S3B: Software-defined Secure Server Bindings

William Koch
Department of Computer Science
Boston University
Boston, MA 02215
wfkoch@bu.edu

Azer Bestavros
Department of Computer Science
Boston University
Boston, MA 02215
best@bu.edu

***Abstract*—For decades, request-routing protocols operating at multiple layers of the network stack have been a staple of Internet services. Commonly deployed request-routing techniques use the requestor's IP address as an identifier of the client. For instance, using DNS as a request-routing protocol, the local DNS resolver's IP address is used as a surrogate identifier of the client in order to assign the client to the closest server. While such coarse associations may be acceptable for performance-centric purposes, they are not appropriate in settings that require fine-grained, enforceable bindings of clients to servers — e.g., to ensure that malicious clients are unable to bypass their bindings and issue their request to a server of their choosing.**

In this paper, we propose S3B (Software-defined Secure Server Bindings), a protocol that provides precise and enforceable client-server assignments. S3B uses a server module to assign clients unique access keys. Using HTTP redirection with the key encrypted as an additional domain label, the name server is able to distribute precise server assignments specific to each client. In addition, the server module maintains an access control list to enforce these assignments. As an implementation of the S3B protocol, we have developed an HTTP/S prototype and deployed it to Amazon AWS. Our performance evaluation suggests that our prototype introduces no discernible overhead for client requests. To evaluate S3B's effectiveness as a security appliance, we developed an application to isolate clients suspected as spiders, capable of virtually immediate containment once detected.

I. Introduction

For decades, request-routing protocols operating at multiple layers of the network stack have been a staple of Internet services and web applications (webapps). By managing how client requests are "routed" (or assigned) to servers, these protocols afford network operators and/or service providers a level of indirection that can be used for a multitude of control and management purposes, including the enablement of advanced localization, load balancing, and caching mechanisms. While request-routing protocols can be implemented at the application layer (*e.g.*, through redirection), the most common and widely-used protocols are extensions of the Domain Name System (DNS) as a directory service.

The efficiency of control and management mechanisms that leverage request-routing protocols depends largely on the ability of these protocols to associate a service request with the client that originated the request. This has been a key limitation of DNS-based protocols, which are unable to identify accurately the client that originated the request since DNS queries are typically made to the name server on behalf of the client by a local recursive resolver. For example, when DNS is used for request-routing in support of load balancing, rather than assigning requests to servers using DNS' default round-robin approach, the local DNS resolver's IP address is used as a surrogate identifier of the client to assign clients to close servers [1], under the assumption that the local DNS resolver is close to the client. This assumption is seldom accurate [2]. Recent extensions to DNS have proposed including a prefix of the clients IP address to be forwarded to the name server to increase the accuracy of the assignment [2].

Even if the IP address attached to a request for service is known, the use of client-side gateways and proxies often result in the inability of a service to distinguish between a potentially very large number of clients. As a result, mechanisms that use IP addresses as identifiers are unable to provide fine controls for applications that require them, *e.g.*, security-related access control applications. As a result, in practice, webapps tend to block entire IP address ranges (*e.g.*, TOR exit nodes) because innocent users cannot be distinguished from malicious ones. Up to 3.7% of the top 1,000 Alexa sites block TOR users, thus legitimate clients fall victim to this practice [3].

In contrast to DNS-based request-routing, HTTP-based request-routing has far greater knowledge of the client and is capable of providing more accurate client-server assignments. HTTP-based request-routing initially uses DNS to assign the client to a virtual surrogate. This additional layer of indirection allows the virtual surrogate to perform deep inspection of the clients request to decide how it should be routed. For example, session affinity (otherwise known as sticky sessions) [4] allows for a load balancer to assign a client to a specific upstream server based on an identifier stored in an HTTP cookie. All future client requests are proxied through the load balancer to the assigned server. Although session affinity can provide a mechanism to bind a specific client to an upstream origin web server, it cannot bind the client to the load balancer assigned by DNS. Due to DNS and HTTP operating independently, the load balancer will blindly accept requests, whether the assignment originated from the name server or not.

Request-routing protocols do not provide the means for enforcing desirable client-server assignments, thus allowing clients to bypass their assignment and access a server of their choosing directly. Even cloud security services attempting to provide access control upstream have not been successful as they have been circumvented by attackers [5]. These cloud services act as reverse proxies for the origin web servers. Although the web server can enforce the origin of the traffic by white-listing the reverse proxies, this is only practical if the IP addresses are static.

In this paper we introduce S3B, a protocol to create precise

and enforceable client-server assignments to solve these aforementioned shortcomings. This is done by assigning each client a unique access key (AK). For a given programmable function, the AK is mapped to the IP address of a web server. To be compatible with the existing name resolution system that DNS provides, the authoritative name server ultimately provides the interface for the client to receive their server assignment.

In order for the name server to learn the AK, we use DNS object encoding [6]. While traditionally used to encode the requested object, we use it to encode client information. First the client is directed to a virtual surrogate running S3B server software (S3Bware) to form a token. The token's payload is an encryption of the client's AK and IP address. The token is appended as an additional domain label and S3Bware issues a redirect to this location. In response to the redirect, a DNS query is performed, transferring the client information from the virtual surrogate to the name server.

We adopt a hierarchical architecture used by software defined technologies (*i.e.*, software-defined everything (SDx)) to allow these assignments to be programmed while also decoupling the management, application, control and data planes. As such, S3B uses a controller with a global view of the system to enable management of the client-server assignment logic. An application makes its decision based on the client AK, IP address, the set of servers online, and other environment conditions (*e.g.*, performance metrics and security threats).

To enforce the assignments, each server runs S3Bware which maintains an access control list (ACL). S3Bware is positioned between the web server and the upstream webapp, allowing requests and responses to be modified, while remaining transparent to the application layer. The S3Bware's ACL is updated remotely by the controller, in effect creating a dynamic application firewall.

In addition to being capable of fulfilling traditional QoS goals, precise and enforceable client-server assignments provided by S3B allow for various security related applications, including but not limited to:

- **Dynamic deception** - Assign malicious clients to honeypot servers.
- **Application Layer (L7) DDoS defense** - Reassign bots to dummy servers.
- **Moving Target Defenses (MTD)** - Continuously modifying client-server assignments to resist reconnaissance.

S3B provides stakeholders of a service (*e.g.*, a website owner) a tool to have a greater degree of control over their clients experience, whether for performance or security reasons. We have developed a prototype service that instantiates S3B and have found client overhead to be negligible. That said, a side effect of unique client domains is an increased storage overhead for the intermediary recursive DNS resolver: Rather than caching a single domain, the resolver must cache a record for every active client it serves. The impact of this added book-keeping depends on the scale of the service; it should be negligible except for services with massive number of users, which are not the types of services that would implement granular control of client-server assignments in this manner due to the additional infrastructure at their disposal.

To demonstrate its use as a security appliance we have created an S3B application, `SpiderTrap` to isolate web spiders to a honeypot server. We envision S3B to be deployed as a cloud service, such that the S3B name server and controller is maintained by the cloud service provider, while allowing the customer to install applications to define how clients should be assigned to servers (similar to Amazon lambda functions [7]). To summarize, this paper makes the following contributions:

- We propose a new protocol, S3B (Software-defined Secure Server Bindings) to improve the security of client-server assignments distributed through DNS providing two key capabilities,
  - Precise, software-defined client-server assignments,
  - Server-side enforcement of the client-server assignment.
- We develop an HTTP/S prototype implementation of the S3B protocol with no discernible client side overhead.
- We implement an S3B security application, `SpiderTrap`, capable of virtually immediate isolation of clients suspected as spiders to a honeypot server.

The remainder of this paper is organized as follows. In Section II we provide a threat model summarizing the adversary. In Section III we provide an overview of the S3B protocol, walking through a real-world example of isolating a web spider to a honeypot server. In Section IV we discuss the details of each S3B component used in the example. We describe the implementation of S3B in Section V and it is evaluated in Section VI. Related work is reviewed in Section VII and we conclude with future work and the conclusion in Section VIII.

II. Threat Model

In this work we consider threats that require a large number of requests to be made in order to be successful (*e.g.* data harvesting, brute force attacks, application layer DDoS). Due to the sheer number of operations required for the attack to succeed, the attacker must rely on software to automate their tasks (*e.g.* a bot). At a minimum the attacker has the following capabilities: (1) they are associated with a benign IP address, and if behind a NAT sharing a public IP address they have the ability to sniff the local network, (2) they are able to forge HTTP requests to appear as if they are originating from a legitimate web browser and (3) they have the same capabilities as a web browser such as storing cookies and following redirects. However we also assume there exists a class of challenges that can be solved by a human but not the attacker (*e.g.* reverse Turing test, human interactive proof (HIP), etc.). These bot prevention mechanisms have become an important tool for any web developer. Nevertheless advances in deep learning has made developing effective challenges difficult allowing researchers to develop attacks for both CAPTCHA [8] and reCAPTCHA [9]. Furthermore demand for cracking CAPTCHAs has fueled an underground ecosystem using human labor and automated solving techniques to solve challenges [10]. While this has evolved into a cat and mouse game, in the end the attacker is operating within a limited budget, time and resources that they can allocate to solving challenges.

S3B is a tool and therefore the tool is only as effective as its user. In this case the user (or defender) expresses their

security requirements and the attacks they wish to defend against in the form of S3B applications IV-B. The defender is capable of detecting a threat through various network sensors installed on its servers and adjusting its security policy on-demand depending on the current threats in order to balance usability and security. In particular the defender can escalate the sophistication of the deployed bot prevention mechanism when threats are detected or suspected [10].

III. Overview

S3B provides a mechanism to assign and enforce a client to specific web server. This provides a foundation for many security related applications such as client isolation. To provide an overview of S3B and its basic functionality, we walk through an example web server architecture using S3B installed with an app to isolate web spiders to a honeypot server. An illustration of the messages passed between the involved parties is displayed in Figure 1.

A spider (while not always malicious) is a common tool used by attackers for a variety of purposes. For example, a spider can be used to scrape or mirror websites, harvest personal identifiable information (*e.g.*, e-mails) and perform reconnaissance by mapping the webapp [11]. A honeypot server is a decoy server used to gather intelligence from a specific threat [12]. Rather than simply blocking the session of the spider, we isolate the spider to a honeypot server consisting of dummy data to monitor the spider's behavior and absorb its computational resources so that it is not used elsewhere. In this example we assume the spider performs headless crawling (*i.e.*, interacts with underlying HTML code, not a browser GUI). This type of spider can be detected when a resource to a honeypot link (a hidden decoy link not seen by clients using the browser) is requested. Spider isolation occurs in two stages. First the spider registers with S3B as a new client to obtain a unique AK, just as any other client would. Once S3B detects a honeypot link has been requested, the second stage updates the spider's server assignments to the honeypot server. In this example, the web architecture using S3B is comprised of a name server, two webapp servers, a honeypot server and a controller.

Client Registration New clients unknown to the system must undergo a one-time registration process to obtain their unique AK which will be used by S3B to map the client to a server (Steps 1-12). The registration process uses a combination of DNS dispatching, DNS object encoding, and HTTP redirection. The spider begins by unknowingly making a request for a honeypot link `foo.com/honey` which initiates a DNS query for `foo.com`. Honeypot links in actual deployment consist of links commonly searched by vulnerability scanners and Google dorks [13]. The name server is a proxy cache, it does not contain any assignment logic. Since the spiders request is a cache miss, the request is sent to controller to be resolved. Without an AK encoded in the DNS query, the client is unknown to the controller. To initialize the registration process a S3B registration app implementing a round robin algorithm responds with the IP address of a virtual surrogate to register the client.

Once the spider receives the DNS answer it sends its HTTP request to the web server. Each web server in S3B is installed with S3Bware, software to register and enforce assignments. Registration is the processes of assigning the client to a unique AK which will be used as the identifier to map the client to a web server. It is implemented to meet the web site's security requirements and risk of attack. It may be a transparent process as described in this example to deter *script kiddies* using off-the-shelf automated software or it may be an interactive challenge for more security conscious applications in order to limit a sophisticated automated adversary from harvesting and stock piling AKs prior to an attack. For example it may require the user to have an account with the web application, submit a code received via text message, or solve a CAPTCHA [14].

Registered clients have the AK stored as an HTTP secure cookie and are included in every request. S3Bware enforces the mappings by maintaining an access control list (ACL). The spiders request to the virtual surrogate, absent this cookie, initializes the registration process. S3Bware generates a random AK for the client. The AK along with the clients IP address is encrypted, to form the payload for token `t1`. Details on how tokens are generated for users having the same IP address are discussed in Section IV-D3. The token is set as an additional domain label creating a new domain `t1.foo.com`. S3Bware responds with a temporary redirect (307) with the location HTTP header set to the new domain and the AK set as a secure cookie. Every future request made by the spider includes the AK in the cookie (omitted from Figure 1).

The redirect initiates a second DNS query. The purpose of the token is to allow client information to be passed from the web server to the name server which the name server cannot obtain directly due to the architecture of DNS. The token in the DNS name indicates to the name server this is an existing client and it has already been registered. The name server first checks its cache, since there is a miss, the request is forwarded to the controller. A second S3B app resolves the server selection for registered clients. However, before returning the result to the name server, the controller messages the assigned server to update their ACL, specifying the client matching the IP address and AK tuple (q.a.z.w, X) should be allowed access to the webapp. Upon receiving the DNS answer, the spider has completed the registration process and is now uniquely assigned to a server.

Client Reassignment S3B provides the ability to modify existing assignments (Steps 13-26) which we use in this example to relocate a spider to a honeypot server. Now assigned to a web server, the spider makes its web request. However the spider unknowingly makes a request for a honeypot link. S3Bware (containing a plugin to support the S3B spider isolation app) detects the honeypot link has been requested and sends a security alert to the controller specifying the IP address and AK of the misbehaving client. The spider isolation app installed on the controller processes the security alert, marks the AK as a spider and responds to the webapp server to update its ACL to redirect clients with the matching AK. As the spider continues to crawl the next link on the web page, the spiders AK is not longer set to 'Allow' in the S3Bware's ACL. S3Bware reacts by issuing a redirect with a new token $t2$ (yet still containing the same IP address and AK) to force the controller to re-assign the client. Because the AK has been marked as a spider, the DNS request received by the controlled is processed by the spider isolation app and assigns

the spider to the honeypot. All future requests by the spider are subsequently made to the honey pot server.

IV. Details

The S3B architecture is inspired by software defined technologies and botnets for their dynamic and flexible characteristics. It's hierarchical design, illustrated in Figure 2, decouples the data and control plane while providing abstraction for applications to create and enforce client-server assignments. The controller has a global view and control of the system through S3Bware installed on each server, essentially creating a botnet. Scalability and resilience of this architecture shares many similarities to SDN [15] such as using multiple controllers to decrease single point of failures. We plan to address these topics further in future work. In this section we describe in detail each plane using a top-down approach.

A. Management Plane

The management plane provides a central interface for administrators to interact with S3B. This interface gives the administrators the ability to install, remove, modify and interact with S3B apps. The implementation of this interface may exist simply as terminal access to the controller or through a web interface.

B. Application Plane

The primary function of S3B apps is to define the logic for client-server assignments. Assignments are determined based on a number of factors received as feedback from the system (*e.g.*, server performance, financial constraints, security threats, etc.). Apps are installed to the controller as middleware. Each controller app is able to register a handler to a number of callbacks issued by the controller. Handlers are chained together, specified by a priority index, allowing multiple apps to register to the same callback. Handlers with a higher priority may modify the response of handlers with lower priority. In addition to providing the client-server assignments, apps may be used to gather metrics, log events and modify the running environment. For example in a cloud settings apps could add and/or remove servers in response to client-server assignments and other environmental changes.

C. Control Plane

The control plane has a global view of S3B. It consists of one or more controllers that administer the actions defined by the apps. The controller is configured with a public-private key pair (pk_{ctrl}, sk_{ctrl}) used to authenticate messages. The controller implements a callback framework to route messages it receives to the appropriate app. Messages are received over an encrypted secure channel and authenticated with a cryptographic signature. Each message is identified by the sources public key (pk_{src}), if verified the message is mapped to its corresponding callback and the registered handlers for the callback are executed. Table I provides a summary of the callbacks.

Source	Callback	Args	Return
NS	`UnknownClient`	None	DNS Answer
NS	`ExistingClient`	IP, AK	DNS Answer
NS, S3Bware	`Join`	Profile	Config
NS, S3Bware, Firewall/IDS	`SecurityAlert`	Alert	Action
NS, S3Bware	`HealthAlert`	Alert	None

TABLE I: Controller Callbacks

Unknown and Existing Clients The controller receives DNS messages from the name server to resolve the client-server assignment. Existing clients are identified by having a token in the domain name, where as this is absent for unknown clients. If a token is present, the controller decrypts the payload using a symmetric key k (also known to S3Bware) to obtain the client's IP address and AK before the callback handlers are called. This process can be offloaded upstream to the name server if it is trusted (Section IV-D2). Once the last app in the callback chain has been called, its DNS answer is sent back to the name server.

Join S3B servers are allowed to join and leave dynamically to support automated scaling. To keep the S3B apps up-to-date of the available servers and initialize S3Bware, the server must send a join message to the controller. The join messages include a profile specifying the server's configuration and capabilities. Profiles are used by apps for help determining the client-server assignment. The join callback returns configuration data to be used by the S3Bware. This includes information such as its assigned role, and parameters for token generation. Because a server may die unexpectedly, S3Bware does not send leave messages. Instead it is up to the apps to actively probe the servers or monitor the cloud environment.

Alerts To complete the assignment decision loop, S3B servers are able to send feedback to the controller. Health alerts allow S3B server to send performance metrics such as resource consumption. Additionally servers are able to send specific security alerts related to misbehaving clients, or detected attacks (*e.g.*, a honeypot link has been requested). Security alerts may be sent on behalf of the S3B by security appliances (*e.g.*, application firewall, intrusion detection system (IDS)). These alerts are essential for establishing optimal client-server assignments.

D. Data Plane

The data plane consists of the S3B servers interfacing directly with the client. In this section we describe the details of each party member.

1) Client: In order for a client to access a website using S3B their browser should allow cookies for optimal performance. Cookies store the client's registration information (*i.e.*, IP address at time of registration, and AK) locally within the clients browser, preventing the client from having to register every time they visit a new page which could be an expensive task.

2) Name Server: The name server is the one who ultimately interfaces with the client to provide them with their server assignment. To authenticate messages sent to the controller the name server is configured with a public-private key pair (pk_{ns}, sk_{ns}). Normally DNS records resolved by the controller are cached by the name server until the records time to live (TTL) expires. To remove the cached record before this time, the name server exposes the `RevokeRecord` command to the controller (Table II).

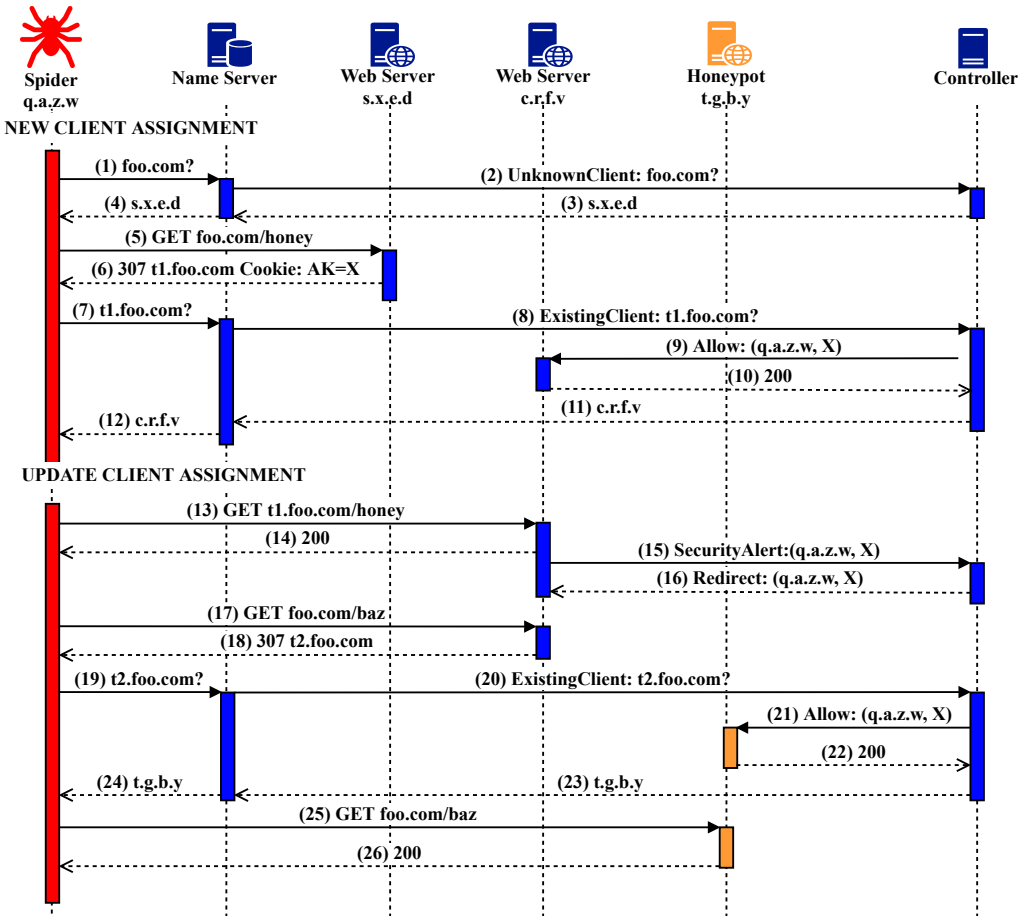


Fig. 1: S3B example demonstrating client registration and re-assignment for a spider isolation application.

Component	Command	Arguments	Return
S3Bware	UpdateConfig	Config	OK or error
	ACAddRule	Rule	OK or error
	ACRemoveRule	Rule	OK or error
Name Server	RevokeRecord	token	OK or error

TABLE II: S3B Data Plane API

Similar to proxies providing multiple modes for TLS [16], S3B offers two modes for handling tokens: passthrough and offloading. In passthrough mode the name server does not have knowledge of the cryptographic key k used to decrypt tokens. This mode is useful in settings in which the name server is not fully trusted (*i.e.*, hosted by a third-party). On the contrary, offloading mode trusts the name server with the key k, to offload computation required for authentication, integrity and decryption of the token. Invalid requests are rejected before passed to the controller.

3) S3Bware: The majority of the heavy lifting is performed by S3Bware. It in located in between the web server and webapp in the software stack. This can be implemented as a server module, middleware, or proxy just as long as it is in-path and able to intercept and modify web traffic. Like the name server it is configured with its own public-private key pair for authenticated messages to the controller (pk_{svr}, sk_{svr}). Additionally it also contains the symmetric key k for token encryption.

To remain synchronized with the controller, when S3Bware is started, it sends a `Join` message to the controller including its public key pk_{svr} and receives its configuration in response. The S3Bware's API exposes an `UpdateConfig` command (Table II) if the configuration changes in the future. S3Bware is primarily responsible for registering new clients and enforcing server assignments made by the controller. If assigned as a registrar, S3Bware will generate AK's and create tokens for clients. Next we will describe the token creation process.

Token Creation A S3B token is an encapsulation of a clients IP address and AK for transport between HTTP and DNS. The token is appended as a domain label to the applications domain, thus binding the AK between the name server and web server. The AK allows the client possessing it access to the server it has been assigned. It must remain secret and protected the same as a session token. Because UDP DNS queries are sent unencrypted [1], the token must protect the AK

[1]Although DNS queries can be sent over TCP with TLS this is hardly deployed.

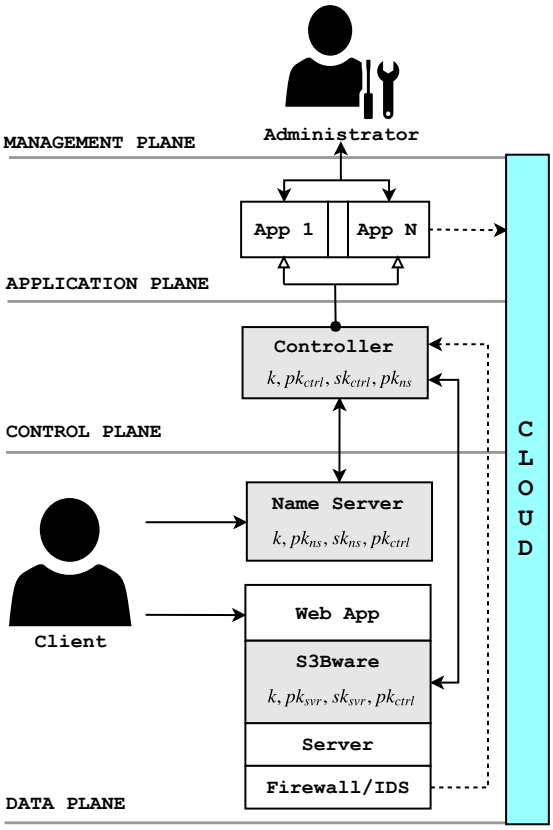


Fig. 2: Overview of S3B. S3B components are shaded gray. Direction of arrow indicates who initiates message exchange. Dashed lines are optional messages.

from adversaries monitoring the network.

There is a size constraint of the token since it must be transported via DNS. Each domain label can have a max of 63 characters such that the first character is a letter, the last character is a letter or digit, and the interior characters may be letters, digits or hyphens [17]. Practically speaking, a token should ideally fit in a single domain label to take advantage of wildcard SSL/TLS certificates. The token is computed by the following equations,

$$T, c = AE_{k,IV}(IP||AK) \tag{1}$$

$$\begin{aligned} \text{token} &= IV||T||c \\ &= \text{S3BwareID}||\text{b36}(\text{S3BwareC}||T||c) \end{aligned} \tag{2}$$

The concatenation of the clients IP address and AK is encrypted with an authenticated encryption (AE) scheme using a symmetric key k and an initialization vector IV to produce an authentication tag T and ciphertext c. AE provides both authentication and integrity of the token. The AE scheme must allow for unique (but not necessarily random) IVs and should be length preserving to minimize its size, for example AES in Galois/Counter Mode (GCM) [18]. The AK is randomly generated and provides enough bits of entropy to resist brute-force attacks [19]. To minimize the length of the IV and support a distributed environment a deterministic IV construction is used [18]. In this construction, the IV is formed by the concatenation of two fields: a fixed field and an invocation field. The fixed field is a unique identifier for each S3Bware (S3BwareID) generating the IV and the invocation field is an incremented counter for each invocation of the encryption function for that particular S3Bware (S3BwareC). To abide by the domain name label naming convention, the fixed field must begin with a letter.

A token supports both IPv4 and IPv6 addresses however the lengths of the AK, IV and tag must be adjusted for the increased IPv6 length. This must be done in a way to not sacrifice security of the token for example by decreasing the size of the IV's invocation field. However by decreasing the size of the invocation field, the key k must be rotated more frequently because the IV can never be used more than once for a given key or its security is compromised. The counter S3BwareC, and the encryption output are in bytes. To be formatted for a domain label these values are concatenated together and base 36 encoded (i.e. lower-case letters and numbers). The S3BwareID is then appended to the front of base 36 encoding to form the token. In Section V we discuss specific instances of the token format.

Assignment Enforcement S3Bware acts as a dynamic firewall, remote controlled by the controller. Each S3Bware maintains an ACL and each rule in the ACL consists of a target, an AK, and an IP address. The target defines what will happen with the request and must be one of ACCEPT, REDIRECT, BLOCK or CHALLENGE. An ACL rule with an (AK, IP) pair is mutually exclusive and cannot exist in more than one list. The ACCEPT target will accept a request with the given (AK, IP) pair. The BLOCK target explicitly blocks an (AK, IP) pair or alternatively just by the IP address while the CHALLENGE target will prompt the user to solve a challenge (*i.e.* to log into the application or solve a CAPTCHA). The REDIRECT target blocks the request, generates a new token for the client and issues a redirect. The controller populates the ACL by the API summarized in Table II.

Algorithm 1 describes the access control process when receiving a client request. First an attempt is made to extract the AK, IP address and a signature (σ) from the cookies. If these cookies are absent the clients source IP address is checked if in the BLOCK or CHALLENGE ACL as a way to control abuse and limit the churn rate. Otherwise the client is registered. For requests including an AK and IP address their integrity is verified with the supplied signature using the symmetric key k. If the signature does not verify a security alert is sent to the controller with the clients IP address and the default response policy is returned. The default policy is specified in S3Bware's configuration returned when the S3Bware joins S3B. It may be to either block the request or redirect.

If the AK and IP address are valid, the BLOCK and CHALLENGE ACL are first checked for the specific pair. If the pair is in the ALLOW ACL the request is accepted and sent upstream. If the pair has yet to match, the clients AK is checked for any matching IP address pair (as specified by the wildcard "*") in the ALLOW ACL. An AK, IP address mismatch could indicate a mobile client who switched networks from the time its original application server assignment was made. In this situation, the client is redirected using a token recomputed using the current source IP address and original AK. Additionally if the client exists in the REDIRECT ACL

it is handled the same way. Finally if the client does not exist in any of the S3Bware ACLs it is handled by a default policy.

Algorithm 1 Access control for client request r and source IP address reqSrcIP returning action for request.

```
AK = getCookieValue('AK')
IP = getCookieValue('IP')
σ = getCookieValue('signature')
if not AK and not IP then
    if (∅, reqSrcIP) in ACL[BLOCK] then
        return block(r)
    else if (∅,reqSrcIP) in ACL[CHALLENGE] then
        return redirectToChallenge(r)
    else
        return register(r)
    end if
end if
if not verified (AK, IP, σ, k) then
    sendAlert(reqSrcIP, ALERT_VERIFY_FAIL)
    return defaultPolicy(r)
end if
if (AK,IP) in ACL[BLOCK] then
    return block(r)
else if (AK,IP) in ACL[CHALLENGE] then
    return redirectToChallenge(r)
else if (AK, IP) in ACL[ALLOW] then
    return allow(r)
else if (AK, *) in ACL[ALLOW] or
(AK, IP) in ACL[REDIRECT] then
    newToken = createToken(AK, reqSrcIP)
    return redirect(newToken, AK)
else
    return defaultPolicy(r)
end if
```

V. Implementation

We have developed an open source S3B implementation [20] consisting of several ready to use S3B apps, a standalone controller written in Go, a CoreDNS [21] module, and an S3Bware Nginx [22] module.

A. Applications

Our implementation provides two applications to support basic operation and a spider isolation app to demonstrate its ability to be used as a security appliance. For basic operation, S3B must be able to provide client-server assignments for registration and to the webapp server. At least one server must join S3B who supports these corresponding roles.

`RRRegistration` Client registration requests are handled by the `RRRegistration` app which uses a round-robin algorithm for server selection. This application registers a handler for the `Join`, and `UnknownClient` callbacks. When the app's `Join` handler is called, the S3Bware's profile (including the server's IP address, unique domain name, public key, and preferred role) is added to an availability list indicating all servers that will perform client registrations. The `UnknownClient` handler is called every time a DNS query is made from an unknown client, and the app selects from the availability list using a round robin algorithm returning the DNS answer with the selected servers IP address. Periodically the app makes a web request to each S3Bware in the availability list to confirm it is still reachable. If not, it is removed from the list.

`RRAssignment` Assignments to webapp servers are done by the `RRAssignment` app which registers handlers for the `Join` and `ExistingClient` callback. This app maintains a separate availability list as the conditions for a joined S3Bware being assigned to the role of a webapp server may differ than one set as a registration server. DNS queries made by existing clients are answered the same way as the `RRRegistration` app, using a round-robin algorithm. S3Bware availability checks are also done the same.

`SpiderTrap` To isolate web spiders to a honeypot server we have developed the `SpiderTrap` app. This app requires at least one S3Bware to have the role of a honeypot server and a supporting S3Bware plugin which is discussed in Section V-D. This application registers a handler for the `Join`, `ExistingClient` as well as `SecurityAlert` callbacks. As S3Bware join S3B, the `Join` handler checks the preferred role and for those set as a honeypot role their profile is added to a honeypot server list. `SpiderTrap` isolates spiders in two stages: (1) detection and labeling of AKs as spiders, and (2) reassignment to the honeypot server.

Detection and labeling occur by the `SecurityAlert` handler reading the alert messages to identify honeypot link requests. The tuple (IP address, AK) identifying the spider included in the `SecurityAlert`, is added to a spider list. The app responds to the `SecurityAlert` that the server should redirect all future requests with the matching AK.

Spider reassignment is initiated when then the spider issues a subsequent request to the webapp, which in turn causes a redirect and an `ExistingClient` message to be sent to the controller. When an `ExistingClient` DNS query is received, the clients AK and IP address are obtained from the token and then matched against the spider list. If a match is found, `SpiderTrap` overwrites the DNS answer provided by `RRAssignment` and replaces the answer with the IP address of a honeypot server. `SpiderTrap`'s `ExistingClient` is inserted into the callback chain with a higher priority than `RRAssignment`'s handler allowing the DNS answer to be overwritten.

B. Controller

Our controller is implemented as a standalone web application written in Go. The API discussed in Section IV-C is exposed through a RESTful web API. The controller is based on a chained middleware model to allow for the inclusion of the S3B apps previously discussed. Each app is able to register to specific messages received through the API.

C. Name Server

We use the CoreDNS [21] DNS server with custom middleware. The implementation provides a cache for DNS query QNames and if there is a miss in the cache the controller is queried to provide an answer. The module communicates with the controller through its RESTful web API and messages are signed with ECDSA using the modules private key. CoreDNS

performs the majority of the heavy lifting, handling the basic DNS server functionality and providing an API for working with DNS records.

D. S3Bware

The core functionality of S3B is provided by S3Bware. Our S3Bware is written as an Nginx module, providing full transparency to upstream servers and applications. The module is able to intercept both requests and responses. Cryptographic operations are preformed by OpenSSL[23] including AK generation, token encryption and inter-system message authentication. Furthermore as an Nginx module, as demand increases S3Bware is able to horizontally scale with its web server.

Assignment Enforcement Our Nginx module uses the `Origin` HTTP header to determine if the request originated from the controller. If the value is equal to the controllers host name it is processed as a controller message, otherwise as a request from a web client. Each message from the controller contains a signature stored in a cookie. If the signature is verified S3Bware processes the message. Otherwise a security alert is sent to the controller and the request is rejected. The S3Bware enforces the client-server assignments through an in memory ACL updated by the control messages defined in Table II.

Registration If the request appears to be from a web client, the S3Bware follows Algorithm 1. To register the client, the S3Bware generates a unique token. Our implementation supports both IPv4 and IPv6 address however due to the increased length of the IPv6 address, the formats differ. The default token formats are illustrated in Table III specifying the number of bits and characters each field occupies. The administrator additionally has the ability to create their own formats to meet their specific needs since there is some flexibility. For example, the S3BwareC could be decreased while increasing the AK. Two functions are helpful for constructing token formats when working with bit fields that must be base 36 encoded (*i.e.*, all fields other than the S3BwareID which must start with a letter),

$$len(chars) = \lceil b/\log_2(36) \rceil \quad (3)$$

$$len(b) = \lfloor chars \times \log_2(36) \rfloor \quad (4)$$

Equation 3 converts the number of bits to the number of base 36 characters, and Equation 4 preforms in the inverse.

The AK must be at least a 64-bits which is minimally recommend by OWASP [19]. This is randomly sampled from OpenSSL's secure pseudo-random generator. The IV's deterministic construction supports up to 26 S3Bware instances to perform encryption as defined by the identifier field (S3BwareID). The AK and IP address are padded to to their full length (*i.e.*, (32+64) bits and (128-64) bits for IPv4 and IPv6 addresses respectively) and then encrypted using AES-256 in GCM mode with the IV and configured key k. The token is formed by appending the S3BwareID with the base 36 encoding of the S3BwareC, tag, and ciphertext. The IV's invocation field S3BwareC is then incremented and saved to the file system.

Once the token has been created, the clients IP address and generated AK is then signed using an HMAC-SHA256 signature with a configured symmetric key k. The AK, IP address, and signature are then set as a cookie with the `HttpOnly` and `Secure` flags enabled. Finally, the S3Bware responds with a temporary redirect (*i.e.*, 307) with the HTTP Location header set to the domain with the token appended as an additional domain label.

		S3BwareID	S3BwareC	T	IP	AK	Total
IPv4	b	5	93	128	32	64	322
	chars	1	18	44			63
IPv6	b	5	31	96	128	64	324
	chars	1	6	56			63

TABLE III: IPv4 and IPv6 token formats specifying the bit length (b) and character count (chars) for each field.

`SpiderTrapMod` To support the `SpiderTrap` app, an S3Bware plugin to detect a spider is used to maintain transparency. This plugin is another Nginx module that intercepts request and responses. Responses are injected with honeypot links in the webpage, while requests are monitored to detect if a honeypot link is made. Honeypot links are relative URLs, randomly generated with a pre-configurable prefix that cannot be fingerprinted. Thus as the plugin monitors requests it can identify if the requested link is a honeypot link it created. If a honeypot link is requested, the plugin will redirect the request to a configurable location (*e.g.*, an arbitrary URL in the webapp) and send a security alert to the controller including the clients AK and IP address.

VI. Evaluation

In this section we evaluate our S3B implementation to identify performance impacts on the client and assess how effective `SpiderTrap` is at isolating spiders. In the following experiments S3B was deployed across three Amazon Ubuntu 16.04 EC2 micro instances in the us-east-1 region. Two web servers ran S3Bware, one of which was configured as the honeypot server, and the second as the webapp. The third server ran the name server and controller. Inter-region deployments can expect additional latencies [24].

A. Registration Overhead

Client registration requires an additional HTTP and DNS request. To evaluate the overhead imposed by the registration process we compare the page request time for when a website is enabled with S3B, and when it is disabled. We took measurements from within AWS as well as from three servers geographically dispersed across the United States using GENI [25]. When running the experiment with S3B disabled we configured CoreDNS to use a static zone file with a wildcard DNS record for our website domain to allow us to generate a unique subdomain for each request to bypass any possibility of DNS caching. When enabled, the `RRRegistration` app was similarly configured to allow arbitrary subdomains. Additionally, Nginx was configured to add HTTP headers to prevent caching. Pages were filled with dummy data incremented by 500KB in size to 4MB which includes the current average page size of 2.6MB [26]. A page size of 0 bytes is a response without a body (*i.e.* equivalent to a HEAD response). We performed 1,000 trials of each page size.

Fig. 3 reports the mean time and 95% confidence interval for each page size at each location. We have separated the registration time (suffix _on_reg) from the overall page request time (suffix _on) when S3B is enabled to clearly see the overhead. Results show registration time is constant as a function of the page size which is consistent with the design. The HTTP redirects are constant in size, thus the only increase in request time is due to the growing size of the page. For a page size of 2.5MB, there was on average a 23.80% increase in overhead, or 155.71ms which is just about at the limit (100ms) in which a human cannot discern any additional overhead [27]. As page sizes increases, overhead decreases. Due to registration being a one-time event we find this overhead to be negligible for most services. For services where this overhead is unacceptable, solutions such as Akamia web security [28] may be more appropriate which is better suited to handle large scale network traffic, however this does come at a cost.

B. `SpiderTrap` Evaluation

In this section we evaluate how effective `SpiderTrap` is at isolating spiders to a honeypot server. First, we conduct experiments to establish the assignment change reaction time of S3B with `SpiderTrap` installed in our deployed environment. This measurement allows us to establish a baseline to predict the number of requests a spider can make with a given bandwidth *before* being isolated. More specifically we measure,

$$t = t_{\texttt{SpiderTrapMod}\rightarrow\text{Ctrl}} + t_{\text{Ctrl}\leftrightarrow\texttt{SpiderTrap}} + t_{\text{Ctrl}\rightarrow\text{Honey}} + t_{\text{Honey}\rightarrow\text{ACL}}$$

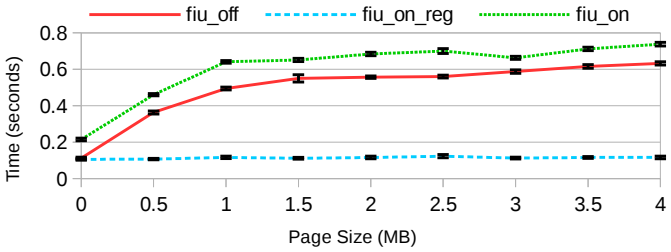
where $t_{\texttt{SpiderTrapMod}\rightarrow\text{Ctrl}}$ is the time it takes the `SpiderTrapMod` to send a `SecurityAlert` to the controller specifying a honeypot link has been requested, $t_{\text{Ctrl}\leftrightarrow\texttt{SpiderTrap}}$ is the processing time `SpiderTrap` takes to establish the new assignment, $t_{\text{Ctrl}\rightarrow\text{Honey}}$ is the time it takes to send the `ACAddRule` message to S3Bware residing on the honeypot server, and $t_{\text{Honey}\rightarrow\text{ACL}}$ is the time it takes the honeypot S3Bware to update it's ACL and enforce the new assignment. From the webapp server we generated 1,000 `SecurityAlerts` and found the mean reaction time t to be $3.79 \pm 0.21ms$ with 95% confidence.

Next we deployed a sample webapp using S3B and `SpiderTrap`. The `SpiderTrapMod` is installed as an Nginx module on the web server hosting the webapp. The webapp serves pages consisting of a honeypot link followed by 1,000 randomly generated links to simulate the limit of links found on a typical webpage [29].

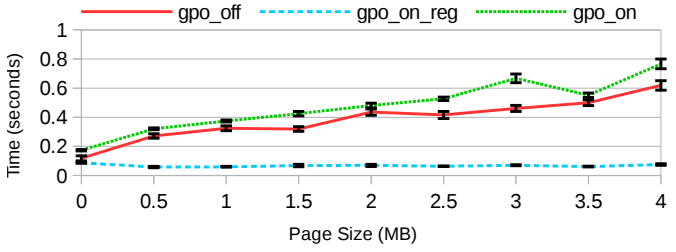
To evaluate `SpiderTrap` we created a spider based on the popular Scrapy framework [30]. We measure the spiders effectiveness as the number of pages crawled from the webapp *after* the honeypot link is requested. We refer to this metric as damage. For each experiment the spider crawls the webpage by depth-first-search as fast as it can until it is isolated. We adjust the number of requests the crawler can make in parallel from 1 to 32, with the default set by Scrapy at 16. The spider is run from the same locations as Section VI-A. For each location we also report the average bandwidth.

Results in Fig. 4 show `SpiderTrap` to be an effective security appliance for isolating spiders. Isolation occurred

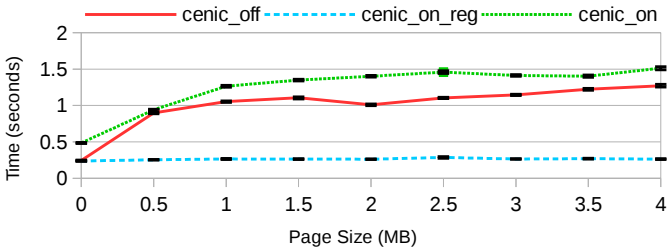
(a) AWS: North Virginia



(b) FIU ExoGENI: Miami, FL



(c) GPO InstaGENI: Cambridge, MA



(d) CENIC InstaGENI: Los Angeles, CA

Fig. 3: Time comparison for page request of a new client with S3B off, and on. When S3B is on, the specific client registration time is reported.

in just a single round of requests and when requests are sequential, on average no damage is inflicted. To increase damage a spider would require a network connection capable of crawling the target pages faster than S3B's reaction time of $3.79 \pm 0.21ms$.

The spiders optimal strategy is to make as many requests in parallel as possible. However the server can counter this strategy by forcing requests to be made sequentially (*e.g.*, Nginx's module `ngx_http_limit_conn_module`) thereby decreasing damage back to zero. Advanced spiders

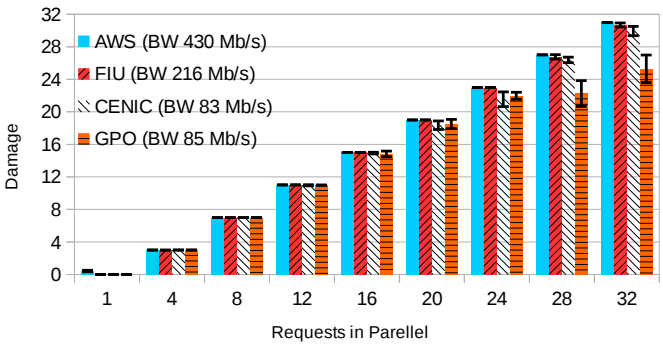


Fig. 4: Spider damage before isolation

may attempt to detect they have been isolated by S3B (*e.g.*, monitoring the IP address of current server, or the number of 307 responses received), clear their cookies to re-register and continue crawling the webpage. In the case of continued abuse, `SpiderTrap` can temporarily challenge, or block new registrations from the spider's IP addresses while still allowing previously registered user to continue using the website without degrading their service.

VII. Related Work

There has been a significant amount of research involved in establishing client-server assignments. Previous work has primarily been motivated by improving quality of service and decreasing request latency. These techniques known as request-routing, route (or assign) a client to a particular server based on a defined policy or set of metrics [6]. The majority of these techniques are either DNS-based or application-layer mechanisms.

DNS-based server assignments traditionally use the location of the local recursive name server to decide which server will be assigned to the client [1]. It is assumed the client is in close proximity to the name server however is has been found that this is not always the case [1][2]. MyXDNS [31] introduces dynamic DNS client-server assignments separating the data and control plane and exposing an API to allow users to specify their own selection policy. S3B has taken a similar approach to allow the customer to specify assignments through an applications, which provides the greatest degree of freedom. However the MyXDNS architecture only considers the IP address of the client for server selection. DONAR [32] introduces additional metrics to improve client-server assignments in addition to location such as server workload and bandwidth costs. DONAR is more restrictive than MyXDNS. Users of DONAR are not free to write their own algorithms. Instead, users specify selection policies based on the performance metrics which is then used by DONAR to solve an optimization algorithm to determine server assignments. To provide more precise mappings, end-user mapping [2] allows a local recursive name server to forward additional information about the client (*e.g.*, the clients IP address prefixed to a certain number of bits) to the authoritative name server. The inclusion of additionally information is made possible to the Internet draft, EDNS0 [33].

There are a number of application-layer server assignment techniques, specifically for HTTP [6][34]. HTTP-based assignment mechanisms provide a greater degree of control due to the knowledge of the client and their request. Assignments decisions are based off HTTP header values and routed using various methods such as URL rewriting and redirection [35]. Additionally a software defined network (SDN) approach called Plug-n-Serve [36] has been proposed providing load-balancing for web traffic using customized routes established based on feedback from the network.

Client-server assignment mechanisms have also been motivated with security in mind. A shuffling mechanism presented by [37] and [38] repeatedly shuffle client-server assignments to separate benign clients from bots involved in a denial of service (DoS) attack. IP hopping mechanisms have been proposed to prevent worms from conducting reconnaissance by rapidly changing the client-server IP mapping [39] [40]. By frequently changing the mapping, the information gathered by hit-list based worms is stale by the next stage of the attack. A variant of IP hopping known as fast fluxing [41] is a technique used by botnets to frequently change the IP addresses of a layer of proxy nodes. This additional layer of indirection adds resiliency to the botnet making it much harder to disrupt.

VIII. Future Work and Conclusion

In future work we will continue to evolve S3B to improve the control administrators have over how their webapps are accessed. We plan to increase the expressiveness of the ACL rules such as adding support for rate limiting requests and limiting the number of connections per IP address. Additionally we will explore S3B's ability to be used for other security related applications such as a defense for application layer (L7) DDoS attacks.

To conclude, this paper introduced S3B (Software-defined Secure Server Bindings). S3B solves two problems that exist in traditional request-routing mechanisms: (1) imprecise client-server assignments and (2) a lack of enforcement of these assignments. Although these issues may not be a concern for performance-centric purposes, from a security standpoint they are critical. Our prototype performance evaluation suggests there is no discernible overhead for client requests. Furthermore we have shown S3B to be an effective security appliance for isolating spiders, capable of virtually immediate containment once detected.

Acknowledgment

We would like to thank our shepherd Georgios Portokalidis and the anonymous reviewers for helping us improve this paper. This work has been supported by the National Science Foundation (NSF) awards #1430145, #1414119, #1717858 and #1718135.

REFERENCES

[1] A. Shaikh, R. Tewari, and M. Agrawal, "On the effectiveness of dns-based server selection," in *INFOCOM 2001. Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, vol. 3. IEEE, 2001, pp. 1801–1810.

[2] F. Chen, R. K. Sitaraman, and M. Torres, "End-user mapping: Next generation request routing for content delivery," in *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 4. ACM, 2015, pp. 167–181.

[3] S. Khattak, D. Fifield, S. Afroz, M. Javed, S. Sundaresan, V. Paxson, S. J. Murdoch, and D. McCoy, "Do you see what i see? differential treatment of anonymous users," in *Network and Distributed System Security Symposium*, 2016.

[4] "Session Affinity," April 10, 2017. [Online]. Available: https://devcenter.heroku.com/articles/session-affinity

[5] T. Vissers, T. Van Goethem, W. Joosen, and N. Nikiforakis, "Maneuvering around clouds: Bypassing cloud-based security providers," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015, pp. 1530–1541.

[6] B. Cain, A. Barbir, R. Nair, and O. Spatscheck, "Known content network (CN) request-routing mechanisms," 2003.

[7] "AWS Lambda," January 2017. [Online]. Available: https://aws.amazon.com/lambda/

[8] E. Bursztein, M. Martin, and J. Mitchell, "Text-based captcha strengths and weaknesses," in *Proceedings of the 18th ACM conference on Computer and communications security*. ACM, 2011, pp. 125–138.

[9] S. Sivakorn, J. Polakis, and A. D. Keromytis, "Im not a human: Breaking the google recaptcha," *Black Hat*, 2016.

[10] M. Motoyama, K. Levchenko, C. Kanich, D. McCoy, G. M. Voelker, and S. Savage, "Re: Captchas-understanding captcha-solving services in an economic context." in *USENIX Security Symposium*, vol. 10, 2010, p. 3.

[11] G. Ollmann, "Stopping automated attack tools," *Whitepaper–NGS software insight security research*, 2005.

[12] L. Spitzner, "The honeynet project: Trapping the hackers," *IEEE Security & Privacy*, vol. 99, no. 2, pp. 15–23, 2003.

[13] F. Toffalini, M. Abbà, D. Carra, and D. Balzarotti, "Google dorks: Analysis, creation, and new defenses," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2016, pp. 255–275.

[14] L. Von Ahn, M. Blum, N. J. Hopper, and J. Langford, "Captcha: Using hard ai problems for security," in *Advances in CryptologyEUROCRYPT 2003*. Springer, 2003, pp. 294–311.

[15] S. H. Yeganeh, A. Tootoonchian, and Y. Ganjali, "On scalability of software-defined networking," *IEEE Communications Magazine*, vol. 51, no. 2, pp. 136–141, 2013.

[16] "Infrastructure layouts when TLS is involved." [Online]. Available: https://www.haproxy.com/doc/aloha/7.0/deployment_guides/tls_layouts.html

[17] P. V. Mockapetris, "Domain names: Implementation specification," 1983.

[18] M. J. Dworkin, "SP 800-38D. Recommendation for block cipher modes of operation: Galois/Counter Mode (GCM) and GMAC," 2007.

[19] OWASP, "Session Management Cheat Sheet - OWASP," January 2017. [Online]. Available: https://www.owasp.org/index.php/Session_Management_Cheat_Sheet

[20] "S3B," December 4, 2017. [Online]. Available: https://github.com/wil3/s3b

[21] "CoreDNS," January 2017. [Online]. Available: https://coredns.io/

[22] "Nginx," January 2017. [Online]. Available: https://www.nginx.com/

[23] "OpenSSL," January 2017. [Online]. Available: https://www.openssl.org/

[24] M. Adorjan, "AWS Inter-Region Latency," 2018. [Online]. Available: https://www.cloudping.co/

[25] M. Berman, J. S. Chase, L. Landweber, A. Nakao, M. Ott, D. Raychaudhuri, R. Ricci, and I. Seskar, "Geni: A federated testbed for innovative network experiments," *Computer Networks*, vol. 61, no. 0, pp. 5 – 23, 2014, special issue on Future Internet Testbeds Part I. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1389128613004507

[26] "HTTP Archive," April 2017. [Online]. Available: http://www.httparchive.org/interesting.php

[27] R. B. Miller, "Response time in man-computer conversational transactions," in *Proceedings of the December 9-11, 1968, fall joint computer conference, part I*. ACM, 1968, pp. 267–277.

[28] "Akamai Web Security," 2018. [Online]. Available: https://www.akamai.com/us/en/resources/web-security.jsp

[29] "Webmaster Guidelines," April 10, 2017. [Online]. Available: https://support.google.com/webmasters/answer/35769

[30] "Scrapy, A Fast and Powerful Scraping and Web Crawling Framework," April 2017. [Online]. Available: https://scrapy.org/

[31] H. A. Alzoubi, M. Rabinovich, and O. Spatscheck, "Myxdns: A resquest routing dns server with decoupled server selection," in *Proceedings of the 16th international conference on World Wide Web*. ACM, 2007, pp. 351–360.

[32] P. Wendell, J. W. Jiang, M. J. Freedman, and J. Rexford, "Donar: decentralized server selection for cloud services," *ACM SIGCOMM Computer Communication Review*, vol. 40, no. 4, pp. 231–242, 2010.

[33] P. Vixie, "Extension mechanisms for dns (edns0)," 1999.

[34] V. Cardellini, M. Colajanni, and P. S. Yu, "Dynamic load balancing on web-server systems," *IEEE Internet computing*, vol. 3, no. 3, pp. 28–39, 1999.

[35] ——, "Redirection algorithms for load sharing in distributed web-server systems," in *Distributed Computing Systems, 1999. Proceedings. 19th IEEE International Conference on*. IEEE, 1999, pp. 528–535.

[36] N. Handigol, S. Seetharaman, M. Flajslik, N. McKeown, and R. Johari, "Plug-n-serve: Load-balancing web traffic using openflow," *ACM Sigcomm Demo*, vol. 4, no. 5, p. 6, 2009.

[37] Q. Jia, K. Sun, and A. Stavrou, "Motag: Moving target defense against internet denial of service attacks," in *Computer Communications and Networks (ICCCN), 2013 22nd International Conference on*. IEEE, 2013, pp. 1–9.

[38] Q. Jia, H. Wang, D. Fleck, F. Li, A. Stavrou, and W. Powell, "Catch me if you can: A cloud-enabled ddos defense," in *Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on*. IEEE, 2014, pp. 264–275.

[39] J. H. Jafarian, E. Al-Shaer, and Q. Duan, "Openflow random host mutation: transparent moving target defense using software defined networking," in *Proceedings of the first workshop on Hot topics in software defined networks*. ACM, 2012, pp. 127–132.

[40] S. Antonatos, P. Akritidis, E. P. Markatos, and K. G. Anagnostakis, "Defending against hitlist worms using network address space randomization," *Computer Networks*, vol. 51, no. 12, pp. 3471–3490, 2007.

[41] J. Nazario and T. Holz, "As the net churns: Fast-flux botnet observations," in *Malicious and Unwanted Software, 2008. MALWARE 2008. 3rd International Conference on*. IEEE, 2008, pp. 24–31.