B-CRAM: A Byzantine-Fault-Tolerant Challenge-Response Authentication Mechanism

Akshay Agrawal, Robert Gasparyan, Jongho Shin
{akshayka, robertga, jongho.shin}@cs.stanford.edu

***Abstract*—B-CRAM is an authentication protocol that enables applications to consult a distributed trusted third-party (TTP) in order to validate end-user identities. Short for Byzantine-Fault-Tolerant Challenge-Response Authentication Mechanism, B-CRAM's novelty lies in its resilience to node failures, malicious or otherwise. The TTP layer, which stores end-user credentials, implements the BFT2F replication algorithm. In particular, given a TTP distributed $3f + 1$ ways, our system guarantees both safety and liveness when no more than f nodes fail; moreover, we bound the space of possible attacks when no more than $2f$ nodes fail. B-CRAM prevents adversaries from obtaining sensitive end-user information, no matter how many TTPs they compromise: Unlike many other authentication protocols, B-CRAM ensures that TTPs do not see plaintext end-user passwords, opting for a public key infrastructure instead. To demonstrate our protocol's viability, we integrated B-CRAM with an SMTP server. Preliminary experiments suggest that our implementation of B-CRAM can support modestly sized populations.**

I. Introduction

The problem of authentication is fundamental in computer networks and security – simply put, it is the problem of verifying identities [3]. A particularly widespread flavor of authentication is that of an application server verifying the identities of its users. In the traditional implementation of client-server authentication, the client stores a set of credentials, typically a username and password, with application servers [6]. This approach unfortunately requires clients to remember multiple credentials, and worse still increases the chance that any one of her credentials will be leaked.

Many modern authentication protocols [6], [16], [8], [13] eschew the traditional model by introducing a trusted third party (TTP) between the client and applications servers. In this model, when a client identifies herself to an application server, the application server requests some TTP-provided proof in order to authenticate her. The burden of storing credentials, then, is lifted off of the application servers and placed squarely on the shoulders of the TTP.

Unfortunately, TTP authentication protocols often suffer from fault-intolerance and security vulnerabilities [9]. For example, in the standard implementation of Kerberos, the TTP is a single point of failure – if it were to fail, all authentication would come to a halt. We can alleviate this shortcoming by distributing the KDC and maintaining its state with a replication algorithm; indeed WebAuth, Stanford's own authentication system, does just that [16].

While replication increases redundancy, it does little to guard against byzantine failures. In the context of authentication, fault-tolerance to malicious attacks on the TTPs is particularly important – a compromised system could falsely authenticate imposters. None of the WebAuth, OpenID, or OAuth protocols require that their TTPs survive byzantine faults [16], [8], [6]. Even if the protocols did replicate their data using PBFT [2], security vulnerabilities would remain. The TTPs in each of these protocols see plaintext passwords – an adversary could dump end-users' passwords after compromising but a single node.[1]

In order to address these shortcomings, we propose a new authentication protocol: Byzantine-Fault-Tolerant Challenge-Response Authentication Mechanism, or B-CRAM. Like the previously mentioned protocols, B-CRAM leverages a TTP in order to authenticate end-users to application servers. B-CRAM's novelty lies in its resilience to byzantine faults. We distribute the TTP, which stores user credentials, $3f + 1$ ways. By requiring TTPs to achieve consensus via BFT2F, our system guarantees both safety and liveness when no more than f nodes fail; moreover, we bound the space of possible attacks when no more than $2f$ nodes fail.

B-CRAM also prevents adversaries from obtaining sensitive end-user information, no matter how many TTPs they compromise: Unlike other authentication protocols, B-CRAM ensures that TTPs do not see plaintext end-user passwords, opting for a public key infrastructure instead. In order to demonstrate the viability of B-CRAM, we integrated it into a custom SMTP authentication protocol.

The remainder of this paper describes our network and security model (section two), provides a brief primer on BFT2F (section three), presents the B-CRAM protocol

[1]The protocol in [17] suggests a revision of Kerberos that uses PKI, but it is not byzantine-fault-tolerant.

(section four), briefly details our implementation (section five), quantitatively evaluates B-CRAM (section six), discusses trade-offs (section seven), and proposes future work (section eight).

II. System Model and Assumptions

Our system consists of users, application servers, authentication servers, and the TTP. *Users* are uniquely identified by their public keys. A user stores her public key and private key, encrypted with their password, in the TTP; we refer to this key pair as the user's credentials. The *TTP* consists of $3f + 1$ replicated machines that maintain their state using the BFT2F replication algorithm, where f is a system parameter. We refer to any one of these $3f+1$ machines as a TTP node. The TTP has zero knowledge of users' passwords; all encryption and decryption of private keys is done locally at the user. An set of *authentication servers* service credential-related requests from users and forward them to the TTP; all requests from an authentication server are signed with its individual public key. These servers act as BFT2F clients [10]. Finally, *application servers* are the quantity to which users authenticate.

A public key infrastructure is assumed to enable TTP nodes to authenticate their peers; each node knows the public key of every other node in the TTP, and the public key of every authentication server. Authentication servers know the public key of every TTP node.

We assume a failure model similar to that assumed by Li and Mazieres [10]: our TTP is a networked, asynchronous distributed system where messages between nodes may be delayed, duplicated, reordered, or dropped; however, the network cannot indefinitely delay communication between honest nodes. We additionally assume independent byzantine failures, where nodes may behave arbitrarily. Adversaries cannot forge signatures.

Notationally, we use σ' to denote a public key, σ to denote a private key, uid to denote a user's id, and pwd to denote a user's password. A subscript on the two former symbols communicates that the key belongs to the owner corresponding to that subscript. We use $\{\bullet\}_{K^{-1}}$ to denote that the quantity $\bullet$ is signed with private key K and $\{\bullet\}_K$ to denote that $\bullet$ is encrypted with secret K.

III. Background: BFT2F and Fork* Consistency

Practical byzantine fault tolerance, as proposed by Castro and Liskov, is brittle: As soon as more than f nodes fail, all safety and liveness guarantees are lost [2]. Li and Mazieres' BFT2F replication algorithm builds upon PBFT to provide guarantees about system state when more than f nodes fail, while simultaneously preserving the safety and liveness guarantees that PBFT provides when at most f nodes fail. When no more than f nodes fail, BFT2F is both safe and live. When more than f but fewer than $2f+1$ nodes fail, BFT2F sacrifices liveness but guarantees fork* consistency, restricting adversaries to a contrived attack space.

Fork* consistency is a variant of fork consistency [10]. A particularly nice property of fork consistency is that compromised TTP nodes cannot convince the TTP to commit an operation that no client requested; i.e., TTP nodes cannot invent operations [12]. Under fork consistency, malicious nodes can disrupt the system in one of two ways: (1) compromising liveness and (2) convincing honest nodes to commit different operations for a given sequence number. While malicious nodes can fork the state of the system, as in (2), fork consistency guarantees that clients track a single fork set – a group of at least $2f + 1$ nodes that agrees upon some history of operations. Intuitively, this ensures that clients never see future inconsistent with their histories, and that no client operation crosses multiple forks.

Fork* consistency differs from fork consistency in that a single operation from a client might cross multiple branches; this could happen if the hash chain digest history known by the client corresponded to some state before the system had forked. BFT2F opts for fork* consistency instead of fork consistency because a two-round protocol is required to satisfy the latter, whereas a one-round protocol can satisfy the first. In our context, an operation crossing multiple branches will not likely cause significant damage.

B-CRAM gets two boons from fork* consistency: It allows us to identify a subset of the dishonest nodes – the intersection of any two fork sets consists exclusively of dishonest nodes – and limits intruders to contrived attacks. Once our authentication servers realize the state is forked, they can raise an alarm and an administrator can flush out the intruders. In the general case, under fork* consistency, dishonest nodes will not be able to authenticate imposters; false authentication might occur, however, if the system forked while a user attempted to change her credentials. We explore the types of fork attacks possible in B-CRAM in the next section.

IV. The B-CRAM Protocol

As the name implies, B-CRAM is a byzantine-fault-tolerant challenge-response authentication mechanism. Challenge-response authentication consists of two phases: a challenge, in which the verifier presents some problem to an authenticatee, and a response, in which the authenticatee proves that it has solved the provided problem [11].

A. The B-CRAM Architecture

Recall that the B-CRAM architecture consists of four entities: users, application servers, authentication servers, and the TTP. A user u stores her credentials with the TTP in the form of a tuple $(uid, \sigma'_u, \{\sigma_u\}_{pwd})$. All communication between the TTP and the user proceeds through an authentication server; that is, application servers are BFT2F clients who track a fork* consistent state of the TTP. The TTP is distributed $3f+1$ ways, providing safety and liveness with up to f failures and fork* consistency with up to $2f$ failures. Authentication servers may be distributed an arbitrarily large number of ways, allowing for scalability.

B. Credential Management

We provide a concise API for efficient credential management. There are three methods for users: creating a credential (`sign-up`), requesting authentication (`sign-in`), and updating a credential (`update-cred`), as shown in Table I.

TABLE I: Credential Management API

Operation	Arguments
`sign-up`	uid, σ'_u, $\{\sigma_u\}_{pwd}$
`sign-in`	uid, $token$
`update-cred`	$\{uid, \sigma'_{new_u}, \{\sigma_{new_u}\}_{pwd}\}_{\sigma_u^{-1}}$

Table II defines the messages referenced in each of the following subsections.

1) Signing Up: In order to register herself with the TTP, a user simply picks a uid and generates a public key σ'_u and a private key σ_u generated with, say, OpenSSL [14]. After picking a password and encrypting the latter with it, she sends a `sign-up` message to an authentication server, who then wraps it in a `BFT2F-OP` message and multicasts it to every node in the TTP. The TTP replicates the operation using BFT2F, failing the operation if uid already exists in its store. The authentication server waits for the TTP to respond with $2f+1$ distinct matching `BFT2F-REPLY` messages. Once it collects these messages, it responds to the user with the appropriate error code (success or failure). The user can then independently verify whether the transaction really did succeed or fail by issuing a `sign-in` request.

2) Signing In: Signing in requires the user to participate in a challenge-response authentication procedure, sketched in Figure 1. In order to prove her identity, a user must declare some user-id, prove that some public-key is bound to that user-id, and finally prove that she owns her declared user-id.

The user begins by declaring herself to an application server to which she wishes to authenticate. The application server issues her a token for accounting and to guard against replay attacks, and challenges her to procure proof that she owns her declared user id. The user issues a `sign-in` request to an authentication server, who then multicasts it to the TTP. Upon collecting $2f+1$ matching `BFT2F-REPLY<sign-in>` messages corresponding to the requested operation, the authentication server sends a `SIGN-IN-REPLY` message to the user, which consists of her public key, her private key encrypted with her password, and a set S of $2f+1$ distinct tuples $(\sigma'_t, \{token + \sigma'_u\}_{\sigma_{t^{-1}}})$, where t is a TTP node. S constitutes proof that σ'_u is bound to the declared user-id. Finally, the user decrypts her password, signs her public key and sends a `SIGN-IN-PROOF` message to the application server, who looks up the user's token and verifies every signature in the message.

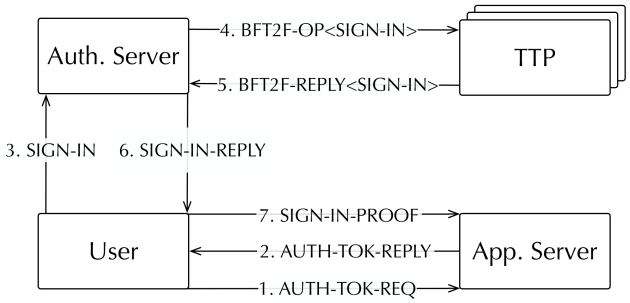

Fig. 1: B-CRAM `sign-in`. The TTP is a PKI key-value store comprised of $3f+1$ nodes running BFT2F. Here, we show an example of a user authenticating herself to an application. Steps 1 and 2 constitute a token-exchange and initiate a challenge – the user must procure proof that a some public key is bound to her. In step 6, the authentication server returns to the user her public and private key pair, the latter encrypted with her password, and a set of $2f+1$ RSA signatures; this step is implicitly a challenge, since the user cannot correctly sign the public key unless she can recover the private key. The user then signs her public key and ships it and the signatures to the application server in step 7, completing the authentication.

3) Updating Credentials: To change her credentials, a user sends an `update-cred` request to an authentication server. This request contains her new set of credentials, signed with her current private key: $\{uid, \sigma'_{new_u}, \{\sigma_{new_u}\}_{pwd}\}_{\sigma_u^{-1}}$, where pwd is usually a new password. The authentication server multicasts the operation to the TTP. If a TTP node determines that the supplied signature does not match that what would be produced by the private key bound to uid, it fails the operation; otherwise, the operation succeeds. In either case, TTP nodes respond to the authentication server, who sends a `SIGN-IN-REPLY` to the user after collecting $2f+1$ matching `BFT2F-REPLY<update-cred>` messages.

In the `update-cred` operation, the TTP implicitly challenges the user to provide it with a valid signature. If it cannot produce such a signature, it fails the challenge-response – this prevents adversaries from maliciously

Direction	Message	Contents	Direction	Message	Contents
c ∈ AS → TTP	`BFT2F-REQ<op>`	op, ts, c, V	c ∈ AS → user u	`SIGN-UP-REPLY`	$errno, uid, \sigma'_u, \{\sigma_u\}_{pwd}$
t ∈ TTP → c ∈ AS	`BFT2F-REPLY<res>`	$c, ts, res, \{t, view, n, HCD^n\}_{\sigma_t^{-1}}$	c ∈ AS → user u	`UPDATE-CRED-REPLY`	$errno, uid, \sigma'_{new_u}, \sigma_{new_u}$
	res : `sign-up`	$\{errno, uid, \sigma'_u, \{\sigma_u\}_{pwd}\}_{\sigma_t^{-1}}$	c ∈ AS → user u	`SIGN-IN-REPLY`	$errno, uid, \sigma'_u, \{\sigma_u\}_{pwd}, S$
	res : `sign-in`	$errno, uid, \sigma'_u, \{\sigma_u\}_{pwd}$, $\{token + \sigma'_u\}_{\sigma_t^{-1}}, \sigma'_t$	user u → app. server	`AUTH-TOK-REQ`	uid
			app. server → user u	`AUTH-TOK-REPLY`	$token$
	res : `update-cred`	$\{errno, uid, \sigma'_{new_u}, \{\sigma_{new_u}\}_{pwd}\}_{\sigma_t^{-1}}$	user u → app. server	`SIGN-IN-PROOF`	uid, $\{\sigma'_u\}_{\sigma_u^{-1}}$, S

TABLE II: Message Definitions. Listed here are all the messages sent between parties in B-CRAM. Operations possible in a BFT2F-REQ map 1:1 to the operations in table I. ts stands for timestamp, V for authentication server c's current version, $errno$ for a return status message, and S for a set of $2f + 1$ distinct tuples $(\sigma'_t, token + \sigma'_{u\sigma_t})$, where t is a TTP node.

changing other users' credentials.

C. Forked States

Since the TTP is replicated using BFT2F, the state maintained by B-CRAM might fork. Forks correspond directly to our credential management API:

1) *User Existence*
   A `sign-up` might be processed by one fork but not another; as such, the TTP could tell some clients that a user exists while informing others that she does not.
2) *Audit Trail*
   Accounting information logged during a `sign-in` (i.e. the log about the circumstances of a user log-in) might differ between two or more branches.
3) *User Credentials*
   Nodes in different fork sets might map different credentials to the same user.

1) Implications for Use Cases: The implications of these forks depend upon the context in which B-CRAM is used. For example, imagine a setting in which B-CRAM was used to provide authentication in the style of OpenID or WebAuth. Forking on user existence and audit trails, while inconvenient for the user, would not cause significant damage alone; however, a fork on user credentials might. In particular, if some user changes her credentials because her password was compromised, malicious TTP nodes could fork the system and prevent the change from propagating throughout the entire system – thus, the adversary who stole her password might retain access to her account across all applications.

We could contain the damage incurred by a user credentials fork if there was some mapping from application servers to authentication servers. Say that each authentication server only served some subset of applications, and then say that malicious TTP nodes caused a a user credentials fork. Unlike the previous example, in this case, our adversary would only retain access to the user's account across a subset of applications; in particular, across those applications which mapped to authentication servers tracking forks in which the credential change was not processed. This might mean that the adversary might be incorrectly authenticated to, say, Facebook, but she won't be granted entry to Amazon.

A mapping from applications to authentication servers could be achieved by partitioning the space of domain names with consistent hashing; if an authentications server received a request for a domain name that it did not serve, it would simply forward it to the correct authentication server. Increasing the number of authentication servers would in theory, then, minimize the risks associated with a user credentials fork. In practice, however, another problem arises: If just one authentication server were compromised by an adversary, then that adversary could route all requests through that authentication server and bypass the keyspace partitioning.

Perhaps the ideal use case for B-CRAM, then, might be one in which each application server was bound to a unique authentication server. This is, in essence, the card-swipe example mentioned in [10]. But the use case extends beyond card-swipe authentication. B-CRAM could be particularly nice for logging into computers at a physical site – e.g., a governmental facility where security was particularly important. Each computer would function both as an authentication server and as an "application server" – in order to log-in, a user would enter his user id and the computer would form the appropriate credential management operation, wrap it in a `BFT2F-REQ`, and forward it to the TTP.

2) Detecting Forks: We enable clients to detect forks by slightly modifying BFT2F. In BFT2F, a node t ignores a request from a client c if the HCD in c's current version vector $cur(V)$ does not match the HCD t sent to c in its last reply [10]. We propose that rather than ignoring the request, t respond to the client with $HCD^{cur(V).n}$. There are two cases:

1) t has not yet committed $cur(V).n$. This could occur if t were in the same fork set as c but was simply lagging behind other nodes, or if t were in another another fork set. In either case, t responds with $HCD^{cur(V).n} = NULL$.
2) t has committed $cur(V).n$. This implies that t

is in a different fork set than the one serving c. Upon receiving t's response, c will see that $HCD^{cur(V).n} \neq cur(V).HCD$ – i.e., it will realize that the state is forked and will raise an alarm accordingly.

TTP nodes can independently check for forks by inspecting commit messages. When committing sequence n, a mismatch between a received HCD^n and a TTP node's expected HCD^n implies a fork.

D. CRAM-CERT on B-CRAM: An SMTP Extension

As a proof of concept, we implemented an original SMTP authentication protocol. Dubbed CRAM-CERT and backed by B-CRAM, our authentication protocol can be swapped out for CRAM-MD5. Based on the HMAC-MD5 algorithm, CRAM-MD5 provides secure authentication in that the authentication token changes for each authentication – eavesdropper cannot use a rainbow table[2] to find passwords [7].

CRAM-CERT opts to use the user's public key, signed by her private key, instead of a password, and RSA signatures instead of MD5 hashes. Then the SMTP server and the user use the public key to upgrade the TCP socket to a TLS socket via the STARTTLS protocol [5]. Algorithm 1 shows an example of CRAM-CERT in action, and Table III compares CRAM-CERT and CRAM-MD5.

Protocol	Token	Authentication Object	Verification
CRAM-MD5	nonce+domain	token+password	hash comparison
CRAM-CERT	nonce+domain	token+σ'_u	signature verification

TABLE III: CRAM-CERT vs. CRAM-MD5. CRAM-CERT, our SMTP authentication extension, uses B-CRAM to authenticate users and then upgrades their connection with STARTTLS.

S: 220 smtp.server.com Simple Mail Transfer Service Ready
C: EHLO client.example.com
S: 250-smtp.server.com Hello client.example.com
S: 250-SIZE 1000000
S: 250 AUTH CRAM-CERT
C: AUTH CRAM-MD5
S: 334
PDQxOTI5NDIzNzJAc291cmNl1ci5hbmRyZuY211LmVkdT4=
C: cmpzMyBlYzNhNTlmZWQ-
zOTVhYmExZWM2MzY3YzRmNGI0MWFjMA==
S: 235 2.7.0 Authentication successful
C: EHLO client.example.com
S: 250-smtp.server.com Hello client.example.com
S: 250-SIZE 1000000
S: 250 STARTTLS
C: STARTTLS
S: 220 Go ahead.

Algorithm 1: An example SMTP authentication using CRAM-CERT on B-CRAM.

V. Implementation

To the best of our knowledge, we offer the first open-source implementation of BFT2F[3]. We implemented the core algorithm and view changes; we have yet to implement checkpointing and garbage collection. The entire B-CRAM stack was implemented in Python, while CRAM-CERT was implemented as a Node.js extension to Haraka [4]. Google's Python Protocol Buffers were used for serialization, and Apache Thrift was used for RPCs made from the authentication servers to the TTP.

We list some takeways from our implementation:

- BFT2F makes it somewhat difficult for our authentication servers to issue multiple concurrent requests. Say that one request from a given authentication server commits at some number of nodes, and then that the same authentication server issues another request but without having seen the BFT2F replies for the first request. The nodes that committed the first request will reject the second one, since its $cur(V).HCD$ will not match the HCD that those nodes have in their replay caches. Since the vast majority of our operations are sign-ins, we sidestep this problem by implementing the read-only optimization mentioned in PBFT and BFT2F [2], [10].[4]
- The number of TTP nodes $(3f+1)$ must be known to the application servers, so that any application server can make sure that it receives at least $(2f + 1)$ valid signature. This number can be disseminated in various ways; a rendezvous point (e.g., a website), DNS entries, etc..
- B-CRAM doesn't expire signatures, but application servers are free to choose their own policies regarding token expiration.

VI. Measurements and Evaluation

In evaluating the performance of our protocol, we used Mininet to simulate a B-CRAM topology consisting of seven TTP nodes, one authentication server, one application server, and one user server; 10 milliseconds of delay was added to each link. Our simulation ran on a single m3.2xlarge Amazon ec2 instance, equipped with eight virtual CPUs and 30 GiB of RAM, and we conducted an experiment to measure latency. Our experiment began

[2]A rainbow hash table is a pre-computed structure that allows attackers to recover passwords from hash values.

[3]The source code is available at https://github.com/akshayka/bft2f.

[4]The read-only optimization forfeits the ability to maintain an agreed-upon log of sign-in metadata; administrators may disable it if they value such accounting information over efficiency.

by signing-up 20 users. It then issued 200 `sign-in` requests, mapped over 20 processes. We averaged 52.6 milliseconds per end-to-end authentication (from step 1 to step 7 in Figure 1), and 580 milliseconds per sign-up. The read-only optimization accounts for the discrepancy between these two numbers.

Our numbers should only improve with more hardware and authentication servers, especially considering that each of our eight virtual CPUs peaked at 100 percent utilization during our experiment. Nonetheless, an average of 52.6 milliseconds per sign-in scales to 1140 sign-ins per minute, likely enough to serve a small population of users (e.g., a small college campus) and certainly enough for the on-site computer access use case mentioned previously.

VII. Discussion

When it comes to fault-tolerance and security, B-CRAM has many advantages over its peers. By replicating the TTP with BFT2F, we not only survive byzantine failures but also leave an audit trail and introduce a mechanism to identify malicious nodes. By taking a PKI approach, we circumvent the problem of leaked passwords. That said, while the TTP will not leak passwords, B-CRAM remains vulnerable to dictionary attacks. Any user can fetch the password-encrypted private keys of all other users in the system. As such, we recommend both choose sufficiently strong passwords and change their credentials periodically.

For the sake of robustness, we do sacrifice efficiency. Public key cryptography is notorious for its expensive computation [1]. Additionally, B-CRAM's use of multicast in BFT2F makes the network latency among TTP nodes relevant for service latency. B-CRAM's robustness also makes it a bit more unwieldy than other authentication protocols, since the user is non-trivially involved in the process. Nonetheless, we could still adapt the protocol for an OpenID or WebAuth-like purpose by, say, implementing a browser extension for users. Finally, because of the TTP, B-CRAM cannot be used for P2P authentication, unlike [15].

VIII. Future Work

Given that BFT2F maintains a significant amount of state, implementing checkpointing and garbage collection is high on our list of priorities. Additionally, it would behoove us to right a rigorous test-suite to that verified our implementation, and to perform more robust experiments in a non-simulated environment.

IX. Conclusion

Existing TTP authentication protocols suffer from both availability problems and security vulnerabilities. We present a B-CRAM, a more robust TTP authentication protocol that uses redundancy, public key cryptography, and BFT2F replication to address both of these concerns. We also tweak BFT2F to allow for automatic detection of forked state by both clients and nodes. We demonstrated the protocol's viability by introducing CRAM-CERT; preliminary measurements suggest that our particular implementation B-CRAM could be deployed as is for modestly sized populations.

References

[1] A Cost-Based Security Analysis of Symmetric and Asymmetric Key Lengths. *EMC*. N.p., n.d. Web. 10 Dec. 2014.

[2] Castro, M., and Liskov, B., Practical Byzantine fault tolerance. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*, pages 173-186, New Orleans, LA, February 1999.

[3] Haller, N. and Atkinson, R., *On Internet Authentication*, RFC 1704, October 1994.

[4] Haraka Manual. Haraka. N.p., n.d. Web. 10 Dec. 2014.

[5] Hoffman, P., *SMTP Service Extension for Secure SMTP over TLS*, RFC 2487, January 1999.

[6] Hardt, D., Ed., *The OAuth 2.0 Authorization Framework*, RFC 6749, October 2012.

[7] Klensin, J., Catoe, R., and Krumviede, P., *IMAP/POP AUTHorize Extension for Simple Challenge/Response*, RFC 2195, September 1997.

[8] Lear, E., Tschofenig, H., Mauldin, H., and Josefsson, S., *A Simple Authentication and Security Layer (SASL) and Generic Security Service Application Program Interface (GSS-API) Mechanism for OpenID*, RFC 6616, May 2012.

[9] Levi, A. and Caglayan, M., *The problem of trusted third party in authentication and digital signature protocols*, Antalya, Turkiye, 1997.

[10] Li, J. and Mazieres, D., Beyond one-third faulty replicas in Byzantine fault tolerant systems. In *NSDI* 07

[11] M'Raihi, D., J. Rydell, S. Bajaj, S. Machani, and Naccache, D., *OCRA: OATH Challenge-Response Algorithm,* RFC 6287, June 2011. Web. 10 Dec. 2014.

[12] Mazieres, D. and Shasha, D.. Building secure file systems ' out of Byzantine storage. In *Proceedings of the 21st Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 108117, July 2002. The full version is available as NYU computer science department technical report TR2002-826, May 2002

[13] Miller, S. P., Neuman, B. C., Schiller, J. I. and Salzer, J. H. *Kerberos Authentication and Authorization System*byMIT Project Athena, 1988.

[14] OpenSSL Overview. OpenSSLWiki. Web. 10 Dec. 2014.

[15] Pathak, V. and Iftode, L., Byzantine fault tolerant publickey authentication in peer-to-peer systems. Computer Networks. Special Issue on *Management in Peer-to-Peer Systems: Trust, Reputation and Security*, 50(4):579596, March 2006.

[16] R. Schemers, and Allbery, R.. *WebAuth Technical Specification*. Stanford University, 23 July 2014. Web. 05 Dec. 2014.

[17] Sirbu, M. A., and Chuang, J., C.-I. 1997. Distributed authentication in Kerberos using public key cryptography. In *Proceedings of Symposium on Network and Distributed System Security* (San Diego, Calif., Feb.), IEEE Computer Society Press, Los Alamitos, Calif.