of-order message delivery does not affect this digest, the messages are taken in lexical order. Note, however, that should messages include a suitable order fingerprint, then lexical order can coincide with delivery or creation order, hence our ordering is unrestrictive. For example, if each message starts with an author identifier and a sequence number, lexical order will group messages by author in the order that they were created.

Since at the setup phase parties confirmed their views of chat participants and *sid* of the chat, all transcripts already agree on the set of participants and the chat instance. As argued in §3.4.3, the only remaining way for an adversary to break consensus is to force different messages in the transcript. The consensus adversary does not (yet) have the signature keys hence he is still not able to inject new messages or impersonate honest users; his only freedom is the hash function that we assume collision and preimage resistant. Thus chat participants obtain assurances about consistency — they reach pairwise consensus in the sense of §3.3.2.

The consensus approach adopted above is crude as it does not attempt to remedy any consensus errors and it only determines consensus at the very end of the chat session. This means participants will be given no guarantee that any level of consensus ever held between them and other participants that fail or are partitioned away before the chat session ends. We have chosen this approach for its simplicity and clarity. Approaches that ensure consensus incrementally throughout the chat session can be implemented at a network layer

beneath mpOTR. (We outline the desired properties for a lower-level incremental consensus protocol in Section 3.5.) In the meantime, $Shutdown()$ detects consensus violations in order to ensure that, regardless of the specific properties of the underlying consensus protocol, our protocol will detect any violations of reliable delivery at the mpOTR level. Furthermore, the signatures used to authenticate messages are transferable within the chatroom since all members have the correct association between the chatroom-specific signature keys and the entities behind the keys. Therefore the protocol can identify malicious users, since an honest party Alice has transferable proofs to convince any other honest party about the origin of the messages that she received. Thus she can prove that she did not modify or inject messages on behalf of other users. Likewise, she can update her transcript with messages that she failed to receive. Ultimately, honest users can agree on a transcript that is the union of all the messages that have reached at least one honest user.

After exchanging all the values, $Shutdown()$ sends the distinguished message "end" indicating $\hat{X}$ will no longer send any authenticated messages. Once $\hat{X}$ has received the "end" message from each other participant, $\hat{X}$ knows that all participants have determined their consensus values and will no longer accept messages from $\hat{X}$. This allows $\hat{X}$ to publish his ephemeral signing key to permit modifying the chat transcript.

Publishing the ephemeral signing key is a delicate issue. If the key is published too soon, the adversary could use the ephemeral signing key to impersonate the current party to others. Therefore, the protocol only publishes the ephemeral signing key at the end of $Shutdown()$ if it can verify that all other parties have agreed that they have determined their consensus values and will only publish their keys or end the session. The adversary can trivially prevent any party $\hat{X}$ from publishing its signing key by preventing the delivery of even one of the "end" messages. However, this is not a problem. The protocol is

deniable even without publishing the ephemeral signing keys. Therefore, we gladly trade the deniability benefits gained by allowing malleability for ensuring that the adversary will not be able to impersonate $\hat{X}$. However, if parties do publish their ephemeral signing keys then the existing transcripts can be tweaked. This a posteriori publication of signing keys allows for a user Alice who accepts a relation between her chatroom signing key and herself to argue that the messages in the transcript are bogus. Indeed the adversary could inject and/or delete messages on behalf of Alice's ephemeral signing key, since all secret information has been made public.

3.5 Future Work

As outlined above, mpOTR provides a number of important properties: confidentiality, authenticity, non-repudiation among chat session participants, deniability through forgeability of session transcripts, and pairwise indication of consensus at the end of a session. However, a number of practical guarantees needed to build a user-friendly chat application have been delegated other protocol layers.

Network delays and benign failures can cause messages to be received out of order or dropped. For usability, mpOTR should deliver messages in causal order [56] and provide retransmission for lost messages so that no reply is received before messages that preceded it.

When network partitions separate the chatroom into multiple disjoint components, each set of connected participants should be able to continue to communicate amongst themselves and resynchronize with participants in other components once the partition is repaired.

When chatroom participants fail before the end of a session, the remaining chatroom members have no way to determine the degree of consensus that held with the failed participant just before the failure occurred. Whenever a participant receives a message the protocol should indicate the level to which consensus holds between the sender and the receiver instead of deferring the task until the end of the session. Similarly, while we have described a strategy for remedying consensus violations using non-repudiation between chatroom members, the protocol should make it automatic.

Lastly, because malicious chatroom participants or servers may attempt to violate consensus through duplicity or use more than their share of network resources to impede the progress of the chat, the protocol should detect duplicity automatically and carefully prevent resource starvation between honest participants.

Providing these properties in an environment where an arbitrary number of malicious (or Byzantine) insiders can collude to thwart these guarantees is no trivial task. Therefore, the next chapter addresses these issues and develops a Causal Broadcast protocol upon which we can run mpOTR in order to achieve each of these desired properties.

Chapter 4

OldBlue: Causal Broadcast in a Maximally Byzantine Environment

4.1 Introduction

In the last chapter we saw the need for highly-available incremental consensus protocol for mpOTR which remains secure in a Byzantine environment. mpOTR's requirements represent a single instance of the more general distributed systems problem of providing Causal Broadcast. Causal Broadcast is one of the most fundamental primitives for group oriented communication. A Causal Broadcast protocol is a broadcast network protocol which ensures that messages are delivered in an order that preserves the potential causal relationships between messages. If message $\mathtt{m}_1$ could have caused the sending of $\mathtt{m}_2$ (the author of $\mathtt{m}_2$ sent or delivered $\mathtt{m}_1$ before sending $\mathtt{m}_2$), a process will not deliver $\mathtt{m}_2$ before it has delivered $\mathtt{m}_1$. Typically, this property is enforced by including in each message either a vector timestamp or the unique message identifiers of causally preceding messages [82,

85]. This allows the recipient of a message $\mathtt{m}_2$ to precisely determine the set of messages M which causally precede $\mathtt{m}_2$ and to ensure that $\mathtt{m}_2$ will only be delivered once all messages in M have been delivered.

Many Causal Broadcast protocols exist. However, most [68, 3, 82, 40, 11, 84] tolerate benign failures only. Solutions based on tamper-proof hardware which tolerate Byzantine failures have been presented. [96, 97] However, such hardware is still not in wide deployment in commodity PCs. One option for providing Causal Broadcast is to build on protocols providing stronger guarantees, such as Reliable Broadcast [46, 21, 86, 6, 40] and Byzantine Agreement [55, 57, 23]. Though, Byzantine Agreement is solvable for $t = n-1$ Byzantine failures in general, many practical Byzantine Agreement and Reliable Broadcast protocols that guarantee liveness among correct processes, can tolerate at most $t \leq \lfloor \frac{n-1}{3} \rfloor$ Byzantine failures. [21, 86, 6, 62, 23, 67, 28] This limit is undesirable in the maximally Byzantine setting of peer-to-peer (P2P) protocols where an adversary can control an arbitrary fraction of system nodes limited only by his resources. Even benign causes such as P2P network churn and temporary network outages can cause large numbers of processes to be temporarily unreachable. During a outage of $t > \lfloor \frac{n-1}{3} \rfloor$, Reliable Broadcast protocols and Byzantine Agreement protocols must sacrifice availability to maintain consistency.

More general Byzantine Agreement protocols that allow up to $n-1$ Byzantine failures fare no better. If $n-2$ nodes may be Byzantine, a Byzantine Agreement protocol must sacrifice availability if even a single process becomes unreachable. To illustrate, in a Byzantine Agreement protocol, a sender sends a message $\mathtt{m}$ and all correct processes must deliver the some message $\mathtt{m}'$. If the sender was correct, $\mathtt{m}'$ must be equal $\mathtt{m}$. Suppose there are only two correct processes $\hat{p}$ and $\hat{q}$ on opposite sides of a network partition and $\hat{p}$ sends $\mathtt{m}$. For availability to be preserved, $\hat{p}$ must be able to deliver $\mathtt{m}$. However, for Byzantine

Agreement consistency to be preserved, $\hat{q}$ must also deliver m. But, due to the network partition, there is no way for m to be communicated to $\hat{q}$. Therefore, either availability or consistency must be sacrificed.

This is a fundamental limitation for distributed systems which attempt to ensure strong consistency properties like Byzantine Agreement and Consensus. The CAP Theorem [19, 42] states that, during a network partition, a distributed system must choose between maintaining availability or consistency. It cannot preserve both properties simultaneously. By contrast, causal ordering can be determined locally within each connected component allowing connected processes to remain available during network partitions while preserving causal consistency within the component. This indicates the need for a protocol that directly ensures Causal Broadcast, is tolerant of an arbitrary number of Byzantine failures, and ensures availability among connected processes during a network partition. In fact, recent research [61], conducted concurrently with our own, proves that a form of causal consistency closely related to the causal consistency presented in this work is nearly the strongest[1] form of consistency that can be achieved in an always-available system that may undergo network partitions.

The requirement to maintain availability when faced with an arbitrary number of Byzantine processes is not a mere academic novelty. As we saw in Chapter 3, it is the environment in which mpOTR operates. Furthermore, it captures the environment of the increasingly popular peer-to-peer (P2P) networking paradigm. A number of large distributed systems are being constructed from computing and network resources contributed by volunteers throughout the globe. P2P's widely distributed nature and lack of central control make it an attractive architecture for building anonymity-preserving and censorship-

[1] Specifically, [61] proves the impossibility of maintaining a variety of causal consistency properties that are slightly stronger than our own. These results are discussed further in Section 4.7.

resistant networks such as Tor [107] and Freenet [104]. Adversaries can volunteer resources to P2P systems in order to attack from within. Because the adversary's level of influence is only limited by the resources she can contribute, participants in P2P networks find themselves in a maximally Byzantine environment — every other participant may be attempting to thwart the goals that the system seeks to achieve. This has led to efforts to design specialized protocols for secure P2P network variants [113, 76, 73]. However, availability of fundamental distributed systems primitives that are secure in this setting, such as Causal Broadcast, would benefit a variety of systems.

In this work we present OldBlue, a Causal Broadcast protocol which allows recipients of a message `m` to ensure that they have delivered all messages causally preceding `m` (before delivering `m`) by embedding the identifiers of all direct causal predecessors in each message. Message identifiers are computed via a message digest over the sender, message body, identifiers of all direct causal predecessors, and the author's signature. Origin authenticity is guaranteed through the use of digital signatures and, because message identifiers form a Merkle hash tree [70] over all preceding messages, the recipient of `m` is guaranteed that, once they deliver `m`, their view of `m` and all of `m`'s causal predecessors is consistent with the sender's view. In OldBlue, correct processes cooperate in order to opportunistically retransmit lost messages.

Like Reiter and Gong's Piggybacking protocol [85], OldBlue piggybacks message digests to ensure that recipients can precisely determine all causal predecessors of each message. Like Psync, OldBlue uses the directed acyclic graph structure of the causal precedence relation to reduce the amount of information that must be sent in each message. Unlike either protocol, OldBlue enforces a formal fairness property and goes to great lengths to prevent starvation between correct connected processes in the presence of

an arbitrary number of Byzantine processes. In contrast to protocols which give stronger delivery and ordering guarantees such as Byzantine Agreement, OldBlue's direct support of causal precedence and fairness ensures availability among connected correct processes during network partitions.

First, we describe our threat model (Section 4.2) before formally defining OldBlue's consistency and availability properties (Section 4.3). We then present the basic protocol (Section 4.4) providing formal proofs that each property has been achieved (Section 4.4.6). Section 4.5 presents preliminary simulation results indicating the network performance of OldBlue. We discuss practical implementation considerations in Section 4.6. Section 4.7 reviews prominent related work in the field. Finally, areas for future work are discussed in Section 4.8.

4.2 Threat Model

OldBlue operates in the following setting. A protocol session takes place between n processes. The processes are connected by an unreliable, asynchronous, multicast medium which may be simulated via point-to-point messages or broadcast.

A Byzantine adversary has complete control over the network. The adversary may modify, insert, delete, duplicate, and reorder messages as she pleases. This ensures that OldBlue's security properties are not dependent on the properties of the underlying network.

The adversary may also *corrupt* an arbitrary number (t) of processes, learning their internal state including both their long-lived and session secrets. The adversary can cause corrupt processes to deviate from the protocol arbitrarily. In contrast to corrupt processes,

correct processes neither divulge their internal state nor deviate from the protocol. Processes with incorrect implementations or hardware failure are considered corrupt and their behavior to be the manner in which the adversary has caused them to deviate from the protocol. Because the adversary has full control of the network, she may prevent a *correct* process $\hat{p}$ from receiving any message(s) sent by other processes, effectively disconnecting them from the network.

In the maximally Byzantine setting of many P2P networks, an adversary may join as many nodes to the system as her resources permit. While we place no limits on the number of corrupt processes (t), some of OldBlue's properties are trivially true unless there are at least two correct processes. Therefore, without loss of generality, in the following discussion we will assume that $t \leq n-2$.

4.3 Protocol Properties

OldBlue provides the following interface to other system layers:

CB.open$(processes, pid, gk, sid)$ Begin a causal broadcast session among *processes* (initialize state, etc.). *pid* is the identifier of the local process. *gk* is the session group encryption key. *sid* is a session identifier unique over all sessions.

id CB.broadcast(`msg`) Invoked by the application layer to broadcast message `msg` to all processes. Returns a unique *id* which distinguishes the current invocation at the local process from all other invocations by all other processes. This permits associating each invocation of *CB.deliver*() below with a specific invocation of *CB.broadcast*(). Because an application may send messages in response to inputs provided by the

adversary, the adversary is free to invoke the $CB.broadcast(\mathtt{m})$ method of correct processes for arbitrary $\mathtt{m}$.

$CB.deliver(\hat{p}, id, \mathtt{msg})$ Callback to application layer signaling delivery of message $\mathtt{msg}$ with id *id* authored by $\hat{p}$. Messages are delivered in causal order.

$CB.close()$ End a session opened by $CB.open()$.

The formal properties of OldBlue are defined in terms of the operation of $CB.broadcast()$ and $CB.deliver()$. OldBlue ensures that messages are delivered in causal order. Informally, if receipt of a message $\mathtt{m}$ could have caused the sending of message $\mathtt{m}'$, we say that $\mathtt{m}$ causally precedes $\mathtt{m}'$. Formally:

Definition 1 (Causal Precedence)**.** *An event a* causally precedes *an event b (written $a \rightarrow b$) if and only if one of the following conditions hold:*

1. *a correct process $\hat{p}$ executes $CB.broadcast(\mathtt{m})$ yielding i (event a) and b is the corresponding message delivery $CB.deliver(\hat{p}, i, \mathtt{m})$ (i.e. broadcast of a message precedes its delivery)*

2. *a correct process executes $CB.broadcast(\mathtt{m})$ (event a) before executing $CB.broadcast(\mathtt{m}')$ (event b) (i.e. temporal progression at a correct process)*

3. *a correct process executes $CB.deliver(\hat{p}, i, \mathtt{m})$ (event a) then $CB.broadcast(\mathtt{m}')$ (event b) (i.e. delivery of $\mathtt{m}$ precedes broadcast of $\mathtt{m}'$)*

4. *there exists some event e such that $a \rightarrow e$ and $e \rightarrow b$ (i.e. transitive closure)*

The causal precedence relation $\cdot \rightarrow \cdot$ defines a partial ordering over events. If $a \rightarrow b$, then we can say that *a* happened before *b*. Any two events *c* and *d* not related by the causal

precedence relation are said to be *concurrent* (i.e. c and d are concurrent if $c \not\rightarrow d$ and $d \not\rightarrow c$). No statements can be made about the execution order of concurrent events.

In the context of Causal Broadcast protocols, all events of interest correspond to broadcast or delivery of messages. Therefore, we will abuse notation and speak of a message $\mathtt{m}$ causally preceding another message $\mathtt{m}'$ ($\mathtt{m} \rightarrow \mathtt{m}'$) where event a is the broadcast or delivery of $\mathtt{m}$, event b is the broadcast of $\mathtt{m}'$ and $a \rightarrow b$ according to Definition 1.

OldBlue captures *potential* causality between messages. It may be the case that a message $\mathtt{m}$ was delivered before a message $\mathtt{m}'$ was broadcast, but that the broadcast of $\mathtt{m}'$ was in no way influenced by $\mathtt{m}$. Without application-specific knowledge, it is not possible to identify this scenario. Therefore, if a message $\mathtt{m}$ *could* have influenced the sending of another message $\mathtt{m}'$, OldBlue conservatively determines that $\mathtt{m} \rightarrow \mathtt{m}'$.

Causal precedence is defined by actions performed by correct processes because corrupt processes can behave arbitrarily. Note the distinction between *actual* causal precedence – as defined above – and *apparent* causal precedence. Causal precedence must be represented in protocol messages. It is possible that corrupt processes may author messages which *appear* to have an arbitrary set of (possibly non-existent) causal predecessors. Therefore, message $\mathtt{m}$ with id i authored by corrupt process $\hat{p}$ enters the causal precedence relation only when a correct process executes $CB.deliver(\hat{p}, i, \mathtt{m})$. We address this issue further in Sect(s). 4.4.3 and 4.4.6.

OldBlue ensures the following guarantees on correct process behavior[2]:

Definition 2 (Validity)**.** *If a correct process $\hat{p}$ executes $CB.broadcast(\mathtt{m})$ yielding i for a message $\mathtt{m}$, then $\hat{p}$ eventually executes $CB.deliver(\hat{p}, i, \mathtt{m})$. If the adversary delivers (without*

[2]OldBlue's properties are expressed in the positive (guarantees on correct process behavior) as opposed to the negative (limitations on the adversary's behavior).

modification) all messages associated with CB.broadcast(m) *to a correct process* $\hat{q}$*,* $\hat{q}$ *will execute CB.deliver*$(\hat{p}, i, \mathtt{m})$.

Validity provides three guarantees. It rules out trivial protocols that deliver no messages. It ensures self-delivery of messages. And, it ensures liveness of connected correct processes – if the adversary faithfully delivers all protocol messages associated with an invocation of *CB.broadcast*(m) to a correct process, that process must deliver m. If the adversary does not faithfully deliver all such associated messages, to a correct process $\hat{q}$, $\hat{q}$ is not required to deliver m. (Indeed, in some cases to do so would violate Causal Consistency.) In this paper, we use the following working definition of associated messages:

Definition 3 (Associated Message)**.** *A message is associated with CB.broadcast*(m) *if it is the initial broadcast of* m*, a request to retransmit a lost message causally preceding* m *sent by a process trying to deliver* m*, or a message causally preceding* m *retransmitted in response to such a retransmission request.*

Validity determines when a message must be delivered but it does not place any restrictions on delivery order. In OldBlue, message delivery order is governed by Causal Consistency:

Definition 4 (Causal Consistency)**.** *No correct process* $\hat{p}$ *will execute CB.deliver*$(\hat{q}, i, \mathtt{m})$ *until it has delivered all messages* $\mathtt{m}'$ *which causally precede* m*. Specifically, if* $\hat{q}$ *(the author of* m*) is a correct process, before executing CB.deliver*$(\hat{q}, i, \mathtt{m})$*,* $\hat{p}$ *will have executed:*

1. *CB.deliver*$(\hat{x}, i', \mathtt{m}')$ *for each such call that occurred at* $\hat{q}$ *before* $\hat{q}$ *executed CB.broadcast*(m) *yielding i*

2. *CB.deliver*$(\hat{q}, i', \mathtt{m}')$ *for each call CB.broadcast*$(\mathtt{m}')$ *yielding* i' *made by* $\hat{q}$ *before* $\hat{q}$ *executed CB.broadcast*(m) *yielding i*

Causal Consistency ensures that all messages are delivered in causal order. Causal Consistency further ensures that, whenever any two correct processes execute $CB.deliver(\hat{q}, i, \mathtt{m})$, these two processes agree on the message contents, authorship, and causal ordering of $\mathtt{m}$ and all $\mathtt{m}' \rightarrow \mathtt{m}$. A corrupt author $\hat{q}$ does not weaken the consistency guarantees ensured by Definition 4. If $\hat{q}$, the author of $\mathtt{m}$, is corrupt, the causal precedence of $\mathtt{m}$ will not be defined until $\mathtt{m}$ has been delivered by a correct process. However, $\mathtt{m}$ will have a set of *apparent* causal predecessors. Because $\hat{q}$ is corrupt, Validity does not require $\hat{p}$ to deliver $\mathtt{m}$. Any correct process $\hat{p}$ that does deliver $\mathtt{m}$ does not know whether $\mathtt{m}$'s author is correct. Therefore, $\hat{p}$ must deliver $\mathtt{m}$ subject to the ordering imposed by $\mathtt{m}$'s apparent causal predecessors to ensure that $\hat{p}$'s operation is correct irrespective of the correctness of $\hat{q}$. To illustrate, let $\hat{p}$ execute $CB.deliver(\hat{q}, i, \mathtt{m})$. Let the next message that $\hat{p}$ sends be $\mathtt{m}'$. $\mathtt{m}$ is now a causal predecessor of $\mathtt{m}'$. Therefore, Validity requires any correct process $\hat{r}$ which receives the messages associated with $CB.broadcast(\mathtt{m}')$ yielding i' to execute $CB.deliver(\hat{p}, i', \mathtt{m}')$. Because $\hat{p}$ is correct, Causal Consistency also requires $\hat{r}$ to $CB.deliver(\hat{q}, i, \mathtt{m})$ and all apparent causal predecessors delivered by $\hat{p}$. Thus Causal Consistency guarantees that correct processes $\hat{p}$ and $\hat{r}$ will agree on $\mathtt{m}'$ and all causal predecessors (including $\mathtt{m}$) after the delivery of $\mathtt{m}'$ regardless of the correctness of $\hat{q}$, the author of $\mathtt{m}$.

In this way Causal Consistency extends the liveness property ensured by Validity. Validity ensures that processes will be able to deliver a faithfully transmitted message authored by a correct process. Causal Consistency ensures that all undelivered causally preceding messages will be delivered as well.

Definition 5 (Authenticity)**.** *In a session identified by sid, every correct process executes* $CB.deliver(\hat{p}, i, \mathtt{m})$ *at most once for each value of i (and any values of* $\hat{p}$ *and* $\mathtt{m}$*) and, if* $\hat{p}$ *is*

correct, then $\hat{p}$ previously executed a call to CB.broadcast$(\mathtt{m})$ *yielding i within session sid.*

Authenticity ensures that any attempt to impersonate a correct process or replay messages will not succeed. Authenticity also ensures that, for every delivered message, if the purported author is correct, the message was transmitted without modification from that author. Because corrupt processes can deviate arbitrarily from the protocol, Authenticity does not place any constraints on their internal behavior.

Authenticity can also facilitate the detection of certain attacks. Suppose the adversary partitions the session into two or more disjoint subsets $\mathcal{P}_1, \mathcal{P}_2$ and causes corrupt $\hat{d}$ to send differing messages $\mathtt{m}_1 \neq \mathtt{m}_2$ to the respective subsets. To prevent each message from requiring the delivery of the other, $\hat{d}$ must author $\mathtt{m}_1$ and $\mathtt{m}_2$ so that neither is a causal predecessor of the other (i.e. $\mathtt{m}_1$ and $\mathtt{m}_2$ are concurrent). After processes in $\mathcal{P}_1$ deliver $\mathtt{m}_1$, the adversary must maintain the partition indefinitely. If the adversary allows transmission of any subsequent message $\mathtt{m}_3$ from $\hat{p} \in \mathcal{P}_1$ to any $\hat{q} \in \mathcal{P}_2$, $\mathtt{m}_3$ will include $\mathtt{m}_1$ as a causal predecessor. This causes $\hat{q}$ to deliver both $\mathtt{m}_1$ and $\mathtt{m}_2$ — two concurrent messages from $\hat{d}$. Under the assumption that $\hat{d}$ is correct, Authenticity requires that $\hat{d}$ invoked $CB.broadcast(\mathtt{m}_1)$ concurrently with $CB.broadcast(\mathtt{m}_2)$. Causal Precedence ensures us no correct process will do so, leading to a contradiction. Therefore $\hat{d}$ must be corrupt and $\mathtt{m}_1, \mathtt{m}_2$ serves as a proof.

Corrupt processes may attempt to monopolize the resources of correct processes to prevent them from making progress. OldBlue prevents this by employing a fair scheduling criterion. Actions that processes perform are represented as *requests* (see Section 4.4.2). Each process explicitly or implicitly issues requests for information required to make progress. (Any process that does not have an outstanding request to retransmit a message is considered to have implicitly requested new messages.) Correct processes fulfill requests according to a fair schedule to prevent starvation of correct processes.

We associate with every process $\hat{p}$ a FIFO queue of requests $R_{\hat{p}}$. We define a function *bool* $eligible(\hat{p}, r)$ that captures external constraints governing whether or not process $\hat{p}$ is able to accept the fulfillment of request r at the time of invocation. E.g. A congestion control mechanism might dictate that process $\hat{p}$ is congested, thus we should not attempt to send a message requested by $\hat{p}$ right now because it is likely to be dropped. We assume that, for each process $\hat{p}$ and request r, $eligible(\hat{p}, r)$ depends only on information controlled by $\hat{p}$ and network conditions. Thus, if $\hat{p}$ is correct, the adversary cannot cause $\hat{p}$'s requests to be ineligible by means other than exercising her ability to drop and delay messages on the network. Furthermore, $eligible(\hat{p}, r)$ will eventually become true for all processes $\hat{p}$ and requests r.

Definition 6 (Fairness)**.** *A scheduling algorithm outputs a sequence of requests from processes* $(\hat{p}_i, r_i), (\hat{p}_j, r_j), \ldots$ *A* fair *scheduling algorithm ensures that each process with eligible requests will have one of their requests scheduled at least once in every n requests. That is to say, for each process* $\hat{p}$ *where* $\hat{p}$ *has an eligible request at the time the* i^{th} *request* (r_i) *is scheduled, there is a request* r_j *in the schedule with* $\hat{p}_j = \hat{p}$, $r_j \in R_{\hat{p}}$, *and* $i \leq j < i+n$.

Fairness ensures that corrupt processes cannot cause a correct process to starve other correct processes. Fairness only governs the outgoing messages sent by each correct process because processes cannot control which messages they receive, nor can they determine the author of a message until after signatures are verified. Correct processes process incoming messages in a first-come first-served order. We adopt a message-based fairness criterion for simplicity, however other definitions may be suitable. Differences in computation time and message transmission time due to message length can be upper bounded by suitable constants allowing message-based fairness to asymptotically approximate definitions capturing bandwidth or computation.

4.4 The OldBlue Protocol

OldBlue is positioned above a network transport protocol layer that supports unreliable asynchronous unordered end-to-end message transmission via multicast (possibly simulated using multiple unicasts). $send(recipients, msg)$ will multicast msg to $recipients$. The callback $recv(sender, msg)$ is invoked when a message, allegedly from $sender$, is received from the network.

We associate with each process $\hat{p}$ a public-private signing key pair ($pk(\hat{p})$, $sk(\hat{p})$). We assume that the correspondence between public signing keys and process identities is known to all processes. We denote signing message $\mathtt{m}$ under private key $sk(\hat{p})$ by $\sigma = S(sk(\hat{p}), \mathtt{m})$. The corresponding verification of the signature σ under public key $pk(\hat{p})$ is denoted by $V(pk(\hat{p}), \mathtt{m}, \sigma)$. We denote encryption of message $\mathtt{m}$ under key gk by $E(gk, \mathtt{m})$, $D(gk, \mathtt{c})$ is the corresponding decryption of ciphertext $\mathtt{c}$. $E()$ is a IND-CPA secure encryption scheme and $D()$ is the corresponding stateless, deterministic decryption function.

We assume the existence of unambiguous serialization and deserialization functions $encode()$ and $decode()$ with $m = decode(encode(m))$ which, respectively, are able to encode a data type for transmission over the network and decode a corresponding language-level data type from the network. We assume that $H()$ is cryptographic hash function that is collision-resistant (it is computationally infeasible to find two inputs $\mathtt{m}_1 \neq \mathtt{m}_2$. such that $H(\mathtt{m}_1) = H(\mathtt{m}_2)$) and one-way (given a value $\mathtt{h}$ it is computationally infeasible to find a value $\mathtt{m}$ such that $H(\mathtt{m}) = h$).

In the following protocol descriptions, we use a Python-like pseudocode to describe OldBlue's algorithms. We deviate from Python syntax in a number of ways to ease reading for those that are less familiar with Python. For example, we use + and - as set union

and difference operators respectively, `this` to represent the current object instead of `self`, `null` in lieu of `None`, and specify return types for methods.

4.4.1 Initialization

The session begins with a call to $CB.open(Processes, Pid, gk, Sid)$ which will initialize the following process state for each OldBlue session.

- *sid*: a nonce used to uniquely identify the current session.
- *Pid*: process identifier of the local process.
- *Processes*: set of process identifiers of all members of session *sid*.
- *Requests*: priority queue of (*pid*, set of outstanding requests for *pid*) tuples maintained in least-recently-used order (initially $Processes \times \emptyset$).
- *Delivered* and $Undelivered$: maps from message id to delivered and received-but-not-yet-delivered message objects, respectively (both are initially $\emptyset$). If a message is received before one of its causal predecessors, it will remain in $Undelivered$ until all causal predecessors have been delivered.
- $Frontier$: set of message id's of leaves of the causal graph (initially $\emptyset$). If $Frontier$ contains the id of message `m`, the local process has not delivered any messages causally newer than `m`.
- *Wire*: map from message id to transport-level representation of messages to allow collaborative retransmission (initially $\emptyset$).
- *gk*: fresh encryption key for session *sid* shared by the members of *Processes*.

The application is responsible for determining session membership and negotiating a shared encryption key and a fresh session id for the session (e.g. by executing an appropriate Authenticated Group Key Agreement protocol providing a fresh, mutually-authenticated, shared key [12]). We assume that all correct processes initialize their session by calling *CB.open*() with the same values of *Processes*, *gk*, and *sid* and differing values of *Pid*. *CB.open*() is defined in Figure 4.1.

```
CB.open(processes, pid, key, sid):
  Processes = processes
  Pid = pid
  Requests = (Processes, set())
  Delivered = set()
  Undelivered = set()
  Frontier = set()
  Wire = set()
  gk = key
  Sid = sid
```

Figure 4.1: Definition of *CB.open*().

4.4.2 Request Fulfillment

At the highest level, OldBlue works as a request fulfillment engine, fulfilling requests in a fair order as determined by the scheduler. OldBlue's public methods add requests to *Requests* for later fulfillment. OldBlue defines three kinds of `Request` objects: `Outgoing`, `LostMsg`, and `Retransmit`. Each `Request` type has a single message id member (`mid`) and defines a `fulfill()` method which takes the actions necessary to fulfill the request. `Outgoing` requests cause messages *CB.broadcast*() by the local process to be sent to the network. `LostMsg` requests ask for retransmission of missing causal predecessors for messages in *Undelivered*. `Retransmit` requests cause retransmission of a message requested

via a `LostMsg` received from other processes.

Scheduling constraints are captured by the *schedule*() method, which dequeues an eligible request from *Requests* subject to OldBlue's Fairness constraints. *schedule*() will block the main thread of execution until a request becomes eligible as determined by the *eligible*() function defined in Section 4.3. This leads to straightforward top-level operation depicted in Figure 4.2.

```
CB.main():
  repeat forever:
    (owner, request) = schedule(Requests)
    request.fulfill(owner)
```

Figure 4.2: OldBlue main loop.

In the following discussion we use the simple FIFO scheduling algorithm of Figure 4.3 to ensure fairness. Let `Request` *choose_request*(*process*, *Requests*) capture any criteria used to discriminate between requests from a given process such as: preferring retransmissions to new messages, preferring retransmissions requested by the most processes, etc.

Other, more complicated algorithms could optimize parameters other than the maximum sequence between requests fulfilled such as the expected sequence length between requests fulfilled, etc.

4.4.3 Message Transmission

Transmission of new messages (Figure 4.4) in OldBlue is straightforward. Each `Message` encodes the author, the message ids of all immediately causally preceding messages, and the message payload provided by the application. The message id of each message uniquely identifies the message within the session by taking a digest over the signed transport-level

```
(process, request) schedule(Requests):
  # Where Requests is a list of process id and request queues:
  # (p, Rp), (q, Rq), (s, Rs), ...
  # Examine in least-recently-serviced order
  i = 1
  while i <= n:
    (p, Rp) = Requests[i]
    Ep = [ r for r in Rp if eligible(p, r) ]
    if Ep != []:
      # p has eligible requests
      r = choose_request(p, Ep)
      # move p to the end of the list
      del Requests[i]
      Requests[n] = (p, Rp - set(r))
      return (p, r)
    i += 1
  return null
```

Figure 4.3: FIFO request scheduling algorithm employed by OldBlue.

payload. This ensures that two messages will have equal message ids if and only if the authors, message ids of all causal predecessors, message payloads, and signatures over the preceding fields are identical. As noted by Reiter and Gong [85], including an author's signature under the message id, prevents corrupt processes from introducing false causal dependencies on messages authored by correct processes (to do so would require finding a collision in the hash function or forging the signature of a correct process). Immediately self-delivering the message ensures that the self-delivery requirement of Validity is satisfied. *CB.broadcast*() adds an `Outgoing` request for the message id to the request queue of the local process.

Fulfillment of the `Outgoing` request actually broadcasts the message to the other processes in the session. *authEncrypt*() uses a standard Encrypt-then-Sign construction [8] providing IND-CCA security to prevent forgery of messages by corrupt processes and to

```
class Message:
  author     # ProcId
  parentids  # Set of MsgId
  payload    # String
  id         # MsgId

id CB.broadcast(msg):
  m = Message(author=Pid,
              payload=msg)
  # Immediate causal preds
  m.parentids = Frontier
  wire_msg = authEncrypt(m)
  m.id = H(wire_msg)
  Wire[m.id] = wire_msg
  # Self-delivery
  deliverMessage(m)
  addRequest(Pid,
      Outgoing(mid=m.id))
  return m.id

deliverMessage(m):
  Delivered[m.id] = m
  Frontier -= m.parentids
  # m is now a leaf of
  # the causal graph
  Frontier += set(m.id)
  CB.deliver(m.author, m.id,
             m.payload)

Outgoing.fulfill(owner):
  send(Processes,
       Wire[this.mid])

String authEncrypt(sender,
                   payload):
  ctxt = E(gk, encode(payload))
  m = encode(Sid, ctxt)
  sig = S(sk(Pid), m)
  return encode(Pid, ctxt, sig)

# Add req to p's request queue
addRequest(p, req):
  # Find i s.t.
  #      Requests[i] == (p, R)
  Rnew = R + set(req)
  Requests[i] = (p, Rnew)

# Remove any requests of p
# equal to req
removeRequest(p, req):
  # Find i s.t.
  #      Requests[i] == (p, R)
  Rnew = R - set(req)
  Requests[i] = (p, Rnew)
```

Figure 4.4: Implementation of CB.broadcast()

preserve confidentiality and authenticity despite external adversaries. Replay of messages across sessions is prevented by including the session id under the signature, ensuring that the signature will only be valid for the current session. To facilitate retransmission of the message, the transport-level payload is saved in *Wire*. The transport-level message also includes the author's *Pid* to allow the message to be retransmitted by processes other than the author.

deliverMessage(m) performs the actual delivery of deliverable message m. m is added to *Delivered*. m replaces any of its immediate causal predecessors in *Frontier* and, finally, m is provided to the application via *CB.deliver*().

4.4.4 Message Receipt

When a transport-level message is received (Figure 4.5), *authDecrypt*() verifies that its signature is valid for the current session and decrypts the message. If any error occurs in deserialization, signature verification, or decryption, *authDecrypt*() returns null. Otherwise, it returns a re-constituted `Message` or `LostMsg` object (Section 4.4.5 describes the handling of `LostMsg`).

`Message.receive()` handles the actual processing of the message. It first performs sanity checks to reject messages with incorrect author fields, senders outside of the process group, or previously received messages. It then removes any requests to retransmit the current message that the local process may have initiated. If the new message had any immediate causal predecessors that have not been received, a request for retransmission is scheduled by adding a `LostMsg` request to the request queue for the local process. The arrival of a message may cause messages in *Undelivered* to become deliverable. Therefore, the new message is added to the set of *Undelivered* messages and all deliverable messages are delivered.

4.4.5 Message Loss and Retransmission

When a `Message` arrives before one of its causal predecessors, correct processes assume that the predecessor has been lost and they will add a `LostMsg` request to their request

```
Upon recv(sender, msg):
  au_msg = authDecrypt(msg)
  if au_msg != null:
    (author, msgOrReq) = au_msg
    msgOrReq.receive(author,
                     msg)

Object authDecrypt(payload):
  parts = decode(payload)
  (author, ctxt, sig) = parts
  sid_ctxt = encode(Sid, ctxt)
  pk = pk(author)
  if !V(pk, sid_ctxt, sig):
    return null
  obj = decode(D(gk, ctxt))
  if obj:
    return (author, obj)
  else:
    return null

bool parentsDelivered(msg):
  for parent in msg.parentids:
    if !Delivered[parent]:
      return false
  return true

bool shouldDiscard(msg, author):
  # Reject pathologies and dupes
  return (author != msg.author
    or author not in Processes
    or Undelivered[msg.id]
    or Delivered[msg.id])
```

```
Message.receive(author,
                payload):
  this.id = H(payload)
  if shouldDiscard(this,
                   author):
    return

  # It's definitely not lost
  removeRequest(Pid,
      LostMsg(mid=this.id))

  # Request missing parents
  for parent in this.parentids:
    if !Undelivered[parent]
       and !Delivered[parent]:
      addRequest(Pid,
          LostMsg(mid=parent))

  # Add to undelivered set
  Undelivered[this.id] = this
  Wire[this.id] = payload
  do:
    anyDelivered = false
    for msg in Undelivered:
      if parentsDelivered(msg):
        # msg is deliverable
        del Undelivered[msg.id]
        deliverMessage(msg)
        anyDelivered = true
  while anyDelivered == true
```

Figure 4.5: Implementation of message receipt

queue (Figure 4.6). Fulfillment of the `LostMsg` request will cause the `LostMsg` request to be forwarded to other processes, serving as a negative acknowledgment and requesting that they retransmit the lost message. The "missing" causal predecessor may not actually be lost, it may merely arrive after a causally newer message making the `LostMsg` request

superfluous. We assume that *eligible*() delays `LostMsg` requests in an attempt to minimize such superfluous requests.

```
LostMsg.fulfill(owner):
  payload = authEncrypt(owner,
                        this)
  send(Processes, payload)
  addRequest(Pid, this)

Retransmit.fulfill(owner):
  if Wire[this.mid] != null:
    send(set(owner),
         Wire[this.mid])
```

```
LostMsg.receive(author,
                payload):
  if author in Processes:
    id = this.mid
    req = Retransmit(mid=id)
    addRequest(author, req)

Retransmit.receive(author,
                   payload):
  # internal use only, ignore
```

Figure 4.6: Implementation of negative acknowledgment and retransmission

Because the `LostMsg` request itself may be lost, correct processes re-enqueue the `LostMsg` request, ensuring that the process will continue to request lost messages until they are received. `Message.receive()` will remove the associated `LostMsg` request from the local process's request queue, when the lost message is received.

When a `LostMsg` request is received from another process, a corresponding `Retransmit` request is added to their request queue. When a correct process fulfills a `Retransmit` request, it will forward a copy of the lost message to the requesting process.

4.4.6 Satisfaction of Formal Properties

In this section, we provide proofs that OldBlue satisfies the formal properties defined in Section 4.3.

Theorem 1. *schedule*() *is fair in the sense of Definition 6.*

Proof. The proof is by contradiction, suppose that $\hat{p}$ had an eligible request r at the time

that some request (r_i) was scheduled. Assume there is a sequence of requests $(\hat{p}_i, r_i), \ldots, (\hat{p}_{i+n-1}, r_{i+n-1})$ where no $\hat{p}_j = \hat{p}$ for $i \leq j < i+n$. By assumption, $\hat{p}$ had an eligible request during these n invocations of $schedule()$. Therefore, at each invocation, $schedule()$ chose an eligible request from the first process with such a request and moved that process to the end of the list. Thus, there must have been n processes with eligible requests before $\hat{p}$ in *Requests*. This is a contradiction because there are only n processes, and thus at most $n-1$ such processes can be ahead of $\hat{p}$ in *Requests*. □

Theorem 2. *OldBlue satisfies Definition 4: Causal Consistency. I.e. no correct process $\hat{p}$ will execute CB.deliver$(\hat{q}, i, \mathtt{m})$, until it has delivered all messages $\mathtt{m}'$ which causally precede* $\mathtt{m}$.

Proof. The proof is by contradiction. Suppose that correct $\hat{p}$ delivers $\mathtt{m}$ before delivering a causally preceding message $\mathtt{m}_1$ with id i_1. Choose $\mathtt{m}_1$ such that $\hat{p}$ has delivered a message $\mathtt{m}_2$ where $\mathtt{m}_1$ is the direct causal predecessor of $\mathtt{m}_2$. (Such an $\mathtt{m}_1$ and $\mathtt{m}_2$ always exist with $\mathtt{m}_2$ possibly equal to $\mathtt{m}$.) Each message contains the ids of immediate causal predecessors, therefore $i_1 \in \mathtt{m}_2.parentids$. Because $\hat{p}$ is correct, $\hat{p}$ delivered some message $\mathtt{m}'_1 \neq \mathtt{m}_1$ with id i_1. Because, $authDecrypt()$ is stateless and deterministic, the transport-level messages decrypting to $\mathtt{m}'_1$ and $\mathtt{m}_1$ must differ. Therefore, the transport-level messages corresponding to $\mathtt{m}_1$ and $\mathtt{m}'_1$ constitute a collision in $H()$ contradicting its collision-resistance assumption and establishing the proof. □

Theorem 3. *OldBlue meets Definition 2: Validity. I.e. correct process $\hat{p}$ self-delivers its own messages and will deliver any message* $\mathtt{m}$ *from correct process $\hat{q}$ if the adversary delivers all associated messages (see Definition 3).*

Proof. Self-delivery is immediate by the implementation of $CB.broadcast()$.

If $\hat{p}$ does not immediately *CB.deliver*() m, it is because some $\mathtt{m}' \rightarrow \mathtt{m}$ has not been received. $\hat{p}$ will issue a `LostMsg` request for $\mathtt{m}'$. Some correct process $\hat{q}$ holding $\mathtt{m}'$ will receive the request and eventually retransmit $\mathtt{m}'$ to $\hat{p}$ because, by assumption, the adversary is delivering all messages associated with m. This process continues until all causal predecessors of m have been received by $\hat{p}$, at which time $\hat{p}$ will *CB.deliver*() m. □

Theorem 4. *OldBlue meets Definition 5: Authenticity. I.e. in a session with id sid a correct process $\hat{q}$ executes CB.deliver($\hat{p}$, i, m) at most once for each value of i and, if $\hat{p}$ is correct, then $\hat{p}$ previously executed CB.broadcast(m) yielding i.*

Proof. `Message.receive()` ensures that at most one message with id *i* will be delivered. If a message arrives a message id equal to that of a previously received message in Delivered or Undelivered, it is ignored.

Suppose correct $\hat{q}$ executes *CB.deliver*($\hat{p}$, *i*, m) but $\hat{p}$ never executed *CB.broadcast*(m) yielding *i* in session *sid*. Either (1) $\hat{p}$ never executed *CB.broadcast*(m) (at all) in session *sid* or (2) $\hat{p}$ executed *CB.broadcast*(m) yielding $j \neq i$. In the latter case, the adversary either (2a) changed the transport-level message, or (2b) she didn't.

In cases (1) and (2a) the transport-level message that resulted in *CB.deliver*($\hat{p}$, *i*, m) is a successful forgery against the INT-CTXT security [8] of the Authenticated Encryption scheme. In case (2b), because the transport-level message received by $\hat{p}$ is identical to that sent by $\hat{q}$, $H()$ has yielded two different outputs (*i* and *j*) for equal inputs, contradicting the assumption that it is a function. □

4.5 Evaluation

To better understand OldBlue's performance we created a protocol simulator using NS-3. We measured OldBlue's throughput and message delivery latency by sending null messages in sessions while varying parameters as indicated in Table 4.1.

For each configuration, we simulated three runs of up to 5 minutes of protocol interaction with the following simplifying assumptions. Processes use the estimated round trip time (RTT) to other processes to delay `LostMsg` requests and retransmissions. Because estimation is not part of this work, we fixed the estimated RTT to other processes at 4x the actual RTT. We conservatively estimated that each encryption or decryption operation took 4μs and that each digital signature sign and verify operation took 480μs. These values were obtained by running a microbenchmark that encrypts and signs 1 KB messages using AES-128 in CTR mode and RSASSA-PKCS1-v1_5 on our test system – an 8-core Intel Xeon 2.67 GHz machine with 12 GB RAM running Ubuntu Linux 10.04 x86_64.

All session processes were locally connected to a simulated half-duplex Ethernet LAN and ran OldBlue over UDP. Because we wanted to test the protocol without assuming hardware or IP multicast, simulated processes unicast messages to all other session processes. In contrast to the protocol pseudocode above, when a process fulfills a `Retransmit` request, it multicasts the lost message to all processes.

Figures 4.7 and 4.8 depict our measurements of the latency and throughput of null

Table 4.1: Simulation parameter choices

Number of processes	2, 3, 5, 10
Packet loss rate	0, 0.01, 0.05, 0.1, 0.2
Round Trip Time (ms)	2, 10, 20, 100, 200
Number of messages	1,000 or 5 minutes of sim. time

message delivery. In figures which vary RTT the network does not drop any packets. In figures which vary packet loss rate, the network RTT is fixed at 2 ms. Message delivery latency is reported in multiples of the network RTT. The latency vs. RTT are ideal in the sense that, for fixed session size, even as RTT increases the latency remains close to a constant multiple of the RTT. An optimized implementation should be able to reduce the overhead. As packet loss rate increases, latency increases.

Because all processes are connected to a simulated half-duplex Ethernet LAN, permitting only one process to transmit at a time, increasing RTT causes a corresponding linear decrease in the effective bandwidth of the network connection. The graph of throughput vs. RTT has an approximate slope of -4. This is expected given the 4x overestimate for RTT that the simulator provides to processes (increasing the effects of increasing RTT fourfold). As a result, this simulation depicts a pessimistic lower bound for throughput. An implementation on a more realistic network with an accurate RTT estimation strategy should show even stronger throughput performance.

Throughput also fairs well under packet loss when network characteristics are taken into account. Because the simulator simulates broadcast by n unicasts a lost message will create $2 \cdot n$ additional messages in the best case (n `LostMsg` requests and a retransmission to n processes). In most cases, the decrease in throughput is consistent with approximately $2 \cdot n$ additional messages being sent as the result of each message loss.

Examining the effect of session size on throughput and latency we see that these performance metrics appear to decrease proportional to the square of the group size. This is unsurprising due to total amount of traffic between all processes growing proportionally to the square of the number of processes because of the use of direct unicast between all processes.

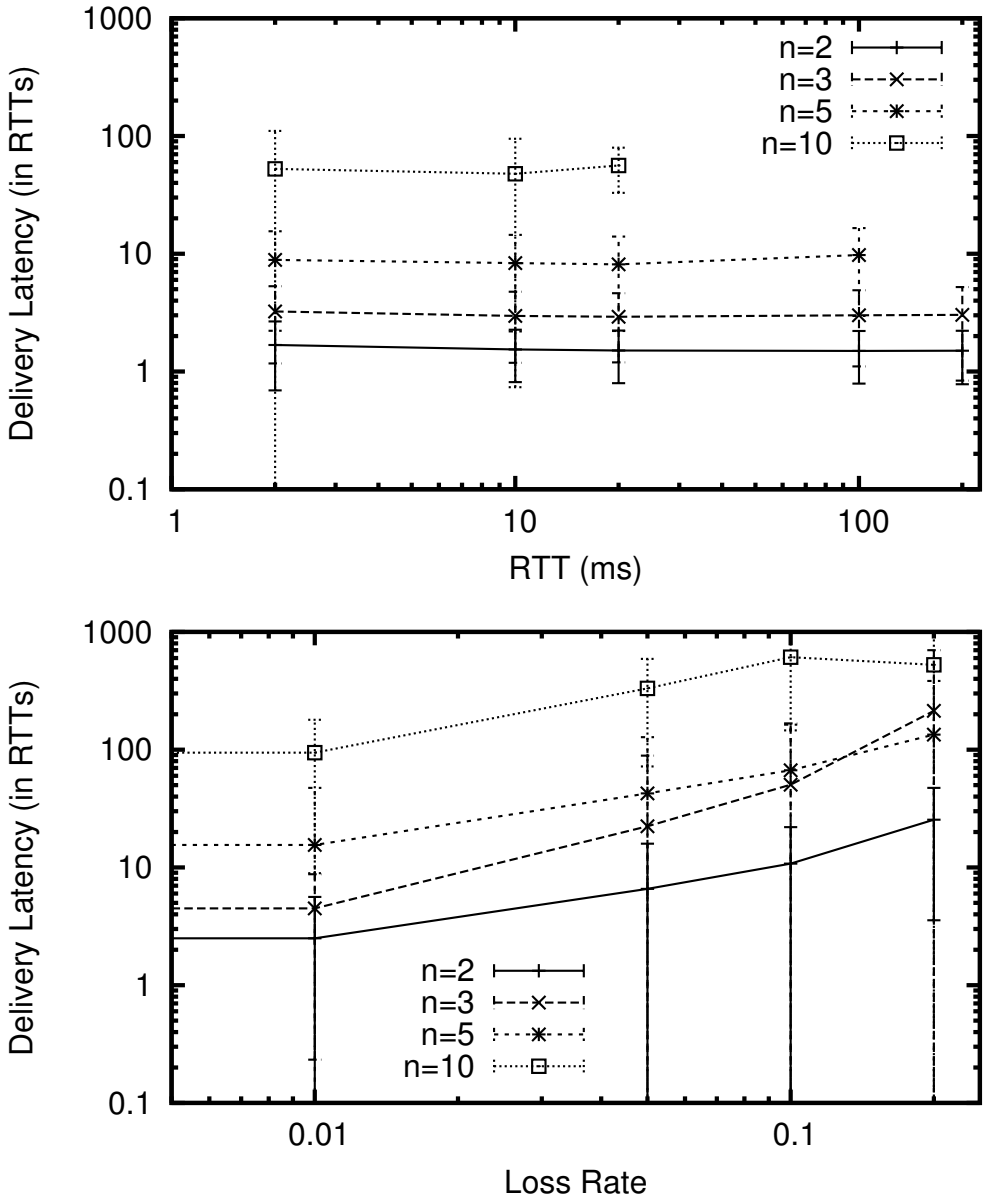

Figure 4.7: Null message delivery latency for sessions of various size with varying RTT (top) and loss rate (bottom).

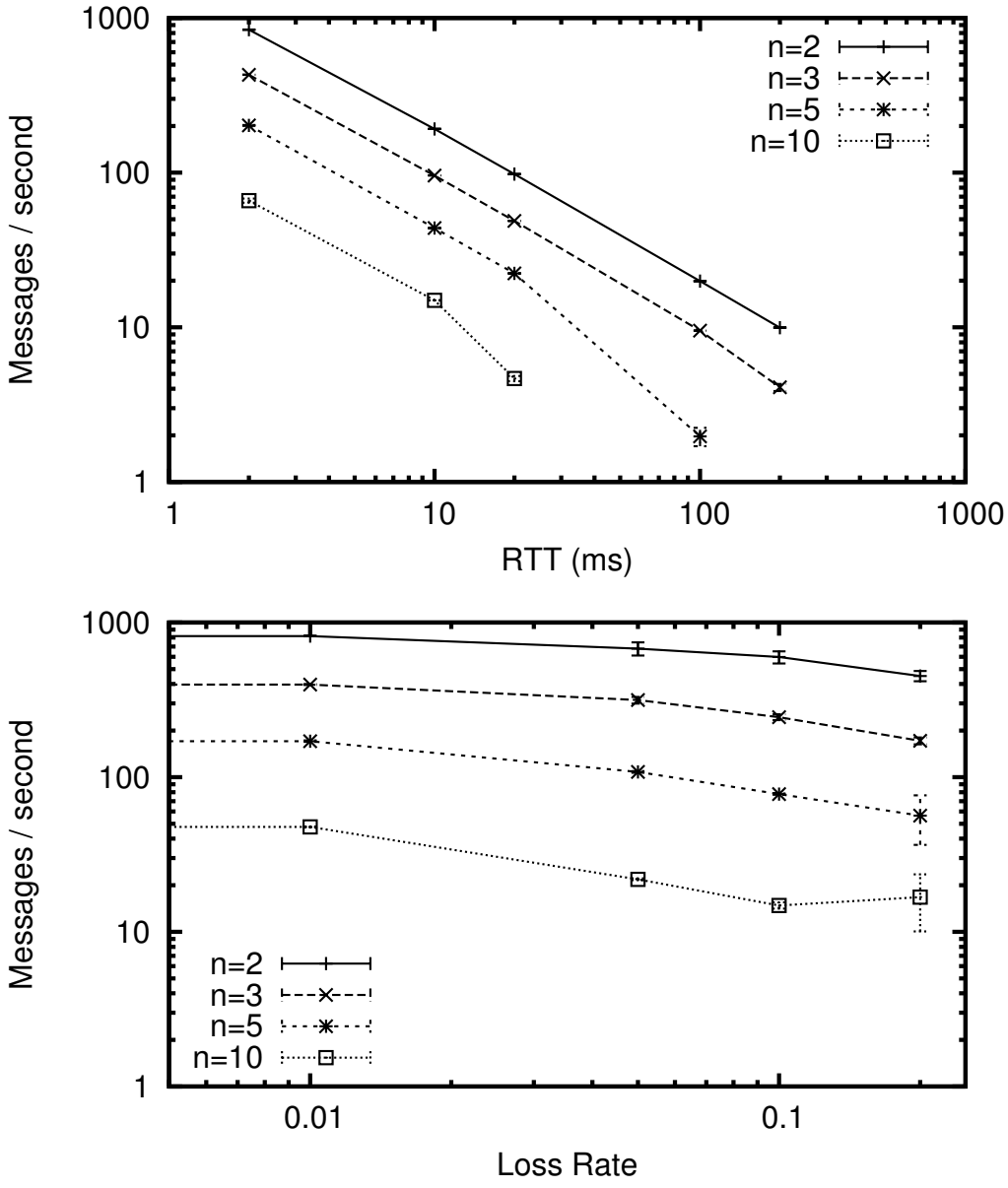

Figure 4.8: Null message delivery throughput for sessions of various size with varying RTT (top) and loss rate (bottom).

4.6 Implementation Considerations

In the foregoing discussion, we have presented simplified primitives for clarity and ease of analysis. Real-world implementations must address a number of additional trade-offs. However, as long as the basic assumptions of the primitives presented above are not violated, implementers can be certain that the implementation will satisfy its formal properties.

Limiting Process State. In practice, the amount of state that a process must maintain must be limited. The size of *Delivered* and *Wire* can be limited by garbage collecting messages that will not require retransmission (i.e. *stable* messages which have been delivered by all processes). The adversary can prevent any message from becoming stable by causing a corrupt process to refuse to deliver it or refusing to transmit an associated message to a correct process, essentially disconnecting the affected process and forcing all connected correct processes to retain all causally newer messages. The protocol will eventually need to block the application or end the session. In this scenario, implementations should enable the application to determine if it should block or end the session by notifying it of the set of unstable messages and the processes that have not delivered them.

The size of *Undelivered* can be bounded by fixing a maximum (L_{U}) and storing only the L_{U} causally-oldest messages from each process. The causal ordering over all messages from a given process can be trivially determined if each correct process includes a monotonically increasing sequence number in each message.

The size of *Requests* can be limited by fixing a maximum number of unsent outgoing messages that may be enqueued before blocking the application, generating `LostMsg` requests on-demand when a `LostMsg` request may be sent, and including a sequence number in `LostMsg` requests so that processes need only store `Retransmit` requests for the L_{L}

most recent `LostMsg` requests from each process.

Minimizing Unnecessary Retransmission Requests. The protocol above enqueues `LostMsg` requests whenever a message arrives before one of its causal predecessors. To avoid sending `LostMsg` requests for messages that have merely been delayed, implementations must delay sending `LostMsg` requests for a reasonable amount of time. Allowing `LostMsg` requests to represent more than one message id can further reduce the number of messages sent.

Minimizing Duplicate Retransmissions. Processes in the protocol above broadcast `LostMsg` messages to all session members and all correct session members will retransmit the message to the requesting process. Techniques to minimize duplicate retransmissions must take into account the properties of the underlying network channel. However, implementations can reduce duplicate retransmissions by having processes request lost messages from other processes one-by-one, allowing explicit cancellation of `Retransmit` requests, or automatically canceling `Retransmit` requests for any process $\hat{p}$ for any messages that are causally older than a newly-received message authored by $\hat{p}$.

Congestion Control. In order to make efficient use of network resources, implementations should implement a congestion control mechanism. In our design, congestion control is encapsulated by the $eligible()$ function which requires that for each process $\hat{p}$ or request r that is not eligible at time t will eventually become eligible at some time $t' > t$. Thus congestion control may only delay fulfillment of a request for a finite span of time. In order to coincide with an intuitive understanding of the Fairness definition, c corrupt processes should only be able to affect the congestion control mechanism proportional to $\frac{c}{n}$.

In a unicast implementation, we believe fair congestion control can be achieved by maintaining a separate outstanding message window for each process. The window size is increased only when the associated process proves the ability to receive messages at the current rate. The window size is decreased in response to authenticated retransmission requests from the process or the absence of such proofs.

In a multicast implementation, we believe fair congestion control can be achieved by maintaining separate window sizes as in the unicast case, and taking their average to calculate a single outgoing window size.

Strengthening Validity. Our Validity definition does not require a correct process to deliver a message if the adversary does not deliver all associated messages. However, as a matter of practicality, we'd like processes to learn if the most recent message in a low-rate session is lost. This can be aided by sending a heartbeat message if a chosen time threshold elapses since the last new outgoing messages was sent. The heartbeat message should include the contents of *Frontier* to allow processes to determine if they have missed a message. Heartbeat messages should be sent outside of the causal ordering so that lost, or delayed, heartbeat messages will not be retransmitted.

4.7 Related Work

Broadcast protocols for distributed systems and mechanisms for preserving various ordering properties have a long history in the literature. Despite this, we are aware of only one protocol [61], developed concurrently with OldBlue, that provides always-available Causal Broadcast in the threat model of Section 4.2. Many protocols [68, 3, 82, 40, 11] are not se-

cure against Byzantine adversaries that can corrupt system members. Many protocols that are secure against a Byzantine adversary [46, 21, 86, 6, 55] either cannot ensure liveness among connected processes during a network partition or are not resilient to adversaries which can corrupt $t \geq \frac{n}{3}$ nodes. We compare concepts shared by most of these systems with OldBlue's requirements below.

Psync. The causal broadcast property provided by OldBlue was partially inspired by Psync [82]. Psync provides causally ordered IPC for distributed systems in an environment subject to benign failures. Because Psync assumes that processes are correct, it is susceptible attack by corrupt processes. For instance, a duplicitous process can cause correct processes to deliver conflicting messages leading to differing causal histories. There is no mechanism to ensure that correct processes will learn that their views differ during subsequent communication. Psync does not attempt to enforce fairness. This allows corrupt processes to expend the resources of correct processes unchecked.

Reliable Broadcast. The properties of OldBlue are very similar to, and were inspired by, Reliable Broadcast. Reliable Broadcast provides a mechanism to broadcast messages such that all correct processes deliver the same set of messages and all correct processes deliver all messages broadcast by all correct processes. Hadzilacos and Toueg [46] demonstrate that Reliable Broadcast can be extended in a modular fashion to provide various guarantees on message delivery order.

A protocol provides Reliable Broadcast if it ensures the following four properties, as given by Cachin et al. [21]. In the following, each message is associated with a tag. The tag $ID.j.s$ is used to indicate the message with sequence number s sent by correct process P_j in session ID.

R-Validity If a correct process has r-broadcast $\mathtt{m}$ tagged with $ID.j.s$, then all correct processes r-deliver $\mathtt{m}$ tagged with $ID.j.s$, provided all correct processes have been activated on $ID.j.s$ and the adversary delivers all associated messages.

R-Consistency If some correct process r-delivers $\mathtt{m}$ tagged with $ID.j.s$ and another correct process r-delivers $\mathtt{m}'$ tagged with $ID.j.s$, then $\mathtt{m} = \mathtt{m}'$.

R-Totality If some correct process r-delivers a message tagged with $ID.j.s$, then all correct processes r-deliver some message tagged with $ID.j.s$, provided all correct processes have been activated on $ID.j.s$ and the adversary delivers all associated messages.

R-Authenticity For all ID, senders P_j, and sequence numbers s, every correct process r-delivers at most one message $\mathtt{m}$ tagged with $ID.j.s$. Moreover, if P_j is correct, then $\mathtt{m}$ was previously r-broadcast by P_j with sequence number s.

The definition of OldBlue purposefully differs from Reliable Broadcast in the following ways:

1. R-Validity requires a correct process to r-deliver a message only if the adversary delivers all associated messages. By contrast, Validity (Definition 2) ensures liveness among processes in a connected component during a network partition by requiring *any* $\hat{q}$ that receives all messages associated with $CB.broadcast(\mathtt{m})$ to deliver $\mathtt{m}$.

2. R-Totality ensures that all correct processes deliver the same set of messages regardless of the correctness of the author. This precludes liveness during arbitrary network partitions. E.g. When each correct $\hat{p}$ delivers $\mathtt{m}$, $\hat{p}$ must send "associated messages" such that all other correct $\hat{q}$ will also deliver $\mathtt{m}$. Causal Consistency (Definition 4)

is limited to pairwise guarantees, between a correct sender and deliverer, to allow liveness during partition.

3. Authenticity (Definition 5) provides both R-Consistency and R-Authenticity.

Consistent Broadcast. A Consistent Broadcast [21] protocol satisfies R-Validity, R-Consistency, and R-Authenticity above but *not* R-Totality. As noted above, R-Validity conflicts with our requirements. Therefore, Consistent Broadcast is unsuitable for our purposes as well.

Byzantine Agreement. Byzantine Agreement, the Byzantine Generals' Problem [57], and the closely related Consensus [46] Problem, refer to the problem of ensuring that all processes in a distributed system agree on a value proposed by one, or more, processes. Like Reliable Broadcast, any protocol which depends on Byzantine Agreement is incompatible with OldBlue's formal requirements. The agreement property of Byzantine Agreement is similar to R-Totality – if a correct process decides x, all correct processes eventually decide x and, if the proposing process was correct, x was the value proposed. Unless all messages are known a priori, this property is incompatible with Validity – processes in a connected component must be able to deliver messages even during a network partition.

CAP and CAC. As mentioned in Section 4.1, the CAP Theorem [19, 42] states that, during a network partition, a distributed system cannot maintain availability while preserving consistency. The specific consistency notion used in the impossibility proofs of [42] is atomic consistency. This prompted a shift in distributed databases toward weaker consistency guarantees [32, 108], such as eventual consistency, that are compatible with always availability. After the development of OldBlue, we have become aware of recent

work by Mahajan et al. [61] which extends the results of [42] in a slightly different framework. The CAC model of [61] reframes the problem in terms of Consistency, Availability, and Convergence. Consistency restricts the order that system events appear to occur. For example, FIFO consistency requires that messages from a single author are delivered in the order they were sent without placing ordering constraints on messages from different authors. Causal consistency is stronger, requiring that messages be delivered in causal order. Availability specifies which system operations are guaranteed to complete under any failure conditions allowed by the network model. For instance, in an always available implementation, if nodes $\hat{p}$ and $\hat{q}$ can communicate and $\hat{p}$ issues a read or write to $\hat{q}$, that read or write will complete regardless of which messages to other nodes are lost. Convergence ensures that when two nodes can communicate, in the absence of further updates, that they will reach the same end state. Convergence deserves explicit treatment because some consistency models, such as fork-causal consistency [61], achieve always availability and consistency by sacrificing convergence — in the instance of a duplicitous adversary that partitions the group into $\mathcal{P}_1$ and $\mathcal{P}_2$ and sends concurrent messages to each partition ($\mathtt{m}_1 \neq \mathtt{m}_2$, respectively), under fork consistency partitions $\mathcal{P}_1$ and $\mathcal{P}_2$ are never permitted to rejoin and arrive at a shared, consistent, global state.

Given these definitions, Mahajan et al., show that, in a Byzantine environment where network partitions may occur, no distributed system can remain always available and convergent while ensuring a number of different causal consistency properties. Therefore, this impossibility result also applies to stronger consistency notions such as Reliable Broadcast, Consensus, and Byzantine Agreement. The authors then present a variant of causal consistency, *bounded fork join causal consistency*, that is compatible with always availability and convergence in a Byzantine environment. Bounded fork join causal consistency has

three key differences when compared to other causal consistency definitions: it limits its serial operation ordering requirement to correct nodes, it allows correct nodes to observe concurrent writes by a Byzantine nodes, and correct nodes bound the number of concurrent writes that Byzantine nodes can perpetrate by proactively detecting concurrent writes and refusing to accept subsequent writes by known-Byzantine nodes that have not already been accepted by another correct node.

Knowing that no restriction can be placed on the behavior of a Byzantine process, we carefully defined all of OldBlue's properties in terms of pairwise guarantees about the operations of correct processes. In OldBlue's definition of Causal Precedence (Definition 1), clauses (2) and (3) restrict serial operation ordering to correct processes and clause (3) allows correct processes to observe concurrent messages from Byzantine processes without violating causal ordering because the causal order of a message is not defined until it is delivered by a correct process. In this way, OldBlue does not fall victim to the impossibility result of [61] and an OldBlue implementation in which correct processes proactively detect duplicitous processes and cease to deliver messages from them provides bounded fork join causal consistency.

KleeQ. KleeQ [84] is a protocol that provides confidentiality and consensus for group communication between ad-hoc groups of processes with intermittent connectivity. Because the authors of KleeQ target the benign failure model, KleeQ does not attempt to enforce Fairness between processes. As presented, even in the presence of Byzantine adversaries, KleeQ will ensure Validity, Authenticity, and eventual detection of Causal Consistency violations.

In OldBlue, we prefer to prevent violations of Causal Consistency in the first place

rather than attempting to detect them at a later time. Because KleeQ is designed to operate over ad-hoc networks with intermittent connectivity, it does not take advantage of broadcast and multicast capabilities or regular connectivity between processes. Interactions between processes are accomplished pairwise, preventing opportunistic load balancing. Thus, to some extent, OldBlue can be viewed as a complementary protocol for the traditional network setting which addresses Byzantine failures at the outset.

4.8 Future Work

In this work, we have made certain simplifying assumptions that should be addressed by future work. For example, we used the $eligible()$ function to abstract strategies which balance timely detection of losses against minimization of unnecessary retransmissions and congestion control. The strategies employed by current Internet transport protocols have not been designed to guarantee correct operation in a Byzantine environment. Further research is necessary to provide multicast congestion control, loss detection, and retransmission delay strategies that are guaranteed to function properly in the Byzantine setting.

Another simplifying assumption that will need to be addressed in practice is session initiation. In our model, a session begins when $CB.open()$ is invoked with the set of participants which will take part in the current session. However, as discussed in [16], just determining when a session has or has not started successfully and between which participants is not straightforward in a Byzantine environment. In practice, a session membership algorithm which resists Byzantine failures will need to be employed.

It may also be possible to provide stronger consistency guarantees in times of full connectivity when few Byzantine failures occur. OldBlue adopts a pessimistic approach, as-

suming that each session may always be in a worst-case scenario. However, the impossibility results of the CAP and CAC theorems apply to times of network partition. It is likely that systems could provide stronger consistency guarantees in times of full connectivity, only relaxing those guarantees to causal consistency in order maintain availability when a partition occurs. This approach may also lead to a number of opportunities for network performance optimization.

Chapter 5

Conclusions and Future Work

The intuition and instincts which serve to protect us in the physical world may not be sufficient to keep us safe online. While major advances in user experience have brought us computing systems that are increasingly intuitive to operate, the hardware and software systems users operate daily have increased in complexity by orders of magnitude, further obscuring the precise technical details of their operation from the average user. As a result, common Internet protocols fail to meet users' reasonable security expectations in a number of ways. In this work we address three major issues in communication privacy and integrity: software integrity in web applications, secure multi-party instant messaging, and integrity in distributed communication protocols subject to Byzantine failures.

As the web has matured, the capabilities of web browsers have evolved from merely displaying static documents to providing a general purpose computing platform capable of simultaneously running multiple client-side applications from different origins. To protect the privacy and integrity of application data, web browsers attempt to enforce isolation between application code from different origins. Unfortunately, cross-site scripting (XSS)

vulnerabilities allow attackers to violate isolation between origins by laundering malicious code through a trusted server. In Chapter 2, we presented Noncespaces, an end-to-end system for preventing XSS attacks. The core insight of Noncespaces is that if the server can reliably identify and annotate untrusted content, the client can enforce flexible policies that prevent XSS attacks while safely allowing rich user-contributed content. The core technique of Noncespaces uses randomized (X)HTML tags to identify and annotate untrusted content, similar to the use of Instruction Set Randomization to defeat binary code injection attacks. Noncespaces frees the server from the burden of sanitizing untrusted content, avoiding all the difficulties and problems with sanitization. Once the server annotates a document node as untrusted, no malicious content in the document node may escape the node or raise its trust classification. Noncespaces-aware clients can reliably prevent all the attacks that the configured web application policy prohibits, and even Noncespaces-unaware clients can prevent node-splitting attacks. We implemented a prototype of Noncespaces as an extension to a popular web application template system and a client-side proxy and show via experimentation that the overhead of our prototype Noncespaces implementation is moderate.

In Chapter 3 we turned our attention to another online communication medium, Instant Messaging (IM). Instant messaging is popular for real-time interactive communication online and can be used in environments where the server-must-be-trusted security model of the web is an ill fit. Unfortunately, most modern IM protocols fail to ensure confidentiality or integrity in situations where the network and/or the server cannot be trusted. OTR [13] brings confidentiality, integrity, and deniability guarantees to two-party IM conversations. However, before our work, there was no solution that provided strong security guarantees for multi-party conversations. Our proposed framework for multi-party Off-the-Record

communication (mpOTR) does not depend on a central server; instead we developed a model that mimics a typical private meeting where each user authenticates the other participants for himself. We identified three main goals for mpOTR: confidentiality, consensus, and repudiation. We achieve confidentiality via standard cryptographic measures. Consensus is based on unforgeable signatures which allow non-repudiation among chatroom participants. Repudiation in front of non-chatroom participants is based on a user's ability to disassociate from the signing key pair. The crucial step in our solution is the distribution of chatroom-specific signature keys, which become the authentication mechanism during the chat. Deniability is a consequence of the forward secrecy and deniability of the key agreement protocol that is used to establish authentic, confidential, deniable channels between pairs of parties.

Though mpOTR provides a basic framework for confidential, deniable multi-party instant messaging there are a number of ways that chat servers and malicious insiders can seek to violate consensus or induce resource starvation between honest chat participants. These issues are not unique to mpOTR, they must be addressed by any distributed group communication protocol which operates in a Byzantine environment. Unfortunately, nearly all previous solutions are either insecure in a Byzantine environment or sacrifice availability when network partitions occur. This prompted us to develop OldBlue, a causal broadcast protocol for distributed group communication (described in Chapter 4). OldBlue guarantees causal consistency and availability between connected participants while tolerating an arbitrary number of Byzantine failures. We have formally defined the availability, consistency, and authenticity properties that OldBlue seeks to achieve. We provide preliminary simulation results to better characterize OldBlue's network performance and prove that the proposed protocol satisfies the formal definitions. Recent research [61] shows

that the consistency guarantees provided by OldBlue appear to be close to the strongest achievable for an always available distributed system in a Byzantine environment.

5.1 Future Work

While we feel that this work advances the state of the art in information security in several online areas, with the rapid pace of technological change, we would be foolish to think that we have closed the book on any of these matters. Here we briefly consider directions for future research.

5.1.1 Cross-Site Scripting (XSS) Defenses

As can be seen from Noncespaces' related work section (Section 2.6), there have been a myriad of approaches to solve XSS both before and after Noncespaces. And yet, XSS still continues to be a major problem years since Noncespaces initial publication in 2009. The web has a significant legacy application problem on two separate fronts: server-side and client-side. Server-side web applications in use today have been built from a wide range of frameworks and languages, making it difficult to provide server-side protection mechanisms that transcend language and software architecture boundaries. Web application developers that wish to employ the latest defenses are faced with visitors using a wide array of browser families and versions on a variety of platforms that may not support the latest client-side technologies and which may have any number of vendor-specific non-standard quirks [83] that introduce opportunities for ambiguity between the client and server.

This is not to say that there has not been progress. A number of template systems have integrated some form of auto-sanitization [29, 81, 103, 41], which attempts to auto-

matically identify and neutralize malicious inputs, making it easy to provide better protection for new applications. However, in many cases [41, 103], the sanitization is context-insensitive leading to the possibility of incorrect sanitization and compromise. Content Security Policy has become a W3C Candidate Recommendation [99] and is being implemented in recent versions of Firefox, Chrome, Safari, and Internet Explorer. [30] However, it will still be some time before any one solution reaches universal adoption. Which of the variety of approaches will provide the best security-effort tradeoff is still unknown. If a language-agnostic solution that could protect legacy server-side code while maintaining compatibility with legacy browsers could be found, it would likely outdistance all efforts to date.

5.1.2 Confidential Multi-Party Communication

In Section 3.5 we highlighted a number of areas of future work for mpOTR, many of which were addressed by OldBlue. However, there are areas where further progress can be made. For example, the development of specialized primitives could improve the efficiency of mpOTR's cryptographic operations. The pairwise Deniable Signature Key Exchange algorithm presented is an obvious target for improvement. Combining the Group Key Agreement that must already be performed with a multi-party Deniable Signature Key Exchange protocol would be likely to reduce its overhead significantly.

Alternate notions of deniability are another interesting area for future research. After examining other notions of deniability [22, 34], we settled on deniability through forgeability of valid transcripts by an outsider as the appropriate deniability notion for mpOTR. There are multiple algorithms compatible with this notion [53, 14, 69]. However, other definitions of deniability are possible and may have unexpected benefits over the notion

employed by mpOTR.

5.1.3 Causal Broadcast in a Byzantine Environment

There are several areas for future work outlined in Section 4.8. The two most interesting are providing stronger consistency properties in the optimistic case and fair use of network resources with existing Internet protocols. A number of Byzantine Agreement protocols contain optimizations which allow greater performance when processes are behaving correctly and which fall back to a pessimistic protocol in the presence of Byzantine behavior [23, 55, 28]. When all participants are fully connected and no nodes are known to be Byzantine, it may be possible to provide stronger consistency while gracefully degrading to causal consistency when a network partition or Byzantine behavior is observed.

New Internet transport protocols are expected to be TCP-friendly: that is, they should use no more than a proportional fair share of network resources when run alongside TCP sessions. In practice, this means implementing congestion control models which achieve the same average flow rates as TCP. The multicast paradigm of OldBlue, collaborative retransmission which allows a lost message authored by $\hat{p}$ to be retransmitted by another process $\hat{q}$, and the possibility that any other process may be Byzantine all complicate the goal of ensuring TCP-friendly congestion control for OldBlue. DCCP [52] attempts to prevent malicious peers from lying about network congestion conditions in order to provide TCP-friendly congestion control for two-party unreliable network communication. Investigation of the applicability of DCCP's, or other multicast congestion control schemes, to the maximally Byzantine setting may yield interesting results.

References

[1] ab - Apache HTTP server benchmarking tool. http://httpd.apache.org/docs/2.2/programs/ab.html, 2010.

[2] C. Alexander and I. Goldberg. Improved User Authentication in Off-The-Record Messaging. In *Proceedings of the 2007 ACM workshop on Privacy in the electronic society*, WPES '07, pages 41–47, New York, NY, USA, 2007. ACM.

[3] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Transis: A Communication Sub-System for High Availability. In *Proceedings of the Twenty Second International Symposium on Fault-Tolerant Computing*, pages 76–84, July 1992.

[4] J. Angwin. The Web's New Gold Mine: Your Secrets. http://online.wsj.com/news/articles/SB10001424052748703940904575395073512989404, July 2010.

[5] D. Austin, S. Peruvemba, S. McCarron, M. Ishikawa, and M. Birbeck. XHTML Modularization 1.1. Technical report, W3C, Oct. 2008.

[6] M. Backes and C. Cachin. Reliable broadcast in a computational hybrid model with byzantine faults, crashes, and recoveries. In *Proceedings of the International*

Conference on Dependable Systems and Networks, DSN 2003, pages 37–46. IEEE, 2003.

[7] E. G. Barrantes, D. H. Ackley, S. Forrest, T. S. Palmer, D. Stefanović, and D. D. Zovi. Randomized Instruction Set Emulation to Disrupt Binary Code Injection Attacks. In *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS)*, CCS 2003, pages 281–289, New York, NY, USA, Oct. 2003. ACM Press.

[8] M. Bellare and C. Namprempre. Authenticated Encryption: Relations Among Notions and Analysis of the Generic Composition Paradigm. *Journal of Cryptology*, 21(4):469–491, Sept. 2008.

[9] M. Bellare and P. Rogaway. Entity Authentication and Key Distribution. In *Advances in Cryptology — CRYPTO '93*, pages 232–249. Springer, 1994.

[10] J. Bian, R. Seker, and U. Topaloglu. Off-the-Record Instant Messaging for Group Conversation. In *Proceedings of Information Reuse and Integration*, IRI '07, pages 79–84. IEEE Computer Society, 2007.

[11] K. P. Birman and T. A. Joseph. Reliable Communication in the Presence of Failures. *ACM Transactions on Computer Systems*, 5(1):47–76, 1987.

[12] J.-M. Bohli, M. I. González Vasco, and R. Steinwandt. Secure Group Key Establishment Revisited. *International Journal of Information Security*, 6(4):243–254, June 2007.

[13] N. Borisov, I. Goldberg, and E. Brewer. Off-the-record communication, or, why not to use PGP. In *Proceedings of the ACM workshop on Privacy in the electronic society (WPES)*, pages 77–84, New York, NY, USA, 2004. ACM.

[14] C. Boyd, W. Mao, and K. G. Paterson. Deniable authenticated key establishment for Internet protocols. In B. Christianson, B. Crispo, J. A. Malcolm, and M. Roe, editors, *Security Protocols, 11th International Workshop, Revised Selected Papers*, volume 3364 of *LNCS*, pages 255–271, Berlin, Germany, 2005. Springer Verlag.

[15] S. W. Boyd and A. D. Keromytis. SQLrand: Preventing SQL Injection Attacks. In *Proceedings of the 2nd Applied Cryptography and Network Security (ACNS) Conference*, pages 292–302, 2004.

[16] G. Bracha and S. Toueg. Asynchronous Consensus and Broadcast Protocols. *Journal of the ACM*, 32(4):824–840, Oct. 1985.

[17] T. Bray, D. Hollander, A. Layman, and R. Tobin. Namespaces in XML 1.0 (Second Edition). Technical report, W3C, Aug. 2006.

[18] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau. Extensible Markup Language (XML) 1.0 (Fourth Edition). Technical report, W3C, Sept. 2006.

[19] E. Brewer. CAP Twelve Years Later: How the "Rules" Have Changed. *Computer*, 45(2):23–29, Feb 2012.

[20] S. Burbeck. How to use Model-View-Controller (MVC). `http://st-www.cs.uiuc.edu/users/smarch/st-docs/mvc.html`, 1992.

[21] C. Cachin, K. Kursawe, F. Petzold, and V. Shoup. Secure and Efficient Asynchronous Broadcast Protocols. Report 2001/006, Cryptology ePrint Archive, 2001. http://eprint.iacr.org/2001/006.

[22] R. Canetti, C. Dwork, M. Naor, and R. Ostrovsky. Deniable Encryption. In B. S. K. Jr., editor, *Advances in Cryptology — CRYPTO 1997*, volume 1294 of *LNCS*, pages 90–104. Springer-Verlag, Aug. 1997.

[23] M. Castro and B. Liskov. Practical Byzantine Fault Tolerance. In *Proceedings of the 3rd Operating Systems Design & Implementation (OSDI)*, pages 173–186, Berkeley, CA, USA, 1999. USENIX The Advanced Computing Systems Association.

[24] CERT Coordination Center. CERT Advisory CA-2000-02 Malicious HTML Tags Embedded in Client Web Requests. http://www.cert.org/advisories/CA-2000-02.html, 2000.

[25] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer. Non-Control-Data Attacks Are Realistic Threats. In *USENIX Security Symposium*, pages 177–192. USENIX The Advanced Computing Systems Association, Aug. 2005.

[26] S. M. Cherry. IM means business. *IEEE Spectrum*, 38:28–32, November 2002.

[27] China employs two million microblog monitors state media say. http://www.bbc.co.uk/news/world-asia-china-24396957, Oct. 2013.

[28] A. Clement, E. Wong, L. Alvisi, M. Dahlin, and M. Marchetti. Making Byzantine Fault Tolerant Systems Tolerate Byzantine Faults. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 153–168, Berkeley, CA, USA, 2009. USENIX Association.

[29] Closure Tools — Security. https://developers.google.com/closure/templates/docs/security.

[30] Content Security Policy — Web Security. https://www.w3.org/Security/wiki/Content_Security_Policy#Implementations, Feb. 2014. W3C.

[31] 2010 CWE/SANS Top 25 Most Dangerous Software Errors. http://cwe.mitre.org/top25/, 2010.

[32] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's Highly Available Key-value Store. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP)*, SOSP '07, pages 205–220, New York, NY, USA, 2007. ACM Press.

[33] M. Di Raimondo, R. Gennaro, and H. Krawczyk. Secure Off-the-Record Messaging. In *Proceedings of the 2005 ACM workshop on Privacy in the electronic society*, WPES '05, pages 81–89, New York, NY, USA, 2005. ACM.

[34] M. Di Raimondo, R. Gennaro, and H. Krawczyk. Deniable Authentication and Key Exchange. In *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS)*, CCS 2006, pages 400–409, New York, NY, USA, Oct. 2006. ACM Press.

[35] C. Dwork, M. Naor, and A. Sahai. Concurrent Zero-Knowledge. *Journal of the ACM*, 51(6):851–898, 2004. http://www.wisdom.weizmann.ac.il/%7Enaor/PAPERS/time.ps.

[36] B. Eich. JavaScript: Mobility & Ubiquity. In G. Barthe, H. Mantel, P. Müller, A. C. Myers, and A. Sabelfeld, editors, *Mobility, Ubiquity and Security*, number 07091 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2007. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany.

[37] Ú. Erlingsson, B. Livshits, and Y. Xie. End-to-end Web Application Security. In *Proceedings of the 11th USENIX Workshop on Hot Topics in Operating Systems*, HOTOS '07, pages 1–6, Berkeley, CA, USA, 2007. USENIX Association.

[38] L. Fadel. Detention Of Al-Jazeera Journalists Strains Free Speech In Egypt. `http://www.npr.org/blogs/thetwo-way/2014/01/29/268481172/detention-of-al-jazeera-journalists-strains-free-speech-in-egypt`, Jan. 2014.

[39] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999.

[40] S. Floyd, V. Jacobson, C.-G. Liu, S. McCanne, and L. Zhang. A Reliable Multicast Framework for Light-weight Sessions and Application Level Framing. *IEEE/ACM Transactions on Networking*, 5(6):784–803, 1997. `http://www.icir.org/floyd/papers/srm_ton.pdf`.

[41] Genshi: Python toolkit for generation of output for the web. `http://genshi.edgewall.org/`, 2008.

[42] S. Gilbert and N. Lynch. Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-tolerant Web Services. *SIGACT News*, 33(2):51–59, June 2002.

[43] I. Goldberg, B. Ustaoğlu, M. Van Gundy, and H. Chen. Multi-party Off-the-Record Messaging. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS)*, CCS 2009, New York, NY, USA, Nov. 2009. ACM Press.

[44] L. Greenemeier. T.J. Maxx Parent Company Data Theft Is The Worst Ever. http://www.informationweek.com/tj-maxx-parent-company-data-theft-is-the-worst-ever/d/d-id/1053522, Mar. 2007.

[45] G. Greenwald. Justice Department's pursuit of AP's phone records is both extreme and dangerous. http://www.theguardian.com/commentisfree/2013/may/14/justice-department-ap-phone-records-whistleblowers, May 2013.

[46] V. Hadzilacos and S. Toueg. A Modular Approach to the Specification and Implementation of Fault-Tolerant Broadcasts. Technical Report TR94-1425, Department of Computer Science, Cornell University, Ithaca, NY, USA, May 1994. http://www.cs.toronto.edu/~vassos/research/publications/HT94/paper.ps.gz.

[47] A. Jacobs. Uighur Scholar in Ugly Confrontation With Security Agents. http://sinosphere.blogs.nytimes.com/2013/11/04/uighur-scholar-ilham-tohti, Nov. 2013.

[48] T. Jim, N. Swamy, and M. Hicks. Defeating Scripting Attacks with Browser-Enforced Embedded Policies. In *Proceedings of the International World Wide Web Conference (WWW)*, pages 601–610, Banff, Alberta, Canada, May 2007. ACM.

[49] G. S. Kc, A. D. Keromytis, and V. Prevelakis. Countering Code-Injection Attacks With Instruction-Set Randomization. In *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS)*, CCS 2003, pages 272–280, New York, NY, USA, Oct. 2003. ACM Press.

[50] E. Kirda, C. Kruegel, G. Vigna, and N. Jovanovic. Noxes: A Client-Side Solution for Mitigating Cross Site Scripting Attacks. In *Proceedings of the ACM Symposium on Applied Computing (SAC)*, pages 330–337, Dijon, France, Apr. 2006.

[51] C. Kirkegaard and A. Møller. Static analysis for Java Servlets and JSP. In *Proceedings of the 13th International Static Analysis Symposium, SAS '06*, volume 4134 of *LNCS*, pages 336–352. Springer-Verlag, August 2006. Full version available as BRICS RS-06-10.

[52] E. Kohler, M. Handley, and S. Floyd. Designing DCCP: Congestion Control Without Reliability. In *Proceedings of the conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '06, pages 27–38, New York, NY, USA, 2006. ACM.

[53] H. Krawczyk. SKEME: A Versatile Secure Key Exchange Mechanism for Internet. In *Proceedings of the 1996 Network and Distributed System Security Symposium (NDSS)*, pages 114–127. The Internet Society, Feb. 1996.

[54] B. Krebs. Target: Names, Emails, Phone Numbers on Up To 70 Million Customers Stolen. *Krebs on Security*, Feb. 2014. http://krebsonsecurity.com/2014/01/target-names-emails-phone-numbers-on-up-to-70-million-customers.

[55] K. Kursawe and V. Shoup. Optimistic Asynchronous Atomic Broadcast. In *Proceedings of International Colloqium on Automata, Languages and Programming (ICALP05)*, LNCS 3580. Springer, 2001. Full length version. Revised April 19, 2002.

[56] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, July 1978.

[57] L. Lamport, R. Shostak, and M. Pease. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[58] Largest dataloss incidents. http://datalossdb.org/index/largest.

[59] L. Law, A. Menezes, M. Qu, J. Solinas, and S. Vanstone. An Efficient Protocol for Authenticated Key Agreement. *Designs, Codes and Cryptography*, 28(2):119–134, 2003.

[60] V. B. Livshits and M. S. Lam. Finding Security Vulnerabilities in Java Applications with Static Analysis. In *USENIX Security Symposium*, pages 271–286. USENIX The Advanced Computing Systems Association, Aug. 2005.

[61] P. Mahajan, L. Alvisi, and M. Dahlin. Consistency, Availability, and Convergence. Technical Report UTCS TR-11-22, Department of Computer Science, The University of Texas at Austin, 2011.

[62] D. Malkhi, M. Merritt, and O. Rodeh. Secure reliable multicast protocols in a WAN. *Distributed Computing*, 13(1):19–28, Jan. 2000.

[63] M. Mannan. Secure Public Instant Messaging. Master's thesis, Carleton University, Ottawa, Canada, August 2005.

[64] M. Mannan and P. C. van Oorschot. A Protocol for Secure Public Instant Messaging. In G. Di Crescenzo and A. Rubin, editors, *Financial Cryptography and Data Security – FC 2006*, volume 4107 of *LNCS*, pages 20–35, Anguilla, British West Indies, 2006. Springer Verlag. Full version available at `http://www.scs.carleton.ca/research/tech_reports/2006/download/TR-06-01.pdf`.

[65] G. Markham. Script Keys. `http://www.gerv.net/security/script-keys/`, Apr. 2005.

[66] G. Markham. Content Restrictions. `http://www.gerv.net/security/content-restrictions/`, Mar. 2007.

[67] J.-P. Martin and L. Alvisi. Fast Byzantine Consensus. *IEEE Transactions on Dependable and Secure Computing*, 3(3):202–215, July 2006.

[68] P. Melliar-Smith, L. Moser, and V. Agrawala. Broadcast Protocols for Distributed Systems. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):17–25, 1990.

[69] A. Menezes and B. Ustaoglu. Comparing the pre- and post-specified peer models for key agreement. In Y. Mu, W. Susilo, and J. Seberry, editors, *Information Security and Privacy – ACISP 2008*, volume 5107 of *LNCS*, pages 53–68. Springer, 2008.

[70] R. C. Merkle. *Secrecy, Authentication, and Public Key Systems*. PhD thesis, Stanford University, Stanford, CA, USA, 1979.

[71] L. A. Meyerovich and B. Livshits. ConScript: Specifying and Enforcing Fine-Grained Security Policies for JavaScript in the Browser. In *IEEE Symposium on Security and Privacy*, pages 481–496, Washington, DC, USA, May 2010. IEEE Computer Society.

[72] Microsoft Developer Network (MSDN). About Conditional Comments. `http://msdn.microsoft.com/en-us/library/ms537512.aspx`, 2007.

[73] A. Mislove, G. Oberoi, A. Post, C. Reis, P. Druschel, and D. S. Wallach. AP3: Cooperative, decentralized anonymous communication. In *Proceedings of the 11th ACM SIGOPS European workshop*, EW 11, New York, NY, USA, 2004. ACM.

[74] MITRE Corporation. Vulnerability Type Distributions in CVE. `http://cwe.mitre.org/documents/vuln-trends/index.html`, May 2007.

[75] Y. Nadji, P. Saxena, and D. Song. Document Structure Integrity: A Robust Basis for Cross-site Scripting Defense. In *Proceedings of the 16th Network and Distributed System Security Symposium (NDSS)*, pages 17–36. The Internet Society, Feb. 2009.

[76] A. Nambiar and M. Wright. Salsa: A Structured Approach to Large-Scale Anonymity. In *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS)*, CCS 2006, New York, NY, USA, Oct. 2006. ACM Press.

[77] E. V. Nava and D. Lindsay. Abusing Internet Explorer 8's XSS Filters. `http://p42.us/ie8xss/Abusing_IE8s_XSS_Filters.pdf`, Sept. 2010.

[78] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans. Automatically Hardening Web Applications Using Precise Tainting. In *Proceedings of the 20th IFIP International Information Security Conference (SEC 2005)*, pages 372–382, May 2005.

[79] The NSA Files. `http://www.theguardian.com/world/the-nsa-files`, 2013.

[80] Opera Browser. `http://www.opera.com/browser/`, Dec. 2008.

[81] owasp-jxt — OWASP's Java XML Templates. `https://code.google.com/p/owasp-jxt/`.

[82] L. L. Peterson, N. C. Buchholz, and R. D. Schlichting. Preserving and Using Context Information in Interprocess Communication. *ACM Transactions on Computer Systems*, 7(3):217–246, 1989.

[83] QuirksMode — for all your browser quirks. `http://www.quirksmode.org/`.

[84] J. Reardon, A. Kligman, B. Agala, and I. Goldberg. KleeQ: Asynchronous Key Management for Dynamic Ad-Hoc Networks. Technical Report CACR 2007-03, Center for Applied Cryptographic Research, University of Waterloo, Waterloo, ON, Canada, 2007.

[85] M. Reiter and L. Gong. Preventing Denial and Forgery of Causal Relationships in Distributed Systems. In *IEEE Symposium on Security and Privacy*, pages 30–40, Washington, DC, USA, 1993. IEEE Computer Society.

[86] M. K. Reiter. The Rampart Toolkit for Building High-Integrity Services. In *Selected Papers from the International Workshop on Theory and Practice in Distributed Systems*, pages 99–110, London, UK, 1995. Springer-Verlag.

[87] W. Robertson and G. Vigna. Static Enforcement of Web Application Integrity Through Strong Typing. In *USENIX Security Symposium*, pages 283–298, Berkeley, CA, USA, Aug. 2009. USENIX The Advanced Computing Systems Association, USENIX Association.

[88] D. Ross. IE8 Security Part IV: The XSS Filter. `http://blogs.msdn.com/b/ie/archive/2008/07/02/ie8-security-part-iv-the-xss-filter.aspx`, July 2008.

[89] RSnake. XSS (Cross Site Scripting) Cheat Sheet. `http://ha.ckers.org/xss.html`, June 2008.

[90] Sahi. `http://sahi.co.in/`, 2011.

[91] Same-origin policy — JavaScript. `http://developer.mozilla.org/En/Same_origin_policy_for_JavaScript`, Jan. 2014.

[92] samy. Technical explanation of the MySpace worm. `http://web.archive.org/web/20060208182348/namb.la/popular/tech.html`, Feb. 2006.

[93] Selenium IDE. `http://seleniumhq.org/projects/ide/`, 2011.

[94] O. Shezaf. The Universal XSS PDF Vulnerability. `http://www.owasp.org/images/4/4b/OWASP_IL_The_Universal_XSS_PDF_Vulnerability.pdf`, Jan. 2007.

[95] Smarty Template Engine. http://www.smarty.net/, June 2008.

[96] S. W. Smith and J. D. Tygar. Signed Vector Timestamps: A Secure Protocol for Partial Order Time. Technical Report CMU-CS-93-116, Department of Computer Science, Carnegie Mellon University, 1993.

[97] S. W. Smith and J. D. Tygar. Security and Privacy for Parital Order Time. In *Proceedings of the Parallel and Distributed Computing Systems Conference*, pages 70–79, Oct. 1994.

[98] S. Stamm, B. Sterne, and G. Markham. Reining in the Web with Content Security Policy. In *Proceedings of the 19th International World Wide Web Conference (WWW)*, pages 921–930, New York, NY, USA, Apr. 2010. ACM.

[99] B. Sterne and A. Barth. Content Security Policy 1.0. Candidate recommendation, W3C, Nov. 2012. http://www.w3.org/TR/2012/CR-CSP-20121115/.

[100] Z. Su and G. Wassermann. The Essence of Command Injection Attacks in Web Applications. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 372–382, New York, NY, USA, Jan. 2006. ACM.

[101] S. Tang, C. Grier, O. Aciicmez, and S. T. King. Alhambra: A System for Creating, Enforcing, and Testing Browser Security Policies. In *Proceedings of the 19th International World Wide Web Conference (WWW)*, pages 941–950, New York, NY, USA, Apr. 2010. ACM.

[102] M. Ter Louw and V. N. Venkatakrishnan. Blueprint: Robust Prevention of Cross-site Scripting Attacks for Existing Browsers. In *IEEE Symposium on Security and Privacy*, pages 331–346, Washington, DC, USA, May 2009. IEEE Computer Society.

[103] The Django template language — Automatic HTML escaping. https://docs.djangoproject.com/en/1.6/topics/templates/#automatic-html-escaping.

[104] The Freenet Project. https://freenetproject.org/.

[105] The Open Web Application Security Project. Cross-site Scripting (XSS). https://www.owasp.org/index.php/Cross-site_Scripting_%28XSS%29, 2010.

[106] TikiWiki CMS/Groupware. http://info.tikiwiki.org/tiki-index.php, 2010.

[107] Tor Project. https://www.torproject.org/.

[108] Tunable CAP Controls in Riak. http://docs.basho.com/riak/1.1.0/tutorials/fast-track/Tunable-CAP-Controls-in-Riak/. Basho Technologies, Inc.

[109] M. Van Gundy and H. Chen. Noncespaces: Using Randomization to Enforce Information Flow Tracking and Thwart Cross-Site Scripting Attacks. In *Proceedings of the 16th Network and Distributed System Security Symposium (NDSS)*, pages 55–67. The Internet Society, Feb. 2009.

[110] M. Van Gundy and H. Chen. Noncespaces: Using randomization to defeat cross-site scripting attacks. *Computers & Security*, 31(4):612 – 628, 2012.

[111] W. Venema. Taint support for PHP. `http://wiki.php.net/rfc/taint`, June 2008.

[112] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Cross-Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. In *Proceedings of the 14th Network and Distributed System Security Symposium (NDSS)*. The Internet Society, Mar. 2007.

[113] P. Wang, I. Osipkov, N. Hopper, and Y. Kim. Myrmic: Provably secure and efficient DHT routing. Technical Report 2006/20, Digital Technology Center, University of Minnesota, 2006.

[114] G. Wassermann and Z. Su. Sound and Precise Analysis of Web Applications for Injection Vulnerabilities. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation*, pages 32–41, San Diego, CA, June 2007. ACM Press New York, NY, USA.

[115] G. Wassermann and Z. Su. Static Detection of Cross-Site Scripting Vulnerabilities. In *Proceedings of the 30th International Conference on Software Engineering*, pages 171–180, New York, NY, USA, May 2008. ACM.

[116] P. Wurzinger, C. Platzer, C. Ludl, E. Kirda, and C. Kruegel. SWAP: Mitigating XSS Attacks using a Reverse Proxy. In *Proceedings of the 2009 ICSE Workshop on Software Engineering for Secure Systems*, pages 33–39, Washington, DC, USA, 2009. IEEE Computer Society.

[117] Y. Xie and A. Aiken. Static Detection of Security Vulnerabilities in Scripting Languages. In *USENIX Security Symposium*, pages 179–192, Berkeley, CA, USA, July

2006. USENIX The Advanced Computing Systems Association, USENIX Association.

[118] W. Xu, S. Bhatkar, and R. Sekar. Taint-Enhanced Policy Enforcement: A Practical Approach to Defeat a Wide Range of Attacks. In *USENIX Security Symposium*, pages 121–136, Berkeley, CA, USA, July 2006. USENIX The Advanced Computing Systems Association, USENIX Association.

[119] M. Zalewski. *The Tangled Web: A Guide to Securing Modern Web Applications*. No Starch Press, San Francisco, CA, USA, 1st edition, 2011.