DIY Hosting for Online Privacy

Shoumik Palkar and Matei Zaharia
Stanford University
[shoumik,matei]@cs.stanford.edu

Abstract

Web users today rely on centralized services for applications such as email, file transfer and chat. Unfortunately, these services create a significant privacy risk: even with a benevolent provider, a single breach can put millions of users' data at risk. One alternative would be for users to host their own servers, but this would be highly expensive for most applications: a single VM deployed in a high-availability mode can cost many dollars per month. In this paper, we propose Deploy It Yourself (DIY), a new model for hosting applications based on *serverless computing* platforms such as Amazon Lambda. DIY allows users to run a highly available service with much stronger privacy guarantees than current centralized providers, and at a dramatically lower cost than traditional server hosting. DIY only relies on the security of container isolation and a key manager as opposed to the large codebase of a high-level application such as Gmail (and all the Google teams using Gmail data). With attestation technology such as SGX, DIY's execution could also be verified remotely. We show that a DIY email server that sends 500 messages/day costs $0.26/month, which is 50× cheaper than a highly available EC2 server. We also implement a DIY chat service and show that it performs well. Finally, we argue that DIY applications are simple enough to operate that cloud providers could offer a simple "app store" for using them.

1 Introduction

Web users rely on centralized providers to host communication services such messaging, email, file sharing, and teleconferencing. These systems offer high availability, low cost, and ease of use: any user can create an account online to use them.

Unfortunately, in exchange for the high availability and low cost these services provide, users sacrifice data privacy. Privacy concerns have been steadily rising on the web due to issues including promiscuous data resale and surveillance [7, 27, 28]. Even if a provider is perfectly benevolent, it only takes a single breach or rogue employee to put users' data at risk, as in the 2015 Yahoo! Mail hack that affected nearly one billion users [38]. Many providers also lock in user data, making it hard to change providers, export data if a service shuts down [17], or control the data's geographic placement.

One strawman alternative would be for users to host their own online applications (*e.g.*, email servers), either by running a server at home or using a VM in a public cloud. Widely used communication protocols such as SMTP and XMPP already support this through their federated design. Unfortunately, hosting highly available servers would be far too expensive for most users. Even ignoring the complexity of server management, which could perhaps be automated by software, a small virtual server in the public cloud costs at least $5/ month to run 24/7, and monitoring and failover cost even more. Users are unlikely to take on this type of expense for every service they use in order to gain more control over their data.

This paper proposes Deploy It Yourself (DIY), a new model for deploying personal online applications at dramatically lower costs while maintaining data privacy. The key idea in DIY is to use emerging *serverless computing* platforms such as Amazon Lambda [22] and Azure Functions [6] to host private online applications. Serverless platforms are highly available, georeplicated systems that can run arbitrary user code but bill usage in a *pay-per-request* fashion at sub-second granularity. These platforms are currently used for event processing or batch computation in web applications, but we argue that they are an ideal fit for private online applications as well: users would pay for just a few hundred requests per day (a few minutes of total computing time), while still enjoying the availability of a best-in-class Internet service.

DIY offers much stronger privacy than centralized services that store and operate over plaintext data. DIY stores *encrypted* user data on cloud storage providers such as Amazon S3, and only decrypts data while processing a request in an OS container. Decryption keys are stored in a key management service such as Amazon KMS [21], which is simple and relatively easy to protect. DIY thus trusts only the isolation mechanisms of the serverless platform and the KMS, in contrast to the massive trusted computing base of an application such as Gmail and all the internal analytics teams at Google that use Gmail data. DIY's narrow interface also means that the container execution could eventually be verified remotely using technology such as SGX [1]. Moreover, DIY gives users full control to migrate their application to another provider, control its geographic placement to avoid unfriendly surveillance laws, or delete data.

We evaluate the feasibility of DIY through a detailed cost analysis of five applications and a minimal prototype of a group chat service. We estimate that based on typical usage at our organization, services such as email and group chat (*e.g.*,

HotNets-XVI, November 30–December 1, 2017, Palo Alto, CA, USA

ACM ISBN 978-1-4503-5569-8/17/11…$15.00
https://doi.org/10.1145/3152434.3152459

Slack) could cost users anywhere from $0.12–$0.26/month in total (including storage and bandwidth), as opposed to roughly $4.50/month for a small virtual server running 24/7 with *no* failover configured on EC2. Likewise, users can run private teleconference servers at $0.11 for an hour-long HD call. Our chat prototype achieves acceptable latency, showing that performance need not suffer with DIY applications. Overall, DIY can be an order of magnitude less expensive than virtual hosting platforms while keeping user data private and maintaining the availability of a top-tier cloud service.

Finally, we speculate that DIY applications may be simple enough to operate that non-expert users could eventually install and manage them through an "app store." This would give software developers (*e.g.*, a startup company) a powerful new vehicle for delivering communication applications with strong security guarantees without having to host and manage a complete multitenant service on their own, and accelerate both innovation and user choice in Internet applications.

2 Target Applications

DIY targets online applications that perform independent computations for each user (or each group of interacting users). Some examples of suitable applications include:

- Chat applications such as instant messenger, IRC and Slack, which relay messages and store history among either pairs or groups of users.
- Email, where servers must be up 24/7 to receive messages and deliver them to multiple client devices.
- File transfer services like Apple's AirDrop or Dropbox, for privately sending a large file to a group of users.
- Video or audio conferencing, where DIY can be used to spin up a private relay server.
- Internet of Things (IoT) services such as a management application for a smart home.

These applications are a good fit for DIY because they have low request volume per user—for example, most people only receive hundreds of emails or chat messages per day. In contrast, applications that might not be a good fit for DIY include content publishing to a large audience (*e.g.*, YouTube) or applications that must perform complex computation across many different users (*e.g.*, news feed ranking in a social network).

3 Goals

DIY's main goals are to provide *high availability*, *low cost*, and *privacy* for online applications. We next discuss these in turn.

3.1 High Availability

DIY's first goal is providing high availability for users' applications. Availability and reliability are features users expect from any service, and are the major reasons centralized providers have grown so popular on the web. Availability requires geo-replicating the service, monitoring its health so new servers may take the place of old ones, and so forth. DIY achieves high availability by using serverless computing platforms, which handle these tasks transparently beneath their API.

3.2 Low Cost

DIY's second goal is to run applications at low cost, because many alternatives to hosting a web service today are free. For example, many centralized providers offer their functionality at no cost in exchange for advertising revenue, or only charge users for premium features. It is thus impractical to ask users to have to pay several dollars a month for every service they use, even if DIY provides stronger privacy guarantees. DIY again leverages serverless platforms to deliver low cost by relying on their fine-grained *pay-per-request* pricing model.

3.3 Privacy

DIY's third goal is privacy. Today's centralized services store and access plaintext user data in order to operate. These services put data at risk for at least five distinct reasons:

(1) Providers may sell user data, as in the case of email cleanup service `unroll.me`, which sold anonymized Lyft ride receipts to Uber [7].

(2) Providers may use data for ad targeting, which leaks information about the user [35]. For example, Facebook reserves the right to use Messenger data this way [14].

(3) Providers use private data for many internal applications (*e.g.*, to build ad targeting or recommendation engines), creating a large trusted computing base that must protect data against internal threats (attackers who gain unprivileged access to database engines, analytics systems, etc).

(4) Providers must trust potentially thousands of employees who have access to user data for testing or maintaining internal systems. Even respected organizations have had employees snoop on user data illicitly [16, 19, 34].

(5) Providers may lock in data, making it hard to switch to different services or controlling its geographic placement.

DIY greatly reduces the trusted computing base required to run an online service. Although DIY itself runs on a public cloud, it does so through a much narrower interface, placing trust only in container hosting and a key management service, in contrast to the vast set of interconnected services at a centralized provider. In addition, unlike centralized services, public cloud providers cannot access plaintext data via these narrow interfaces without a serious and *explicit* breach of their security models. Finally, DIY gives users back control over data placement, migration and deletion.

Threat Model. DIY relies on two systems to be reliable in its threat model: the serverless computing platform (and in particular, its ability to hide the execution and state of user functions from attackers) and the key management service (including providing the user's key only to her serverless functions). Beyond this, we assume an attacker that has access to the cloud provider's internal network, to other cloud services (e.g., storage) and to Internet traffic between the user and the cloud.

The data stored in these external services and communicated between the cloud and the user is encrypted.

DIY aims to protect the plaintext content of the user's data, *e.g.*, emails, chatlogs, etc. DIY does not attempt to guard against traffic analysis or access pattern attacks, although it may make it harder to run classic attacks such as VM colocation [30] due to the short lifespan of serverless functions.

We make no assumptions about how serverless functions' code is stored (*i.e.*, it may be unencrypted and accessible by adversaries, as is typical in current offerings), but do assume that the cloud provider faithfully executes the correct function code, and also assume that the function code itself is trusted (*e.g.*, the function will not send user data to an adversary).

Why is DIY More Secure? DIY improves privacy over centralized services for several reasons. First, user data is only decrypted within the isolated container running a serverless function. Decryption keys reside within secure key management services which even employees of the cloud provider cannot access. We believe that a breach of the key management service (a hardened, audited system [9] whose main goal is securing encryption keys) is significantly more difficult to achieve than a breach caused by a rogue employee snooping through user data at a web company. The provider thus cannot sell user data or leak it indirectly through ad targeting.

Second, the trusted computing base of a DIY service is much smaller than that of systems like Gmail. The DIY architecture requires trusting only the isolation mechanisms of the serverless platform and the key management service, as opposed to the plethora of analytics systems which support and monetize the centralized service by reading plaintext user data. In short, in a centralized service the line between the web service a user interacts with (*e.g.*, Gmail) and the systems supporting it are blurred, but in DIY the user only needs to trust a small audited code base.

Third, DIY reduces the number of individuals employed by the cloud provider that need to access user data. While companies such as Google are economically incentivized to provide employees with user data access (*e.g.*, to improve ad targeting), public cloud providers generally guarantee that they will minimize access to user data internally [5]. Key management services go as far as guaranteeing that *no* employee can access user data, and are further incentivized to keep keys secure, both to maintain product reputation and to avoid litigation [23].

Finally, users have the freedom of migrating their data across providers at any time, *e.g.*, to move out of insecure geographic regions or clouds. With centralized services, this is often not possible. For example, deleting an email may delete a record in a database, but data may have already been indexed, used to train a machine learning model, or copied into other services. Users thus have little control over where their data goes once they trust a centralized service with it.

Securing DIY with Enclaves. DIY could further protect function execution using hardware enclave technology such as Intel SGX [1], which cryptographically attests for the privacy and integrity of the function's execution. A serverless platform with enclave support could load the function into an enclave, perform its attestation, and then execute it in a manner that the client can verify. Key management services could also use enclaves to ensure that they only provide a user's key to enclaves running that user's code. Use of SGX in serverless platforms has not been explored widely and is not offered by providers today, but we believe that integrating enclave technology into these platforms is an interesting avenue for future work.

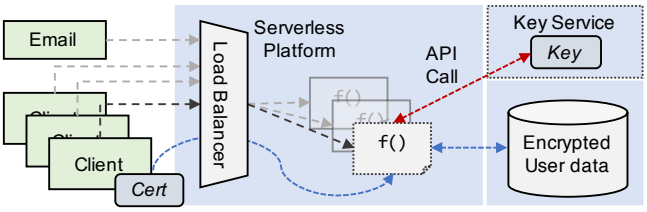

Figure 1: In DIY, requests spawn serverless functions, which access a key to securely retrieve data. Dotted boxes represent the trusted computing base: the isolation mechanisms of the OS container running the function and the key manager.

4 DIY Architecture

In this section we describe a design for DIY using features found on existing cloud platforms.

DIY relies on a fine-grained pricing model which charges requests based on execution time. For example, Lambda allocates functions a limited amount of memory (128MB to 1.5GB at the time of writing), and charges by GB-seconds (the amount of time the function executes weighted by the amount of memory allocated to the function). Amazon currently charges a flat $0.20 fee for every million requests and $0.00001667 for every GB-second, with one million free requests and 400,000 free GB-seconds each month. Execution time is measured in increments of 100ms. Providers thus impose no charge on an idle function, and cost scales *per request* (unlike serverless functions, even virtual servers which are billed at a fine granularity must always be running to listen for new requests). Automatic scaling and a pay-per-request pricing model are the key advantages of using serverless functions for hosting web services.

Figure 1 summarizes a design for DIY. The user first installs a serverless function and an *event trigger* which calls the function (*e.g.*, a message arriving at port 25 for an SMTP server). The function runs the code to process a single request. The serverless platform handles scaling, georeplication, load balancing among functions, and so forth automatically.

The user configures a storage provider such as Amazon S3 to store *encrypted* users data. An encryption key accesses the stored data, and as outlined in §3.3, we assume that the serverless function runs in an opaque container and can securely access the key without a cloud provider extracting it. The function can thus decrypt data in order to process requests.

Users store decryption keys in a secure key management service such as Amazon KMS [21]. The serverless function makes an API call to the key management service to obtain the key during execution, so the function only contains the key in its memory during execution (*i.e.*, the stored function

Storage	Compute	Availability	Total
Transfer: $0.09 Storage: $0.17	$4.32	Auto-scale: Free	$4.58

Table 1: Monthly cost of running an email service on AWS (most costs do not depend on request volume).

code does not contain the key). The management service authenticates the function's API call either via a client certificate provided with the request or by configuring the serverless function with appropriate permissions (*e.g.*, using IAM roles in AWS). DIY secures network requests to the function using standard encryption protocols such as TLS/SSL.

In all, the serverless function follows a standard workflow. After receiving a client request, the function accesses a key via an API call to the key manager, and then makes an API call to a storage provider to retrieve relevant data. It then decrypts the fetched data, processes the request, writes new data (*e.g.*, a new chat message or a new email), and returns data requested by the user as a response. Plaintext user data is only present in a small trusted computing base (the container running the serverless function). Note that features like hardware-based enclaves such as SGX further strengthening the privacy of DIY by removing the container isolation mechanism from the trusted computing base.

After installing the serverless function, registering a trigger, and associating it with a key and storage provider, deployments require little maintenance by the user. Indeed, we imagine cloud providers facilitating and managing deployment of DIY services through an "app store"-like user interface, where users download applications from a marketplace and associate it with an encryption key, with storage and permissions set automatically by the interface. We discuss what such a store might look like in §8.

5 Strawman: Traditional Hosting

As one possible strawman solution, consider hosting a personal email server using a VM-based server on a public cloud such as Amazon EC2. The server must always be able to respond to client requests, but is usually idle and likely receives at most hundreds of requests per day. VMs have similar security guarantees to using a serverless platform (like in DIY), but must configure high availability manually.

Table 1 summarizes the cost of running an email server on AWS [3] with costs broken down by component. With no replication, Amazon charges $4.58 for the smallest VM instance type. Replicating the instance to another geographic region doubles this cost. In contrast, §6 shows that we can deploy a DIY email server configured with high availability and with the same privacy guarantees for less than $0.30/month due to its pay-per-request model.

We note that services which host an email server for the user (which have the same privacy disadvantages of centralized systems) cost anywhere between $2/month [29] to $5/month [15]. DIY is less expensive than these services and provides stronger privacy guarantees since data in these hosted services is generally not encrypted.

6 Evaluation

We evaluate DIY in two ways. First, we estimate the cost of several services running on DIY using Amazon Lambda with typical request rates and usage patterns. These results show that DIY is practical even with today's serverless pricing, and supports a variety of common web services. Next, we describe a prototype XMPP-based chat service. Our prototype motivates DIY by showing it performs comparably to a web server at much lower cost and almost no management overhead.

6.1 Cost Analysis

Table 2 shows estimated costs for deploying several web services with DIY's design. Unless otherwise noted, for each service, we use Amazon Lambda as the serverless computing platform and Amazon S3 as the storage provider[1]. Accounting for both compute and storage, the costs of running these services is very practical: for most services, users should not expect to pay more than $1/month, a substantial saving compared to the cost of hosting a single server (see Table 1).

Group Chat. Group chat is a natural DIY service. Each user will send only up to hundreds of requests per day (for reference, the authors' Slack group sends an average of 5000 Slack messages *per week* among a group of 15 people) and each request requires only the user's data (*i.e.*, messages which the user should download or append). At 2000 messages sent/received per *day*, users can deploy a DIY chat service for free on Amazon Lambda. Assuming 2GB/month of data transfer and storage, users pay $0.14/month for these services.

Email. Email is another natural service for DIY. A serverless SMTP service can forward outgoing mail and encrypt and store incoming mail into a storage provider like Amazon S3. While Lambda currently does not support SMTP endpoints, we can use Amazon's SES service to provide the send service, and use Lambda as a hook to encrypt email (*e.g.*, using PGP encryption) before storing it. The compute cost for DIY email remains free until roughly 33,000 emails are sent or received *daily*; users pay $0.26/month for storage (assuming 5GB of data). DIY could also support features like spam detection using widely used open source detectors such as SpamAssassin [33].

Cloud Based File Transfer. DIY can be used to create a file storage and transfer server, providing a service similar to Apple's AirDrop service. Clients connect to the service with a request to transfer a file by filename and a recipient. The sender uploads the file to temporary storage, and the receiver downloads the file simultaneously. Table 2 shows an estimate for the cost of this service, assuming a 1GB file transfer. We assume only a small number of requests per day (most users likely don't transfer large files too often), but allocate more memory to the Lambda function to buffer the file. The service runs serverless functions for free in the typical case, with costs coming from file storage and data transfer.

[1] Amazon DynamoDB is a low-latency alternative to S3.

Application	Provider	Daily Requests	Compute Time per Request	Lambda Mem. (MB)	Monthly Storage (GB)	Monthly Compute Cost	Monthly Storage + Transfer Cost	Total Monthly Cost
Group Chat	Lambda	2000	500 ms	128	2	$0.00	$0.14	$0.14
Email	Lambda	500	500 ms	128	5	$0.00	$0.26	$0.26
File Transfer	Lambda	100	2000 ms	1024	2	$0.00	$0.14	$0.14
IoT Controller	Lambda	100	500 ms	128	1	$0.00	$0.12	$0.12
Video Conferencing	EC2	1	15 min call	-	1	$0.01	$0.83	$0.84

Table 2: Per-user costs of potential DIY services. We report the costs at typical request rates. In comparison, running a single dedicated cloud server on EC2 costs roughly $4.50. Services which host, *e.g.*, email have similar price points.

IoT Controller. An IoT controller, which relays user queries to IoT devices, is another service which fits the DIY architecture. Users send requests to the serverless function's endpoint. The function stores statistics/metadata about the queries and then relays the request to an IoT connected device. The function may also serve other features available in centralized IoT services, such as dashboards and alerts (where alerts are generated by IoT devices making requests to the service). At typical requests rates users can run this service for $0.12/month.

Private Video Conferencing. A video conferencing service is similar in design to a text-based chat service, but has stricter delay requirements and more demanding throughput requirements. The service works best on a platform where the serverless function allowed multiple incoming connections and relays data among connected clients. Since Lambda does not support multiple connections yet, we use a `t2.medium` EC2 instance (with 4GB of RAM), which is billed per second. Costs are quite practical: a 15 minute video call every day would cost roughly $0.84 per month, including data transfer and temporary storage. For data transfer, we assume Skype's recommended bandwidth of 3 Mbps for HD video calls [32], which translates to around 10GB transferred per month. A single hour-long HD call will cost roughly $0.11.

6.2 Prototype Implementation

As a proof-of-concept, we also implemented an instant messaging server using Amazon Lambda based on the XMPP protocol. Our implementation supports basic session initiation and message exchange. We tested all components in `us-west-2`.

Our server deviates from standard XMPP to work around limitations with existing serverless platforms. First, messages are tunneled through HTTPS, because Lambda only supports HTTP(S)-based endpoints. Second, XMPP over HTTP uses long-polling to receive messages. We implement long polling by having the serverless function post encrypted messages to Amazon's Simple Queue Service, which the client then long polls. The queuing service provides one million free requests per month and charges $0.40 for every million requests thereafter. Clients poll 876,000 times per month (assuming the maximum 20 second poll interval), which is well within the free tier.

We deployed our code on a 448 MB lambda function. Table 3 summarizes our findings. Even though our function only uses 51MB of memory, allocating 448 MB gave significantly better latencies than a 128 MB function; we found that API calls to S3 took significantly longer when we allocated less memory to the function. Note also that the Lambda execution time is only 138ms; most of the time in the end-to-end chat latency comes from waiting for a message to be delivered via SQS.

Statistic	Value
Med. Lambda Time Billed	200 ms
Med. Lambda Time Run	134 ms
E2E Chat Latency	211 ms
Lambda Memory Allocated	448 MB
Peak Memory Used	51 MB
Med. Lambda Cost per 100K Requests	$0.014

Table 3: Statistics collected for our chat service.

Users can send over 25,000 messages per day without incurring any compute cost and pay $0.09 per GB of transfer. We expect this cost to be low. We also note two qualitative advantages of DIY. First, services are easy to deploy. With some support from cloud providers, we can imagine a marketplace which users can use to start their own web services. Second, DIY keeps data *private* while providing similar availability guarantees as centralized services. We believe these two points present an exciting prospect: for the first time, any web user has the opportunity to reclaim control of their data and deploy their own web service as easily as running a program on a laptop.

7 Related Work

Web users are increasingly concerned with privacy, giving rise to a number of end-to-end encrypted applications such as Signal [31] and WhatsApp [37]. However, the protocols backing these applications run on clients and cannot, *e.g.*, host an SMTP server, since this service need access to plaintext data. DIY is a broader model to achieve stronger privacy for applications where servers need to process data.

DIY is related to designs that decouple user data from web services, such as Amber [10]. Unlike these works, DIY proposes a way for users to host their own web services rather than relying on centralized entities to manage data. Picocenter [41] proposes a new hosting platform for "mostly idle" applications that can efficiently swap out OS containers, and notes that it could be used for email servers. DIY shows that these services can run on *existing* serverless platforms such as Lambda at similar performance and cost, and also studies how to minimize an application's trusted computing base to improve privacy.

There have been several proposals for peer-to-peer protocols for services like social networking [8, 24], email [20], and VoIP [18]. These protocols may fit into the DIY design where each user stores her own data and connects to other users' services to exchange data. They could be significantly cheaper and more reliable to run on serverless platforms than on users' personal computers.

Finally, various systems propose performing computations over encrypted user data [11, 25, 26, 36] using cryptographic primitives such as homomorphic encryption. Unlike DIY, these techniques come with large performance overheads and require adoption from centralized web services. However, the threat model of these approaches is stronger than DIY's, since they do not require trusting a third party provider for key management. No-trust cryptographic communication protocols [2, 12] similarly have stronger threat models, but are harder to deploy (*e.g.*, they require running a mixnet).

8 Discussion and Open Questions

We have shown that DIY could give users significantly stronger privacy at low cost, without sacrificing the availability of current centralized Internet services. We now discuss questions that remain in improving the performance, usability and security of DIY, as well as possible future research directions.

8.1 DIY App Store

In order to materially improve web privacy, DIY applications must be easily deployable for the average Internet user. With some support from cloud providers or a third-party marketplace, we believe users may be able to install DIY applications with one click via an "app store"-like interface, similar to smartphone app stores today. These applications can be audited for security (as in the iOS app review process), and further secured via sandboxing mechanisms such as Native Client [39]. Users can then update or delete applications (and any corresponding data) at any time. The app store would also handle application resources (*e.g.*, setting up serverless functions, configuring storage, installing keys, etc) on behalf of the user and report their total resource consumption in a centralized UI, similar to the storage management interfaces on current smartphones.

We also believe that many application developers might be incentivized to adopt the DIY model because they would no longer have to handle securing each user's data, and could focus on functionality rather than service management and security. Today, any developer with an idea for a useful server-side application (*e.g.*, a competitor to Gmail, Slack or Evernote) must build and operate a complete, secure multitenant offering and gain users' trust to bring their idea to market. This creates substantial capital cost, operating expenses and business risk. In contrast, with a DIY app store, the developer could publish an application that gets automatically deployed in an isolated environment for each customer, and where the app store platform can give each user simple controls over what the application may do (*e.g.*, restrict the application to only communicate with this company's VPN). If this approach is successful, it cloud enable a proliferation of high-quality server-side apps similar to today's mobile app ecosystems, and greatly reduce the barriers to innovation and user choice for online applications.

To facilitate building DIY applications, we imagine that developers might extend the APIs in existing web programming frameworks, such as Django [13]. These APIs already handle concerns such as connection management and sessions, and are already being extended to run on serverless platforms [40].

8.2 Security Guarantees

Serverless functions today run in opaque containers, and hence provide better security than a centralized service which has access to all plaintext data under a threat model where the cloud provider does not try to extract information from the container. However, even in cases where the cloud provider is *not* trusted, solutions like Intel SGX [1] could provide strong security. We have not yet explored in detail how to apply these technologies in DIY, leaving this task to future research.

DIY applications are also susceptible to DDoS attacks, which can impose high financial cost to the user. These attacks may be mitigated by throttling requests using tools provided by the cloud provider (*e.g.*, AWS provides free basic DDoS protection [4]), but we have not explored them in detail.

8.3 Server Platform Limitations

Current serverless platforms such as Lambda only run functions in response to HTTP(S) requests or other classes of internal events (*e.g.*, posts to an Amazon SQS queue or uploads to S3). It would be interesting to expand cloud platforms so they can efficiently store arbitrary TCP servers with the same availability guarantees as current serverless platforms. This may be purely an engineering task, or may require some help from the application, *e.g.*, for load balancing. Likewise, a second limitation we found is that platforms do not easily support long idle connections (the function is billed while the HTTP request is active). Being able to suspend the user's container while a TCP connection remains open [41] could further improve these platforms' programmability and performance.

9 Conclusion

We have proposed Deploy It Yourself (DIY), a new model for deploying personal online applications at dramatically lower costs while maintaining data privacy. Users can deploy a highly available email server which keeps data private for only $0.26/month using DIY, and host a private hour long HD video call for only $0.11. We argue that DIY applications are simple enough to develop and deploy that cloud providers could offer a simple "app store" for using them. The prospect of online applications with a familiar developer API *and* effortless deployability is an exciting one: for the first time, perhaps, since the days of ARPANET, *any* Internet user will be able to run her own service with strong personal data privacy guarantees without sacrificing the functionality and availability of the centralized services we all rely on today.

Acknowledgements

We thank Pratiksha Thaker, Deepak Narayanan, Saba Eskandarian, and the many members of the Stanford InfoLab for their valuable feedback on this work. This research was supported in part by affiliate members and other supporters of the Stanford DAWN project – Intel, Microsoft, Teradata, and VMware – as well as NSF CAREER grant CNS-1651570. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

[1] Anati, Ittai and Gueron, Shay and Johnson, Simon and Scarlata, Vincent. Innovative Technology for CPU Based Attestation and Sealing. In *Proceedings of the 2nd international workshop on hardware and architectural support for security and privacy*, volume 13, 2013.

[2] S. Angel and S. T. Setty. Unobservable Communication over Fully Untrusted Infrastructure. In *OSDI*, pages 551–569, 2016.

[3] Amazon Web Services Simple Monthly Calculator. https://calculator.s3.amazonaws.com/index.html.

[4] AWS Shield. https://aws.amazon.com/shield/.

[5] AWS Data Privacy. https://aws.amazon.com/compliance/data-privacy-faq/.

[6] Azure Functions. https://azure.microsoft.com/en-us/services/functions/.

[7] S. Biddle. Stop Using Unroll.me, right now. It sold your data to Uber. https://theintercept.com/2017/04/24/stop-using-unroll-me-right-now-it-sold-your-data-to-uber/.

[8] A. Bielenberg, L. Helm, A. Gentilucci, D. Stefanescu, and H. Zhang. The Growth of Diaspora-a Centralized Online Social Network in the Wild. In *Computer Communications Workshops (INFOCOM WKSHPS), 2012 IEEE Conference on*, pages 13–18. IEEE, 2012.

[9] AWS Key Management Service Cryptographic Details. https://d0.awsstatic.com/whitepapers/KMS-Cryptographic-Details.pdf, 2015.

[10] T. Chajed, J. Gjengset, J. Van Den Hooff, M. F. Kaashoek, J. Mickens, R. Morris, and N. Zeldovich. Amber: Decoupling User Data from Web Applications. In *HotOS*, volume 15, pages 1–6, 2015.

[11] H. Corrigan-Gibbs and D. Boneh. Prio: Private, Robust, and Scalable Computation of Aggregate Statistics. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 259–282, Boston, MA, 2017. USENIX Association.

[12] H. Corrigan-Gibbs and B. Ford. Dissent: Accountable Anonymous Group Messaging. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 340–350. ACM, 2010.

[13] Django. http://djangoproject.com.

[14] Facebook Data Policy. https://www.facebook.com/full_data_use_policy.

[15] GoDaddy Email Hosting. https://www.godaddy.com/email/professional-email.

[16] Ex-Googler Allegedly Spied on User E-mails, Chats. https://www.wired.com/2010/09/google-spy/.

[17] Google Talk is Being Discontinued. https://news.ycombinator.com/item?id=13950002.

[18] S. Guha and N. Daswani. An experimental study of the skype peer-to-peer voip system. Technical report, Cornell University, 2005.

[19] Five IRS Employees Charged with Snooping on Tax Returns. https://www.wired.com/2008/05/five-irs-employ/.

[20] W. R. Kallman, D. L. Hoffman, and M. T. Mitchell. Peer-to-peer email, Oct. 20 2015. US Patent 9,166,937.

[21] Amazon Key Management Service. https://aws.amazon.com/kms.

[22] Amazon Lambda. https://aws.amazon.com/lambda.

[23] Data Breach Lawsuit. https://www.classaction.com/data-breach/lawsuit/.

[24] G. Mega, A. Montresor, and G. P. Picco. Efficient dissemination in decentralized social networks. In *Peer-to-Peer Computing (P2P), 2011 IEEE International Conference on*, pages 338–347. IEEE, 2011.

[25] R. A. Popa, C. Redfield, N. Zeldovich, and H. Balakrishnan. CryptDB: protecting confidentiality with encrypted query processing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 85–100. ACM, 2011.

[26] R. A. Popa, E. Stark, S. Valdez, J. Helfer, N. Zeldovich, and H. Balakrishnan. Building Web Applications on Top of Encrypted Data Using Mylar. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 157–172, Seattle, WA, 2014. USENIX Association.

[27] PRISM Surveillance Program. https://www.theguardian.com/us-news/prism.

[28] UK Report Finds Rising Digial Privacy Concerns. https://techcrunch.com/2016/04/21/uk-report-finds-rising-digital-privacy-concerns/.

[29] Rackspace Email Service. https://www.rackspace.com/en-us/email-hosting/webmail.

[30] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 199–212. ACM, 2009.

[31] Signal. https://whispersystems.org.

[32] How Much Bandwidth does Skype Need? https://support.skype.com/en/faq/FA1417/how-much-bandwidth-does-skype-need.

[33] Apache SpamAssassin. http://spamassassin.apache.org.

[34] Rogue tax workers snooped on ex-spouses, family members. https://www.thestar.com/news/canada/2010/06/20/rogue_tax_workers_snooped_on_exspouses_family_members.html.

[35] P. Vines, F. Roesner, and T. Kohno. Exploring adint: Using ad targeting for surveillance on a budget âĂŤ or âĂŤ how alice can buy ads to track bob. In *Workshop on Privacy in the Electronic Society*, 2017.

[36] F. Wang, C. Yun, S. Goldwasser, V. Vaikuntanathan, and M. Zaharia. Splinter: Practical Private Queries on Public Data. In *NSDI*, pages 299–313, 2017.

[37] WhatsApp Security. https://www.whatsapp.com/security/.

[38] Yahoo Says 1 Billion Accounts Were Hacked. https://www.nytimes.com/2016/12/14/technology/yahoo-hack.html?_r=0, 2016.

[39] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *Security and Privacy, 2009 30th IEEE Symposium on*, pages 79–93. IEEE, 2009.

[40] Zappa. https://github.com/Miserlou/Zappa.

[41] L. Zhang, J. Litton, F. Cangialosi, T. Benson, D. Levin, and A. Mislove. Picocenter: Supporting Long-lived, Mostly-idle Applications in Cloud Environments. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, pages 37:1–37:16, New York, NY, USA, 2016. ACM.