UNDERSTANDING CLOUD APPLICATION SECURITY VIA MEASUREMENTS

by

Liang Wang

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2018

Date of final oral examination: 12/07/18

The dissertation is approved by the following members of the Final Oral Committee:
Michael Swift, Professor, Computer Sciences, UW-Madison
Thomas Ristenpart, Associate Professor, Computer Science, Cornell Tech
Aditya Akella, Professor, Computer Sciences, UW-Madison
Kassem Fawaz, Assistant Professor, Electrical and Computer Engineering, UW-Madison

© Copyright by Liang Wang 2018
All Rights Reserved

To my family.

Acknowledgments

Many, many, many thanks to my advisors, Professor Thomas Ristenpart and Professor Michael Swift, for their mentorship and patience. I have been working with Tom for more than five years. He taught me how to identify an important research problem, systematically approach the problem, and work towards the optimal solution. He has spent enormous amount of time helping me improve my paper writing skills, and also giving me a ton of good advice on my pretensions Basically, I learned everything about research from Tom. His guidance and training changed me from a fresh graduate with zero experience in research to an independent researcher. And if not for Tom, I would not have had the chance to work with Mike, who is as amazing as Tom. Mike really cares about his students, and is always available and ready to answer all kinds of questions which are not limited to research. I am grateful for many insightful discussions with him, and always impressed by his attention to technical details and extraordinary knowledge in system research. I feel so fortunate to be co-advised by both of them.

I thank Professor Aditya Akella and Professor Kassem Fawaz for being my committee members, for taking the time to study my research and providing useful feedback. I have worked with Aditya on three projects, and I am also very appreciative of his scientific advice and knowledge.

I am fortunate to work with many talented researchers during my Ph.D, without whom none of my work would have been possible. Many thanks to Vincent Bindschaedler, Juan Caballero, Drew Davidson, Kevin P. Dyer, Alexis Fisher, Aaron Gember, Paul Grubbs, Keqiang He, Mengyuan Li, Jiahui Lu, Bilge Mutlu, Antonio Nappa, Rafael Pass, Ivan Pustogarov, Abhi Shelat, Thomas Shrimpton, Daniel Szafir,

and Yinqian Zhang. Special thanks to Professor David Cash and Dr. Gilad Asharov. Thank David for his help in my job search and his enormous contribution to our STRESS attacks project. And, our paper about blind CA will never get published without Gilad.

I would like to thank all my friends and colleagues, without whom my Ph.D life would be boring — Rahul Chatterjee, Adam Everspaugh, Peng Liu, Lei Kang, Venkatanathan Varadarajan, Yan Zhai to name a few.

I also thank all the people who have provided valuable discussions and insights about my research.

Thank my parents for their unconditionally emotional and financial support throughout my graduate career. Unlike most Chinese parents, they give me the freedom to make my own decisions about life and career. Special thanks to my wife, Xiuting Wang, for her constant support and encouragement, for countless sacrifices she has made for me, and for her love, care, and patience during the long Ph.D journey. I can't achieve anything without you.

Special thanks to Alice and Bob, who just want to have a normal conversation but unfortunately suffer so many attacks. Their stories help me and many other students learn cryptological protocols more efficiently. I wish they could live in a world with perfect security.

Contents

Contents iv

List of Tables vi

List of Figures viii

Abstract x

1 Introduction 1

1.1 *Machine Learning Based Attacks on Cloud-based Obfuscating Tools* 3

1.2 *Side-channel Attacks on Search Indexes of Multi-tenant Cloud Services* 7

1.3 *Characterizing Security and Performance of Serverless Computing* 11

1.4 *Outline* 12

2 Detecting Cloud-based Censorship Circumvention Systems 13

2.1 *Background* 14

2.2 *Trace Analysis Framework* 17

2.3 *Data Collection* 18

2.4 *Semantics-Based Attacks* 21

2.5 *ML-based Attacks* 27

2.6 *Evaluation* 30

2.7 *Entropy-Based Attacks* 35

2.8 *Estimating the Impact of False Positives* 42

2.9 *Conclusion* 44

3 Attacks on Shared Search Indexes in Multi-Tenant Cloud Services 46
3.1 Background 47
3.2 Survey of Multi-tenant Search Side channels 51
3.3 DF Side Channels in Enterprise Systems 55
3.4 Attack Goals and Notation 58
3.5 Subroutines of STRESS Attacks 59
3.6 STRESS Attacks 66
3.7 Case Studies of Modern Services 70
3.8 Countermeasures 85
3.9 Conclusion 89

4 Characterizing Serverless Computing Platforms 91
4.1 Background 91
4.2 Methodology 94
4.3 Serverless Architectures Demystified 95
4.4 Resource Scheduling 100
4.5 Performance Isolation 108
4.6 Resource Accounting 114
4.7 Conclusion 115

5 Summary and Future Work 116

Bibliography 121

List of Tables

2.1 A summary of campus network datasets and breakdowns of TCP flows by services. "TCP-other" are flows with non-HTTP/SSL/TLS protocols. "TCP-unknown" are flows of which protocols are failed to identified by Bro. "Deployed PTs" shows the Tor pluggable transports that had been deployed by the time we collected the traces. 20
2.2 Breakdown of PDFs by their categories. The percentages of all PDFs found are shown in parentheses. 22
2.3 Percentage of Alexa top 10 K servers that return a given type of response for each type of request (rounded to the nearest whole percent). The bolded entries indicate the standard response(s) for a given request type. 24
2.4 The effect of training and testing in the same or different environments. Reported is the average true-positive rate (average false-positive rate in the parentheses) across classifiers for all obfuscators using the dataset labeling the row for training and the dataset labeling the column for testing. 32
2.5 False positives of classifiers on the campus network datasets. The value in the parentheses is the false-positive rate of the selected classifier on a given campus network dataset. Recall that the number of flows tested for *OfficeDataset*, *CloudDataset*, and *WifiDataset* are 1.22 M, 7.48 M, and 5.32 M respectively. 33

2.6 Breakdown of the numbers of flows from our campus traces incorrectly labeled by our ML classifiers as the indicated obfuscator. The values in the parentheses are the percentage of flows labeled by Bro as the indicated protocol that represent false positives (e.g., 0.12% of HTTP flows are mislabeled as obfsproxy3 by our classifier). “Unknown” means Bro fails to identify the protocol of the flow. The total number of false positives across all protocols is shown in the final row. 33
2.7 A comparison of the average false-positive/false-negative rates (in percentage) of different tests for obfsproxy handshake message detection, across ten rounds of randomized validation/test splits. While the entropy distribution test (only shown for $k = 8$) was chosen in all ten rounds, we show the test-set performance of the other style of tests for comparison. All these tests are performed with the payload length check. 40
2.8 Summary statistics for DPI load (number of flows per second) and the slow-path load—the number of flows per second flagged as any obfuscator by our best-performing ML classifiers. 44

4.1 A comparison of function configuration and billing in three services. (*: We infer the OS version of GCF by checking the help information and version of several Linux tools such as `APT`.) 93
4.2 The average (over 10 runs) probabilities (as percentages) of getting N-way single-account coresidency (for $N \in \{1, 2, 3, 4,\}$ and $N > 4$, when launching 1,000 function instances in **Azure**. Here $N = 1$ indicates no coresidency among the functions. 103
4.3 Coldstart latencies (in ms) in AWS, Google, and Azure using Nodejs 6.* based functions for comparison. 105

List of Figures

2.1 Trace analysis framework 17
2.2 A comparison of true-positive (**left**) and false-positive (**right**) rates by features used. "E" indicates entropy-based feature, "T" the timing-based features, "H" the packet header feature set, and "EH" indicates a combination of entropy-based and packet-header features. Note that for clarity the graphs have truncated y-axes. 31
2.3 CDF of lengths (**left**) and entropies (**right**) of all URIs extracted from three campus network datasets. 41

3.1 A typical multi-tenant ES deployment consisting of several shards, and an example of inverted indexes and query filtering in ES. Documents from different users are in different colors. 52
3.2 Example relevance scores returned by the GitHub API when searching for the same term several times. **Left** shows the score variations in 60 seconds, and **right** shows the score variations in 2 hours. The time intervals for **left** and **right** are 2 s and 60 s respectively. *Y-axis does not start from zero*. 60
3.3 The changes of $\text{score}(t, d)$ as $\text{df}(t, D)$ increases. The scores when $\text{df}(t, D) = 1$ and $\text{df}(t, D) = 2$ are highlighted. Y-axis does not start from zero. . . . 77
3.4 An overview of the average relative and absolute errors for DF prediction for all n_{DFE} on GitHub. The first row targets estimation for a random 16-byte alphabetic string and the second row is for random 16-byte number. 77

3.5 The distribution of the absolute errors when $n_{DFE} = 50$ (GitHub). 78
3.6 The average false-positive rates for different lengths of alphabetic-character-only term and numeric-character-only terms across three shards in GitHub and Orchestrate.io. 79
3.7 Results of search quality experiment. MAP is "mean average precision". P@n is the precision only considering the top n documents returned for the search, averaged across all queries. Higher scores are better. 88

4.1 VM and function instance organization in AWS Lambda and Azure Functions. A rectangle represents a function instance. A or B indicates different tenants. 96
4.2 I/O-based coresidency test in **AWS**. 97
4.3 The total number of VMs being used after sending a given number of concurrent requests in **AWS**. 101
4.4 Median coldstart latency with min-max error bars (across 1,000 rounds) under different combinations of function languages and memory sizes in **AWS**. *Y-axis is truncated at 1,000 ms*. 104
4.5 Coldstart latency (in ms) over 168 hours. All the measurements were started at right after midnight on a Sunday. Each data point is the median of all coldstart latencies collected in a given hour. For clarity, the y-axes use different ranges for each service. 106
4.6 The median instance CPU utilization rates with min-max error bars in AWS and Google as function memory increases, averaged across 1,000 instances for a given memory size. 109
4.7 (a) CDFs of CPU utilization rates of instances (1,000 for each type) and (b) the median CPU utilization rates across a given number of coresident instances (50 rounds) in Azure, with min-max error bars. 109
4.8 Aggregate I/O and network throughput across coresident instances as concurrency level increases. The coresident instances perform the same task simultaneously. The values are the median values across 50 rounds. 110

Abstract

Cloud computing evolves rapidly and has inspired numerous novel cloud applications. In many cases, an application may share the same physical machine with other tenants' applications, and interact with third-party cloud services to gain on-demand resources or to achieve certain functionality. Such development and deployment patterns lead to security and privacy concerns, such as: Will the services an application depends on act as expected? Can using cloud features and services in particular ways cause security issues? Are tenants well isolated? However, existing research mostly focus on security issues in the underlying infrastructures of clouds. Known measurement tools are mainly designed for performance measurement, or have been deprecated due to the ever-changing cloud environment, or only work for popular applications. Therefore, one may not be able to use these tools to answer these questions. Overall, whether certain cloud applications can achieve their desired security and privacy guarantees, unfortunately, remains unknown.

We fill this gap, offering a set of novel measurement tools to facilitate future cloud research. We exercise these tools to examine various types of cloud-based applications in order to give preliminary answers to the above questions. First, we examine the detectability of Tor meek, a cloud-based traffic censorship circumvention tool, and find the way meek uses clouds produces unique traffic fingerprints. We develop decision-tree-based traffic analysis attacks against meek, and evaluate our attacks using a new trace analysis framework against a terabyte of real network traces. Our attacks only require 6-13 comparisons and little DPI state, but can reliably detect meek, as well as three other in-use anti-censorship tools, with high true-positive rates and sufficiently low false-positive rates (0.006% - 0.02%).

Second, we discover exploitable side channels that can potentially leak information about other tenants' documents in search indexes of modern multi-tenant cloud services. We develop the first practical attacks, called STRESS (Search Text RElevance Score Side channel), for mapping out the number of search indexes used by a cloud service, discovering sensitive terms in other tenants' documents, and estimating term popularity. Our attacks work efficiently on live cloud services such as GitHub. Finally, we conduct a large-scale measurement to characterize the architectures, performance, and resource management efficiency of three popular serverless computing platforms: AWS Lambda, Azure Functions, and Google Cloud Functions. We explain how the platforms isolate the functions of different tenants, which has important security implications. We characterize performance in terms of scalability, and resource scheduling efficiency, with highlights including that severe contention between functions can arise in AWS and Azure, which could make applications vulnerable to performance degradation attacks, and that Google had bugs that allow customers to use resources for free.

1 Introduction

More and more applications are utilizing public clouds for their benefits (e.g., flexibility, scalability and cost-effectiveness). For instance, a study used active DNS interrogation to show that 4% of the Alexa most popular one million websites [1] were using Infrastructure-as-a-Service (IaaS) clouds in some capacity [2]. The rapid evolution of cloud computing brings forth numerous novel applications, and completely changes the way applications are developed and deployed. In many cases, applications owned by different tenants may run on the same physical machines in clouds, being isolated by virtual machines or containers (multitenancy); and, applications interact with third-party (cloud) services to gain on-demand resources or to achieve certain functionality. Tenants spend less effort on many complex management tasks, as cloud providers will manage underlying infrastructures and provide performance, security, and privacy guarantees (e.g., access control, identity management, physical security) to their applications.

Despite cloud computing provides great benefits to modern applications, security and privacy concerns arise. For instances, as most of the cloud services are opaque, tenants do not know whether the services they depend on will act as expected. Besides, many applications are migrated to clouds without carefully considering the security and privacy implications of using particular cloud features and services. Whether peculiarities could arise due to the new environment and the new ways in which applications are built remains unknown. One particular

concern is about the security of multi-tenant applications. Multitenancy serves as a common practice for increasing resource utilization and reducing costs to both applications and their customers [3]. However, various kinds of resources, from physical infrastructures to software, could be shared among tenants in multi-tenant environments, raising the possibility of performance degradation and side-channel attacks. Specifically, poor resource isolation may enable a malicious tenant to perform side-channel attacks to steal other tenants' sensitive data from other applications, such as private cryptographic keys, and cause serious financial loss. Though numerous works specifically have studied cross-tenant side-channel attacks in IaaS [4, 5, 6, 7, 8] or Platform-as-a-Service (PaaS) [9] clouds, most of them focus on the isolation issues in the underlying physical infrastructures of clouds (e.g., CPUs). Other potential sources of side channels, e.g., resource sharing at application level, have not been thoroughly studied. Hence, we do not know if tenants are properly isolated and whether there exist unexpected ways that could introduce side channels in many multi-tenant applications.

Unfortunately, existing measurement tools, to the best of our knowledge, are not sufficient for researchers to study cloud applications to answer these questions. Cloud applications and environments change constantly, which could make measurement time-sensitive. Because of this, many measurement techniques for cloud research outdated (e.g., [4, 10]). Besides, as cloud applications are so diverse, measurement studies often focus on a few popular types of applications (e.g., cloud-hosting websites [2]). A large number of categories of applications have not received sufficient attention. And finally, measuring cloud applications is a challenging task because of the opaque, complex infrastructure and management ecosystems and dependencies behind cloud applications. Existing tools may only be designed for characterizing easy-to-measure aspects of applications (mostly, performance), rather than for getting a deeper visibility into how application work, and thus, are not suitable for security research. Overall, we need more measurement tools and techniques.

In our work, we fill this gap by developing novel measurement tools or techniques to facilitate future research on cloud security and privacy, and exercise these

tools to investigate security and privacy issues in various types of cloud applications from different perspectives in order to preliminarily answer the aforementioned questions. We start with examining the detectability of a cloud-based censorship circumvention tool, and realize that the way it uses clouds produces unique traffic patterns, making it vulnerable to traffic analysis attacks. We further investigate a set of multi-tenant cloud applications, and find exploitable side channels in their search interfaces. Finally, we examine the platforms for a new type of cloud computing service, and find their tenants could be vulnerable to performance degradation and side-channel attacks. More details are as follows:

1.1 Machine Learning Based Attacks on Cloud-based Obfuscating Tools

Nation-states and other Internet censors use deep-packet inspection (DPI) to detect and block use of censorship circumvention tools, which are used for bypassing network censorship. DPI enable censors to inspect network packet payloads to recognize the tools' protocol headers or other telltale fingerprints contained in application-layer content of network packets. In response, researchers and activists have proposed a large number of approaches for obfuscating the network protocol being used. These obfuscation tools can be loosely categorized as either attempting to randomize all bytes sent on the wire [11, 12, 13, 14] or attempting to look like (or mimic) an unblocked protocol such as HTTP [15, 16, 17, 18]. Examples of network obfuscators from each of these classes are now deployed as Tor pluggable transports [19] and with other anti-censorship tools [20, 14]. Currently, the available evidence indicates that existing DPI systems are easily subverted by these tools [15], and that nation-state censors are not currently blocking their use via DPI [21]. Thus these systems provide significant value against today's censors.

Particular approaches leverage the features or services offered by popular cloud providers. For example, CloudTransport tunnels traffic over HTTPs connections to a cloud storage service such as Amazon S3 [22], and GoAgent proxies are hosted in

Google App Engine [23]. Meek uses a technique called domain fronting to tunnel traffic over HTTPS connections to popular cloud load balancers such as Amazon CloudFront and Google App Engine [24, 25]. Meek is designed to be resistant to traffic analysis and thus be widely considered the hardest to detect by DPIs. IP or domain blocking are not effective in this case due to the dynamic nature of clouds. Blocking an entire cloud service could cause prohibitively high collateral damage for censors.

Can censors easily adapt and deploy new DPI algorithms that accurately detect these protocol obfuscators? Houmansadr et al. [26] proposed a number of attacks for detection of mimicry obfuscators, but they do not measure false positives. [1] It may be that their attacks are undeployable in practical settings, due to labeling too many "legitimate" connections as emanating from an anti-censorship tool. What's more, the randomizing and tunneling obfuscators, which are the most widely used at present [21], have not been evaluated for detectability at all. (Despite folklore concerns about possible approaches [24].) In short, no one knows whether these obfuscators will work against tomorrow's censors.

We provide the first in-depth, empirical investigation of the detectability of two variants of meek (meek using Amazon CloudFront and meek using Google App Engine), along with three obfuscation tools that are in wide use in Tor, including: obfsproxy3, obfsproxy4, and FTE [15, 27, 28]. Our results suggest that all of the in-use protocol obfuscation mechanisms can be reliably detected by our new machine learning (ML) based attacks or entropy-based attacks that require little in the way of payload parsing or DPI state. Particularly, the way meek uses clouds produces unique traffic patterns that make it more detectable.

For more systematical evaluation of our attacks, we built a trace analysis framework and exercised it with a variety of datasets. Prior attack evaluations [15, 26] have focused primarily on small, synthetic datasets generated by a researcher running a target tool in a specific environment, and evaluating true-positive and false-negative rates of a certain attack. This may lead to over-estimation of tool

[1]We consider the flows generated by censored applications and non-censored applications as positive examples and negative examples, respectively.

efficacy in practice: actual false-positive rates may be prohibitively large, and the synthetic traces used in the lab evaluations may be unlike the network traces seen in real environments. We address both of these issues by employing an experimental methodology that more closely reflects the setting of nation-state level DPI. This trace analysis framework can be used for evaluating other traffic analysis based methods, e.g., methods for detecting traffic generated by malicious applications in cloud environment.

We collected nearly a terabyte of packet traces from routers associated to various networks at our university campus. Together these have about 14 million TCP flows. Given the size of the covered networks (five /16 networks and three /24 networks), this represents a large, diverse dataset suitable for assessing false positives. We supplement with researcher-driven traces of obfuscation tools (which do not appear in the traces already), collected across a number of client environments.

Using these datasets, we explore previously proposed DPI-based attacks against obfuscation systems and develop new ones. To begin, we evaluate a collection of *semantics-based attacks*, which attempt to detect a mimicry obfuscator by looking for deviations from expected behavior of the cover protocol. This type of attack was recently introduced by Housmansadr et al. [26]. Next, we examine *machine-learning-based attacks*. These use decision trees that are trained from traces of both obfuscated and non-obfuscated traffic. Some of the features used were inspired by suggestions by the meek designers [24], but we are the first to build full attacks and evaluate them. Finally, we explore *entropy-based attacks*, which seek to detect when network packet contents (in whole, or in part) appear to be encrypted. encryption may be in whole, or in specific parts where non-obfuscated flows would not be. Our investigations reveal that:

- Semantics-based attacks can sometimes have prohibitive false-positive rates (up to 37% in the case discussed in §2.4.1). False positives here arise because many non-censorship tools deviate from standards in ways similar to mimicry protocols. In these cases they are unlikely to be useful by censors. We also

show that other semantics attacks, including one suggested in [26], have relatively low false-positive rates (0.03% as in §2.4.2).

- Randomizers such as obfsproxy [12], which only emit random payload bytes, are reliably detected by a combination of entropy-based tests and simple heuristics (e.g. length checks). In particular, these tests are applied only to the beginning of the first packet payload. This attack abuses the distinction between conventional protocols' plaintext headers and the lack of same in randomizing obfuscators. In short, having "no fingerprint" is itself a fingerprint (§2.7.1).
- Format-transforming encryption (FTE), as currently deployed in Tor, is reliably detected by simple tests on the entropy and length of the URI appearing in the first FTE-produced packet (§2.7.2).
- Tunneling protocols such as meek[24], broadly considered the most secure current proposal for protocol obfuscation, are reliably detected by classifiers trained on traffic-analysis and entropy features (§2.5). The trained classifiers are simple decision trees, and do require the DPI to maintain state for the first part of a flow.

In summary, our analyses show how to reliably detect all of the currently deployed Tor pluggable transports. For example, our ML-based attacks against meek achieve relatively low false-positive rates (less than 0.02%) and high true-positive rates (more than 98%), using decision tree and the first few packets in a flow for feature extraction. The resulting decision trees only require between 6 and 13 integer comparisons for evaluation, and thus can be relatively easy for censors to deploy.

1.2 Side-channel Attacks on Search Indexes of Multi-tenant Cloud Services

Modern cloud services provide full-text search interfaces to enable users to easily navigate potentially large document sets. Search systems such as Elasticsearch [29] and Solr [30] are both used by individual enterprises and offered as hosted services for other companies. Databases such as MySQL include similar search interfaces for columns containing unstructured text [31].

The canonical search API allows querying one or more keywords (or *terms*) to obtain an ordered list of matching documents. The response may additionally provide a real-valued score for each document. Which documents to return and their scores are determined using a relevance algorithm, most often term-frequency inverse-document frequency (TF-IDF) [32, 33] or one of its variants such as BM25 [34]. To compute these scores quickly, the search system maintains an inverted index that contains precomputed document frequencies for each term and term frequencies for each document-term pair.

Maintaining an index incurs overhead, and so best practice guides [35, 36] suggest configuring multi-tenant search systems to use shared indexes: each index is computed over (many) different users' documents. When a user issues a search query, the system first get all the documents that meet the search conditions, and then post-processes the results to filter out any documents the user should not have access to, and returns the remaining results. However, this filtering-based approach includes a side channel: one user may be able to determine the document frequency of a term, thereby potentially inferring if other users' documents include that term. This observation was first made by Büttcher and Clarke [37] in the context of local file systems. But to date no side-channel attack has been demonstrated exploiting the observation because doing so requires overcoming a number of significant challenges, including: the precise scoring functions used in real services are proprietary and unknown, a user's documents may be assigned to one of many possible indexes, API rate limiting, and more.

We provide the first treatment of logical side-channel attacks on modern multi-tenant search services. We begin by investigating representative open-source systems and assessing whether the basic document frequency (DF) side channel mentioned above exists. We setup local installations of systems including Elasticsearch/Solr and MySQL, following best practice guides for multi-tenant search. In all systems surveyed, we confirm that DF leakage can occur.

Despite this, and akin to early work on more well-studied side channels such as those based on CPU caches [38, 39, 40], it is not a priori clear how an attacker can exploit DF scores in realistic settings. In modern multi-tenant infrastructures, there exist a number of challenges: the precise scoring functions used in real services are proprietary and unknown, a user's documents may be assigned to one of many possible indexes, noise in relevance scores arises due to the number of files fluctuating frequently over time as users add or remove files, indexes may not remove keywords from an index even after a file is deleted, many APIs rate limit queries to search indexes, and more. It could also be, of course, that some sophisticated enterprise services do deploy proprietary countermeasures.

We develop STRESS[2] attacks, which consist of a multi-step methodology for exploiting DF side channels. Our attacks overcome the challenges mentioned and, ultimately, realize the first demonstrated cross-user side-channel attacks in this setting.

Our framework begins by providing three low-level tools that aid in attacks. First is a new approach that we call *score dipping*. By examining the changes in relevance scores, it provides a basic ability to infer, for a single index that includes an attacker document, whether there exists another document on the index containing a specific keyword. The insight is that an attacker can abstract away details of the scoring function, relying only on the assumption that scores decrease with increasing DF. Score dipping improves on prior ideas [37] for how to exploit the side channel because it can be used without precise knowledge of the scoring function used by a service plus, as we will experimentally show, it is robust to noise.

[2]**S**earch **T**ext **RE**levance **S**core **S**ide channel

In large-scale systems there will be a large number of shards across which an index is split, and score dipping alone is not effective in this setting. Each shard can be thought of as a logically isolated portion of the index, and a scoring function only takes into account documents assigned to the shard. In targeted attacks against a particular victim, attackers must have the ability to place one or more documents on the same shard as the target's documents. But the search service controls shard assignment, typically randomly load balancing new documents across them. Thus attackers are faced with an analogous issue to the co-location challenge that must be overcome in cross-user side-channel attacks in public infrastructure-as-a-service (IaaS) [4, 5, 6, 7, 8] or platform-as-a-service (PaaS) [9] clouds.

As a first step towards attacking a multi-shard system, we show how to use score-dipping to construct our second low-level tool, called *co-shard tests*, against multi-shard systems. Our co-operative co-shard test allows an attacker to determine if two attacker-owned documents have been assigned to the same shard. Specifically, we use score-dipping to build a covert-channel between different documents that are owned by the same user or different, co-operating users, and hence determine if they are on the same shard. This channel however does not on its own achieve co-location on a shard with a victim's documents, since the channel is only between attacker documents.

We next propose a new and different approach to obtaining co-location with a victims' documents, and in the process also learn about the service backend. Instead of trying to just obtain co-location with a target, we use our co-operative co-shard test to build our third low-level tool that we call a *shard map*: A set of documents in which each document is present on a distinct shard. We show that it is possible even on large-scale services to build complete shard maps, i.e., ones that appear to cover all shards used by the system. A complete shard map already reveals the number of shards, but more damagingly will be useful as a preliminary step for more granular attacks. We show how to do the following using a shard map:

- *DF estimation*: We can reverse-engineer each shard's unknown scoring function using a curve-fitting strategy. This yields a function that maps a term's search score to an estimate of that term's DF on a shard. This allows, among other things, trending: the ability to count the number of (private) documents mentioning a word. For example, if one knows an identifier used by a particular company using GitHub, our technique allows counting the number of private files they have stored on the service.

- *Brute-force term recovery*: We can use our shard map to test if a given term exists anywhere in the system, thereby allowing an attacker to brute-force recover moderately high-entropy values from victim repositories. While the side-channel attack does not reveal to the attacker which repositories contained the term, we propose scenarios that nevertheless allow the extraction of sensitive information such as credit card numbers, social security numbers, passwords, and more.

We evaluate the viability of STRESS attacks in practice with case studies of GitHub, Orchestrate.io, and Xen.do. As a sample of our results, we demonstrate on GitHub (in a responsible way, see discussion in §3.7) that one can build a 191-document shard map in 104 hours with a single account. We estimate that it would take about a day to brute force a space of 10^6 possible terms on every shard. For example, if one knows the bank identification number (BIN) and last four digits of a credit card number stored in a GitHub repository then the rest of the card's number can be brute-forced in under a day with 191 free accounts (c.f., [41] for discussion of credit card numbers and other information being stored on GitHub). We also discuss how stripping relevance scores (but still ranking documents) is likely to be inadequate.

We conclude by discussing potential countermeasures, suggesting in particular a new countermeasure which replaces actual document frequencies with ones trained from public data. We discuss the merits of this approach and routes to deployment.

1.3 Characterizing Security and Performance of Serverless Computing

Cloud computing has allowed backend infrastructure maintenance to become increasingly decoupled from application development. Serverless computing (or function-as-a-service, FaaS) is an emerging application deployment architecture that completely hides server management from tenants (hence the name). Tenants receive minimal access to an application's runtime configuration. This allows tenants to focus on developing their functions — small applications dedicated to specific tasks. A function usually executes in a dedicated *function instance* (a container or other kind of sandbox) with restricted resources such as CPU time and memory. Unlike virtual machines (VMs) in more traditional infrastructure-as-a-service (IaaS) platforms, a function instance will be launched only when the function is invoked and is put to sleep immediately after handling a request. Tenants are charged on a per-invocation basis, without paying for unused and idle resources.

Serverless computing originated as a design pattern for handling low duty-cycle workloads, such as processing in response to infrequent changes to files stored on the cloud. Now it is used as a simple programming model for a variety of applications [42, 43, 44]. Hiding resource management from tenants enables this programming model, but the resulting opacity hinders adoption for many potential users, who have expressed concerns about: security in terms of the quality of isolation, DDoS resistance, and more [45, 46, 47, 48]; the need to understand resource management to improve application performance [49, 50, 51, 47, 52, 53]; and the ability of platforms to deliver on performance [54, 55, 56, 57, 58, 59]. While attempts have been made to shed light on platforms' resource management and security [60, 61], known measurement techniques, as we will show, fail to provide accurate results.

We therefore perform the most in-depth study of resource management and performance isolation to date in three popular serverless computing providers: AWS Lambda, Azure Functions, and Google Cloud Functions (GCF). We first use

measurement-driven approaches to partially reverse-engineer the architectures of Lambda and Azure Functions, uncovering many undocumented details. Then, we systematically examine a series of issues related to resource management: how quickly function instances can be launched, function instance placement strategies, and more. We further explore how CPU, I/O and network bandwidth are allocated among functions and the ensuing performance implications. Last but not least, we explore whether all resources are properly accounted for, and report on two resource accounting bugs that allow tenants to use extra resources for free. Several security issues are identified and discussed.[3], including: (1) The lack of performance isolation in AWS between function instances from the same account caused up to a 19x decrease in I/O, networking, or coldstart performance, making functions vulnerable to performance degradation attacks; (2) Azure had exploitable placement vulnerabilities: a tenant can arrange for function instances to run on the same VM as another tenant's, which is a stepping stone towards cross-function side-channel attacks; (3) An accounting issue in GCF enabled one to use a function instance to achieve the same computing resources as a small VM instance at almost no cost.

1.4 Outline

This dissertation is arranged as follows: we introduce our work on ML-based attacks against cloud-based obfuscator, side-channel attacks against search indexes of cloud services, and security issues in three serverless computing platforms in Chapter 2, Chapter 3, and Chapter 4, respectively. In each of the chapter, we first introduce necessary background and related work in the *background* section, followed by the details of our results. We conclude and discuss future research directions in Chapter 5.

[3]We responsibly disclosed our findings to related parties before this paper was made public.

2 Detecting Cloud-based Censorship Circumvention Systems

Censorship-circumvention systems are designed to help users bypass Internet censorship. As more sophisticated deep-packet-inspection (DPI) mechanisms have been deployed by censors to detect circumvention tools, activists and researchers have responded by developing network protocol obfuscation tools. These have proved to be effective in practice against existing DPI and are now distributed with systems such as Tor.

In this chapter, we introduce our novel machine learning (ML) based attacks, entropy-based attacks, and a framework for evaluation that uses real network traffic captures to evaluate detectability, based on metrics such as the false-positive rate against background (i.e., non-obfuscated) traffic. We first exercise our framework to show that some previously proposed attacks from the literature are not as effective as a censor might like. We go on to show meek, a cloud-based obfuscator which is widely considered the hardest to detect by DPI, and the other three in-use Tor obfuscators (two variants of obfsproxy and FTE) can be reliably detected by our attacks with sufficiently low false-positive rates for use in many censorship settings.

.

2.1 Background

Network censorship. Nation-states and other organizations assert control over Internet communications by blocking connections to websites based on IP address, application-layer content of packets, active probing of remote servers, or some combination of the preceding. For a more fine-grained taxonomy of attacks see [26].

IP filtering is a commonly-used censorship technique. Here the censors monitor a list of IPs and block all communications whose source or destination IP that belongs to the list. IP filtering can be deployed in border ASes or local ISPs by nation-states censors [62]. To bypass IP filtering, a simple method is to use anonymous proxies. By deploying proxies outside the censored network, users within the censored network can submit traffic past censors to the (unfiltered) proxy IP. Various proxy-based censorship circumvention systems have been developed such as freegate, Ultrasurf and JonDo. Tor [63], an onion routing system, helps circumvent IP-based filtering due to its use of a large number of proxies (and bridges). A more recent proposal is to use domain fronting, in which one sends traffic through an unwitting reverse proxy [24].

In addition to filtering by IPs, and partly because of the burden of maintaining an exhaustive and accurate list of proxy IP addresses, censors have increasingly also deployed deep packet inspection (DPI) techniques. This enables censors to attempt *protocol identification*: inspecting the application-layer content of packets (e.g., application-layer headers) as well as packet size or timing information, in order to classify the traffic as generated by communication protocols associated with an anti-censorship tool. In the past, Tor traffic itself contained unique application-layer fingerprints patterns that can be recognized by DPIs [64]. In this work we focus primarily on DPI-based censorship.

A final class of protocol identification attacks uses active probing of a remote server. The Great Firewall of China, for example, is known to attempt Tor handshakes with destination IP addresses should a DPI test flag a flow to that IP as

possibly emanating from a Tor client [64]. We will consider an active attack briefly in consideration of filtering out a (passive) DPI test's false positives (§2.8).

Network protocol obfuscation. In response censors' efforts to carry out protocol identification, researchers have developed a number of approaches to protocol obfuscation. In large part, the goals have been to force DPI to misidentify flows of a censored protocol as those of a protocol that is not blocked, or to prevent DPI from recognizing the flows' protocol at all. The latter helps in the case that censors do not block unidentified flows. Suggested obfuscation techniques roughly fall into three categories:

- *Tunneling*: A logical extreme of mimicry is to simply tunnel data over a (typically encrypted) cover protocol. Intuitively this should provide best-possible mimicry as one is, in fact, using an existing implementation of the cover protocol. An example now deployed with Tor is meek, which uses domain fronting and tunnels traffic over HTTPS connections to popular cloud load balancers such as Google (we will refer to this as *meekG*) and Amazon (*meekA*). Meek is designed to defeat DPI-based attacks and considered as the hardest to detect.

- *Randomizers*: A randomizing obfuscator aims to hide all application-layer static fingerprints, usually by post-processing traffic with an obfuscation step that emits only bits that are indistinguishable from random ones. Examples are Dust [14], ScrambleSuit [13], and the various versions of obfsproxy [12, 65]. The last are currently deployed with Tor.

- *Protocol mimicry*: A mimicry obfuscator attempts to produce traffic that looks to DPI as if it were generated by some "benign" protocol, also called the cover protocol. One example of light-weight mimicry is format-transforming encryption (FTE) [15, 66], now deployed with Tor and implemented elsewhere [20]. It encrypts messages to produce ciphertexts that match regular expressions commonly used by DPI for identifying protocols. Less efficient obfuscators like Stegotorus [16], SkypeMorph [17], CensorSpoofer [18] and Marionette [67] use heavier steganographic techniques to produce messages

that look like a cover protocol. Marionette also provides mechanisms to mimic higher-level protocol behaviors, to perform traffic shaping, and to protect against some forms of active attacks.

As mentioned, several of these are now in-use with Tor as pluggable transports (PTs). A PT is just Tor's terminology for an obfuscator that works in their framework to obfuscate the traffic between Tor clients and bridges [19].

Unobservability. Houmansadr et al. [26] suggest that protocol-mimicry obfuscators will not foil future censors because they do not provide (what they refer to as) complete unobservability. They informally define the latter to be achieved only when a mimicry obfuscator faithfully follows the standards of the target protocol, in addition to imitating all aspects of common implementations. They give a number of DPI-based detection techniques for mimicry obfuscators and show that these have few false negatives (missed obfuscated flows) and high true positives (correctly identified obfuscated flows) using synthetic traffic generated by the researchers. Their attacks target semantic mismatches between obfuscated traffic and the cover protocol, for example checking for valid application-level headers for files, that header values such as length fields are correct, etc.

These semantics-based attacks have not, however, been assessed in terms of false positives: a flow that is labeled as having been obfuscated but was actually not generated by the targeted anti-censorship tool. A high false-positive rate could make such attacks less useful in practice, since it may lead to blocking too many "legitimate" connections or overwhelm systems performing additional checks after the DPI first labels a flow as obfuscated, as in the case of the Great Firewall's secondary active probing mentioned above. In fact, as we mentioned earlier, there are realistic settings in which a false-positive rate that seems small (e.g. 0.2%) may be troublesome for censors.

Thus a question left open by this prior work is: *Do the semantics-based attacks proposed in [26] have prohibitively high false-positive rates?* Clearly a negative answer would help us answer our main question above, but, as we will see, some semantics-based attacks do not work as well as a censor might like.

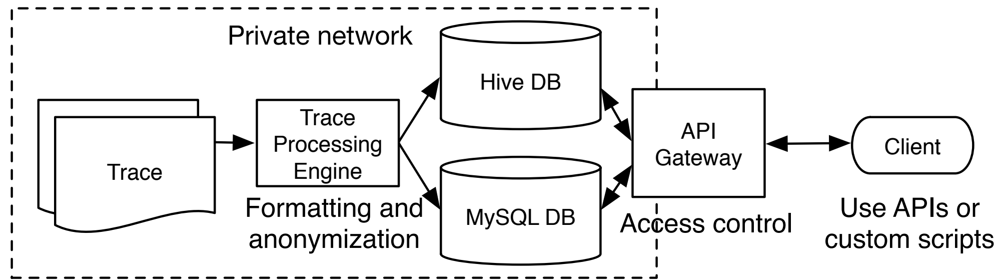


Figure 2.1: Trace analysis framework

2.2 Trace Analysis Framework

Prior evaluations of obfuscator detection techniques [26, 15] have focused primarily on small, synthetic datasets generated by a researcher running a target tool in a specific environment, and evaluating true-positive and false-negative rates of a certain attack. This may lead to over-estimation of tool efficacy in practice: actual false-positive rates may be prohibitively large, and the synthetic traces used in the lab evaluations may be unlike the network traces seen in real environments.

We instead employ an experimental methodology that more closely reflects the setting of nation-state level DPI. We implement a framework for empirical analysis of obfuscator detection techniques. It will leverage two groups of datasets: the first is a collection of network packet traces collected at various locations at our university at different points in time, and the second is synthetic traffic generated by target obfuscators. We use our packet traces to examine false positives and the synthetic packet traces to study true-positive rates of our attacks. (in fact, this framework can be used for evaluating previous proposed attacks as well.)

As shown in Figure 2.1, the framework consists of two main components: trace processing engine and API gateway. During trace processing, we use Bro 2.3.2 [68] with the "-r" option to analyze the collected network traces, and format and store the results into MySQL tables. Each table corresponds to a ".log" file generated by Bro, and it stores the information for flows of a given type (UDP, HTTP, SSL,

etc.). Each flow is assigned an unique flow ID, which is also generated by Bro. IP addresses in flows are anonymized.

Also, for each trace packet, we compute an MD5 hash to generate a packet ID, and store the packet ID, the flow ID of the associated flow, the raw packet content (in hexadecimal) and the packet timestamp into an Apache Hive database [69]. The usage of Hive facilitates the management and processing of terabytes of data. Users only need to query the MySQL database to get the basic statistics of a trace, while using Hive for more time-consuming, sophisticated analysis such as analyzing packet payloads.

We design a set of APIs to analyze the above data. These APIs are encapsulations of Hive or MySQL queries. For some simple analyses (e.g., counting the packets with a given keyword in the payload), pure Hive queries are enough. To facilitate more complex analysis and provide more flexibility, we leverage User-Defined Functions (UDFs) in Hive to allow users to provide their own mapper/reducer scripts [70].

Since network traces are sensitive data and need well-protection, it's reasonable to store the traces in the machines in a private network. A component that we call *API gateway* serves as the bridge between public networks and the private network. The front-end of API gateway is running on a public-facing machine and perform authentication and authorization on user requests (e.g., if a user could access a given trace or do certain actions like "delete data") based on predefined policies., while the back-end interprets user requests and perform corresponding operations on data.

2.3 Data Collection

We use two major types of datasets: (1) packet-level traffic traces collected at various locations in a campus network, and (2) packet-level traces for Tor obfuscators traffic collected in controlled environments.

Campus network traces. Over a period of 43 hours between Sep. 6, 2014 to Sep. 8, 2014, we monitored all packets entering or leaving a /24 IPv4 prefix and a /64 IPv6 prefix belonging to our university. This resulted in 389 GB of network traffic with full packet payloads. We call this dataset *OfficeDataset*.

Another dataset *CloudDataset* was collected between June 26, 2013 and to June 27, 2013 (over a period of 24 hours), and contains all traffic recorded between our entire campus network and the public IP address ranges published by EC2 and Azure. The third dataset *WifiDataset* constitutes all packets captured over a period of 12 contiguous hours in April 2010 from roughly 1,920 WiFi access points belonging to our campus. It contains data exchanged between all devices connected to the campus wireless networks and other (internal or external) networks. A summary of these three datasets is shown in Table 2.1.

The most recent trace could, hypothetically, contain flows corresponding to actual use of Tor with the obfuscators turned on. In our analyses, we ignore this possibility and assume that all traffic is non-obfuscated. Given that these flows are only used to assess false positives (FPs), our assumption is conservative when we argue the FP rates are low.

These traces contain potentially sensitive information of network users. We obtained an IRB exemption for these analyses. We performed analysis on an isolated cluster with no connectivity to the Internet, and with suitable access controls so that only approved members of the research team were able to use the systems. Only the bare minimum of research team members were approved to access the machines.

Tor traces. A Tor traffic trace captures the network traffic exchanged when a client visits a website over Tor configured to use a specific obfuscator. To collect a trace, we follow the procedures for collecting traces for website fingerprint attacks as described in [71]. We built a framework to automate these procedures. Our framework uses the Stem Python controller library for Tor, and the Selenium plugin for

	OfficeDataset	***CloudDataset***	***WifiDataset***
Collection year	2014	2012	2010
Deployed PTs	Obfs3/FTE/meek	-	-
Size (GB)	389	239	446
Total flow No. (M)	1.89	9.34	13.17
TCP flow No. (M)	1.22	7.48	5.32
TCP-HTTP (%)	37.0	73.6	76.6
TCP-SSL/TLS (%)	45.3	5.4	12.9
TCP-other (%)	0.2	0.2	0.1
TCP-unknown (%)	17.5	20.8	10.4

Table 2.1: A summary of campus network datasets and breakdowns of TCP flows by services. "TCP-other" are flows with non-HTTP/SSL/TLS protocols. "TCP-unknown" are flows of which protocols are failed to identified by Bro. "Deployed PTs" shows the Tor pluggable transports that had been deployed by the time we collected the traces.

automating control over a Firefox Browser when visiting websites [72, 73].[1] We record all traffic using *tcpdump* (or *WinDump* on Windows) at the same time.

Before and after visiting a website, our framework visits the "about:blank" webpage and dwells there for 5–15 seconds. The first time this is done is to ensure that the obfuscator connection is fully built; in our experiments, we found that most obfuscators forced a few seconds (usually less than 5 seconds) delay when building connections after Tor starts successfully. The second visit to "about:blank" is for making sure we can capture any lingering packets.

We collected three sets of Tor traces under different combinations of network links (with different capacities), end-host hardware, and operating systems, and labeled them as *TorA*, *TorB* and *TorC*. We collected *TorA* and *TorB* on Ubuntu 12.04 (32-bit) virtual machines (VMs) and *TorC* on a Windows 7 (32-bit) virtual machine. The Ubuntu VMs are built on the same image. All VMs run on VirtualBox 4.3.26 and are configured with 4G RAM and 2 virtual processors. The VMs for *TorA* run

[1] Also, our Firefox browser uses the exact same profiles as the default browser in the Tor Browser bundle (TBB) 4.06. The versions of obfuscators and Tor we used are also the same as those being used in TBB 4.06.

on a workstation and are connected to a campus wired network, whereas the VMs for *TorB* and *TorC* are run on a laptop and connect to a home wired network,

Each of these three datasets contains 30,000 traces collected as follows: (1) For each target obfuscator, we used our trace collection framework to visit Alexa Top 5,000 websites to collect 5,000 traces (labeled as *obfs3*, *obfs4*, *fte*, *meekG*, and *meekA*, corresponding to obfsproxy3, obfsproxy4, FTE, meek-google, and meek-amazon respectively); (2) In addition, we visited the same set of websites without Tor and obfuscators to collect 5,000 traces and labeled them as *nonTor*.

A handshake message of a flow is the application-layer content of the first client-to-server packet in the flow. We extract 5,000 handshake messages from each of obsproxy3, obfsproxy4, SSL/TLS, HTTP, and SSH flows to construct a new dataset. The first two types of flows are sampled randomly from Tor datasets and the other types of flows are from unused campus network traces (recall that we only use a part of the collected campus network traces to construct the campus datasets). We call this dataset *HandShakeDataset* and use it when examining attacks based (only) on handshake messages.

2.4 Semantics-Based Attacks

We seek to determine whether in-use obfuscators can be reliably detected by censors. The starting point is previously proposed attacks. As described in §2.1, Houmansadr et al. suggest a variety of attacks against mimicry obfuscators. For example, a Stegotorus client may generate invalid PDF documents, whereas legitimate traffic presumably does not. Their attacks therefore use the deviations of a target system from expected behavior as evidence for detecting mimicry obfuscators. We call these attacks *semantics-based attacks*.

In this section, we evaluate three semantics-based attacks, two proposed by Houmansadr et al. against Stegotorus and one suggested by Dyer et al. [15] for detecting FTE. None of these attacks have been evaluated in terms of false positives, and the latter has not received any analysis at all. Looking ahead, we find that the first attack is unlikely to work well in practice due to the high false-positive

	Standard	**Malformed**	**Partial**	**Other**	**Total**
OfficeDataset	26 (89.7)	0	3 (10.3)	0	29
CloudDataset	4,293 (62.8)	1,313 (19.2)	338 (4.9)	895 (13.1)	6,839
WifiDataset	1,860 (46.7)	1,252 (31.5)	572 (14.4)	295 (7.4)	3,979
Total	6,182 (57.0)	2,565 (23.6)	913 (8.4)	1,190 (11.0)	10,847

Table 2.2: Breakdown of PDFs by their categories. The percentages of all PDFs found are shown in parentheses.

rates we discover. The other two work better, but have deficiencies that our later attacks avoid.

2.4.1 Stegotorus PDF attack

Description. The Stegotorus HTTP obfuscator attempts to hide Tor traffic in commonly-seen documents, such as PDF and JavaScript. However, StegoTorus-HTTP does not guarantee the semantic correctness of the generated file. The authors in [26] proposed an attack that can detect files (more specifically, PDF files) generated by StegoTorus at a low cost and line speed. The key idea is to check the validness of the *xref* table in a PDF file.

Clearly the efficacy of this attack relies on the assumption that PDFs generated by non-obfuscated traffic, e.g., normal HTTP, indeed has valid *xref* tables. This was not evaluated in [26], presumably due to a lack of access to real traffic.

For an HTTP flow, we use a Python library pyndis to reassemble HTTP sessions. If the *Content-Type* in the response of a session is “application/pdf”, we assume the response body is a PDF file. We extract the content from the response, store the content as a PDF file, and use the PyPDF2 library to test the semantic correctness of the file. We define four categories of PDFs:

- Standard: According to [74], a standard PDF file should start with “%PDF” and end with “%%EOF”, and have a *xref* keyword in the content. The PDFs in this category have these keywords and also pass the PyPDF2 semantic check.

- Malformed: The PDFs have "%PDF", "%%EOF" or *xref* keywords (at least one keyword), but do not pass the PyPDF2 semantic check.
- Partial: A request may have a *Range* field that specifies the parts of a PDF file the client wants (e.g., only request the first 200 bytes of a file). The PDFs carried by the HTTP response of which status code is "206 Partial Content" is "partial PDF". The PDFs in this category will not pass the semantic check because they are only parts of the original PDFs.
- Other: All PDFs that are not in the aforementioned three categories. The PDFs in this category do not pass the semantic check.

We find in our tests these four categories are mutually exclusive. Only the PDFs in *Standard* can pass the PyPDF2 semantic check, and the PDFs in other categories could not pass the test for different reasons.

Results. We performed the test described above on the *OfficeDataset*, *CloudDataset* and *WifiDataset* university capture datasets. As shown in Table 2.2, the false-positive rate can be as high as 43% across all the datasets, should the censor mark any non-standard PDF flow as Stegotorus, as per the suggestion in [26]. More restrictive checks would still have high false-positive rates.

To explain these results, we observe that there are many reasons that non-obfuscator PDFs fall into *Malformed* or *Other* category. For instance, the file is encoded or encrypted, or there are bugs in the PDF generation software [75]. Partial content is also widely used by browsers and applications. As an example usage, FireFox will build several connections to fetch different parts of a single resource with range request in parallel, and reconstruct the resource by itself. The PDF content in each connection is incomplete, so a censor could not directly do semantics checks. A similar technique is called *Byte Serving* [76]. An application can retrieve the necessary portion of a file instead of the entire file. For example, a byte-serving-enabled PDF viewer can request and load a large PDF file from a file server page by page ("page on demand"). If a user only reads a few pages, the censorship system will never see the whole file.

Request	None	200	3xx	4xx	5xx	Other
GET long	3	12	< 1	**83**	2	0
GET non-existing	2	18	< 1	**79**	1	< 1
HEAD existing	2	**95**	< 1	2	1	< 1
OPTIONS common	3	**82**	< 1	13	2	< 1
DELETE existing	3	72	< 1	**21**	**4**	< 1
TEST method	4	66	< 1	**19**	**10**	< 1
GET wrong protocol	24	10	33	**28**	**5**	< 1

Table 2.3: Percentage of Alexa top 10 K servers that return a given type of response for each type of request (rounded to the nearest whole percent). The bolded entries indicate the standard response(s) for a given request type.

2.4.2 HTTP response fingerprinting attack

Description. In [26], the authors suggest that a censor can fingerprint a StegoTorus server by observing the server's reactions to different types of HTTP requests, since the StegoTorus server would respond differently from a genuine HTTP server. Of course HTTP servers in practice may have various implementations and it's possible that the HTTP servers used by real websites have the same HTTP-response-based fingerprint as the StegoTorus server. This would lead to false positives.

We reverse-engineered *httprecon*, the tool used for fingerprinting HTTP servers in [26]. We wrote our own version, a Python script that sends the same set of requests. One exception to this is that we omitted a particular request that is often viewed as an attack by server operators, as we used the scanner with public servers and therefore, could only send non-malicious requests.[2] We used our implementation to scan each of the Alexa top 10 K domains to quantify false-positive rates. For each target server, we stored its response header for each request into a MySQL database for analysis. We removed the servers that fail to respond to the *GET existing* request or respond with non-200 code, which may indicate the corresponding websites are down, and got 9,320 servers.

[2]This was not an issue in [26] as they did not apply their scanner to public servers.

Results. Table 2.3 shows the percentages of servers that return a given type of response for each type of request. The responses are put into six categories, with boldface indicating the *standard* response (as described in [26]) for a given type of request. Since *GET existing* requests always receive a "200 OK" for the servers examined, we remove the corresponding row from the table. We can see that for some types of requests, a majority but not all of the target servers return standard responses. Only 73 servers (0.8%) respond like a "standard" server.

Examining the response headers more closely, we observe that the target servers do not follow standards in various aspects (other than the response code):

- For a *GET existing* request, a server should set the *Connection* field in the response to *keep-alive*. We find of the response headers of all the servers, 1,547 (17%) don't have the *Connection* field, 6,503 (71%) are *keep-alive* and 1,126 (12%) are *Close*.

- For a *TEST method* request, a server should set *Connection* field to *Close*. However, we find 1,408 (15%) responses have no *Connection* field, 5,572 (61%) are *keep-alive*, 1,823 (20%) are *Close*, and 378 (4%) fail to respond.

- For an *OPTIONS common* request, a standard server should set the supported HTTP methods in the *Allow* line. We find 8,026 (87%) the responses don't have this field, only 844 (9%) return the standard supported methods (as defined in RFC), 45 (0.5%) return non-standard methods, and 266 (3%) fail to respond. We observe a total of 36 unique non-standard methods.

We find only three servers that have the same HTTP-response fingerprint as Stegotorus, suggesting that the false-positive rate of this active attack is actually quite low (or 0.03%).

We note that there are a total of 1,447 unique HTTP-response fingerprints. About 40% of these fingerprints are shared by more than 2 servers. The most-seen fingerprint is shared by 845 servers (9% of all the servers examined). Looking ahead, one might therefore update the Stegotorus server to have the same fingerprint as these servers, which will cause a false-positive rate of close to 10% and, moreover,

flag network flows associated with 5 of the top 100 domains and 74 of the top 1 K domains.

2.4.3 FTE content-length attack

Description. One possible approach for detecting FTE, as discussed by the authors of [15], is to check the correctness of the *Content-Length* fields of an HTTP message (HTTP request or HTTP response), since the HTTP message generated by the currently used version of FTE has an invalid *Content-Length* field that is mismatched with the real length of the content.

Results. For an HTTP flow, we use the same technique as we used in the PDF attack to reassemble HTTP sessions. Then, we check if the message has the *Content-Length* field in a message, calculate the length of the message body if it has the field, and compare the calculated length with the length specified in the *Content-Length* field. The false-positive rate is defined as the number of sessions with incorrect content length over the total number of sessions (in percentage). The false-positive rates are 1.86%, 1.95% and 3.5% for *OfficeDataset*, *CloudDataset*, and *WifiDataset* respectively.

Further examining the false positives, we find 34.2% of them are caused by early terminated connections, and 6.8% are caused by Transfer-Encoding fields (which trumps any existing Content-Length fields). More sophisticated checks could remove these false positives. There are other reasons for such mismatches, including extra control bytes added in the message body by some versions of web browsers; non-ASCII characters in the content-body, and bugs in web applications [77, 78].

2.4.4 Discussion

The semantics-based attacks we analyzed are relatively costly in terms of performance, because they require flow reconstruction (in the first and last case) or active probing (in the second case). The active probing and FTE attacks have arguably low false-positive rates, whereas the first PDF analysis has a clearly prohibitively large one. Based on our analysis of the three semantics-based attacks, we suggest that

semantics-based attacks should account for the noisy, non-standards-compliant nature of the web.

2.5 ML-based Attacks

Using machine learning (ML) for traffic classification is not a novel idea [79, 80, 81], but we are unaware of any prior work that has attempted its use in detecting obfuscated traffic. In this section, we develop attacks using ML with carefully selected features.

Our classification model will be binary, with output labels “positive” (obfuscated) and “negative” (not obfuscated). During training, we will label all training traces that were generated by a targeted obfuscator with “positive”, and all other traces as “negative”. Thus, we will have a classifier for each obfuscator that aims to distinguish a given obfuscator from all other traffic. When attempting to detect the use of any Tor pluggable transports, one could run all of our tests in parallel (one targeting each obfuscation method) and if any of them returns “positive” then the meta-decision would be that Tor with some pluggable transport is in use.

Before describing the actual classifiers, let us discuss the features that they will use, training considerations, and the details of how we evaluated their efficacy.

Features. Our ML-based attack will make use of three types of features. The first are *entropy-based features*. For a given flow, we calculate the Shannon entropy of every packet payload, and use the maximum/minimum/average entropies in packets in each direction (upstream or downstream) as features.

The second feature type is *timing-based features*, which are based on an observation regarding traffic patterns in meek. According to the documentation, meek clients will send packets to a meek server periodically, to check if the server has data to send. This results in timing patterns in the TCP ACK traffic of meek connections that differ from typical TLS connections [24]. To capture this as a feature, we calculate the intervals between two consecutive TCP ACK packets (in the same direction) in a flow, and group the intervals into 30 bins. Based on our observations

of meek traffic, the intervals are usually very small: 55% of them are less than 10 ms, and 99.9% are less than one second. So we limit the maximum considered ACK interval to one second. We use $(x, y]$ to denote a bin of width $y - x$, accepting all numbers r such that $x < r \leqslant y$. (All numbers are milliseconds.) Across our 30 bins, the widths are as follows: between 0 and 10 the bin width is 1 (i.e., $(0, 1]$, $(1, 2]$, etc.), between 10 and 100 the bin width is 10, between 100 and 1000 the bin width is 100, and the final bin is $(1000, \infty]$. We use variable-width bins to balance timing granularity and computational efficiency, and because the interval distribution is skewed towards lower values. Instead of using absolute numbers, our per-bin feature is the percentage of inter-ACK intervals that fall into a given bin.

Finally, we consider *packet-header features*. The first of these is the percentage of TCP ACK packets sent in each direction. This is motivated by the behavior of meek just discussed; we expect to see a larger-than-normal number of TCP ACK packets in meek flows. The second follows previous works (e.g. [82, 83]) by exploiting packet length as a distinguishing characteristic. For a given flow, we use the five most common packet-payload lengths (in each direction) as a feature.

Windowing strategies. We will have two main goals for our classifiers. The first is to accurately detect the obfuscated traffic under consideration. The second is to detect obfuscated traffic as soon as possible within a flow, which helps to minimize the amount of state required. So, to accommodate both goals, we allow the classifier to train over a window of flow traffic. Inspired by [84, 85, 86], we consider two windowing strategies:

1. The classifier inspects the first X packets of a flow (including SYN and ACK packets), and extracts features from these packets to do classification.

2. The classifier extracts features from the packets within the first X seconds of a flow to do classification.

Examining the distributions of durations and sizes (number of packets in a flow) of all obfuscator traces collected, and considering that we want the classifier to be able to make a decision as soon as possible, we limit the range of X in our tests as

follows. For the packet-count strategy, $X \in \{30, 35, 40, \ldots, 300\}$, and for the time-based strategy $X \in \{2, 3, 4, \ldots, 10\}$. Choosing the endpoints for the packet-count strategy, we can hope to classify between 90 and 99 percent of obfuscator flows before they terminate; simulating online classification, as opposed to after-the-fact. The same reasoning applies to the boundaries for the time-based strategy.

Details of classifier evaluation. Our initial measure of classifier performance uses the *TorA*, *TorB*, and *TorC* datasets. Recall that these include both synthetic Tor traces under each of the pluggable transports, as well as synthetic traces for SSL/TLS and HTTP.

For any given dataset, we perform nested cross validation with ten outer folds and ten inner folds. We measure performance for that dataset by taking an average over ten outer folds. In an outer fold, we perform a stratified random sampling of the target dataset to select 70% of the traces to create the test set, and use the remaining 30% for training and validation of classifiers for the inner folds.

Each inner fold is as follows: First, we perform a second stratified random sampling, taking a one-third for training and leaving the remainder for validation. Note that there are perhaps more standard choices for the split sizes, but we do not expect different choices to significantly impact results. The large test set size will tend to produce conservative estimations of our classifiers' performances.

Next, we fix a classification strategy by choosing: classification algorithm (K-nearest neighbor, Naive Bayes, or CART), windowing strategy (number of packets or time for each allowed X), and feature set (any single feature, any pair of features, or all features). There are 1,344 different classification strategies, in total. For a given strategy, we train five classifiers, one for each of the obfuscators. Each of these is tested on the validation set, and we record the average performance of the five classifiers. This gives an average measure for a particular classification strategy.

We pick the parameter combination that results in the classifiers that perform best on average across all inner folds. Then, using the "winning" classification strategy, we finally train a new classifier using *all* traces from the training/validation portion. This final classifier is what is tested on the test set.

As mentioned, we repeat the training and testing for 10 randomized 70-30 outer folds, and we will report the averages over these folds in a moment. In addition to reporting true-positive and false-positive percentages, we will report area under the precision-recall curve (PR-AUC) (c.f., [87]). A higher PR-AUC indicates a higher true-positive rate and lower false-positive rate. Specifically, a classifier with PR-AUC equal to one is perfect: it gives only true positives with no false positives. We calculate PR-AUC using the scikit-learn tool [88].

Looking ahead, we will sometimes also report on the average result of testing classifiers not chosen by the training/validation regime in order to understand the benefit of using some particular features.

2.6 Evaluation

We will first analyze the performance of ML-based attacks using the synthetic datasets, and then present the false-positive rates seen on the campus traces. To summarize, the best classifier achieves a high average PR-AUC (0.987), a high average true-positive rate (0.986), and a low average false-positive rate (0.003), across all five obfuscators as measured on the same synthetic data set it is trained on. The classifiers all perform significantly worse when tested on a synthetic dataset for which they were not trained. Finally, the highest false-positive rate of any classifier as measured on the campus datasets (none of which were available during training) is 0.65%.

Classifier parameters. Using *TorA*, we found that the best-performing classifiers were essentially always CART decision trees using the packet-count windowing strategy. The best classifiers used between 280 and 300 packets, which we consider too large for practicality. However, classifiers using up to 30 packets already perform within 0.3% PR-AUC of the best performing, and so we from now on restrict our attention to them.

Feature performance. We next discuss how the different types of features effect classifier performance. Recall that we used entropy-based, packet-timing, and

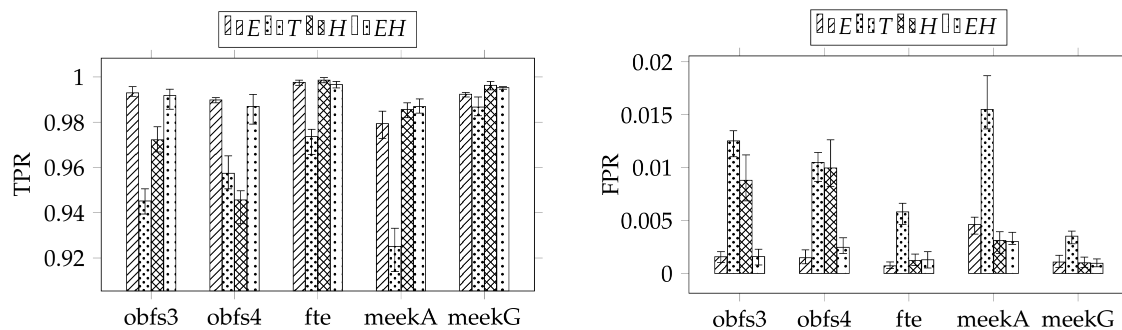


Figure 2.2: A comparison of true-positive (**left**) and false-positive (**right**) rates by features used. "E" indicates entropy-based feature, "T" the timing-based features, "H" the packet header feature set, and "EH" indicates a combination of entropy-based and packet-header features. Note that for clarity the graphs have truncated y-axes.

packet-header features. In Figure 2.2 we compare classifier true positive and false positive rates when testing specific combinations of features for the *TorA* dataset. We can see that using only entropy-based and/or packet-header features can already achieve high true positives and low false positives, with pretty low variance across folds (indicated by the error bars). Timing-based features showed a higher false-positive rate, and we conclude that a combination of entropy-based and packet-heading features performs best. Our training procedure indeed always selected the combination of the entropy-based and packet-header features.

Unexpectedly, entropy-based features work well for detecting meek. Examining the number of packets with non-zero-byte payload in the first 30 packets of the meek traces and the SSL/TLS traces in the Tor datasets, we find 70% of SSL/TLS flows have more than 18 packets with non-zero-byte payload, whereas at least 96% of the meek traces have less than 18 packets with non-zero-byte payload. So the number of samples for calculating entropy statistics in a meek trace and a SSL/TLS trace are often different, which biases the minimum, average, and maximum entropy scores used as features. This bias is caught by our classifier to differentiate between meek and SSL/TLS.

TRAIN\TEST	***TorA***	***TorB***	***TorC***
TorA	0.99 (0.002)	0.88 (0.01)	0.52 (0.02)
TorB	0.93 (0.009)	0.99 (0.002)	0.78 (0.03)
TorC	0.57 (0.12)	0.64 (0.12)	0.99 (0.002)

Table 2.4: The effect of training and testing in the same or different environments. Reported is the average true-positive rate (average false-positive rate in the parentheses) across classifiers for all obfuscators using the dataset labeling the row for training and the dataset labeling the column for testing.

Portability. We now turn to testing the "portability" of this ML approach. We used the two additional data sets *TorB* and *TorC*, which are collected in environments distinct from that of *TorA*. Whereas *TorA* is collected on an Ubuntu VM connected to a campus network, *TorB* is an Ubuntu VM connected to a home wireless network and *TorC* is a Windows VM connected to a home wireless network. We build three distinct classifiers using each of the three datasets using our procedure as above, but now augmenting the testing phase to also test against a stratified random sample of 70% of each of the other two datasets. The resulting matrix of average true and false positive rates (across all target obfuscators) is given in Table 2.4. The diagonal corresponds to training and testing on the same environment, whereas scores off the diagonal correspond to training and testing on different environments.

As can be seen, the ML classifiers do very well when trained and tested in the same environment. However, using the classifiers to attempt to classify a network flows generated by a distinct operating system and/or network significantly hinders performance. When using the same operating system, but different networks (the *TorA*/*TorB* and *TorB*/*TorA* entries) one sees less drastic reduction in performance. Changing operating systems however has large impact, with true positive rates being as low as 52% and false-positive rates reaching 12%. This provides some evidence that censors will indeed need to train classifiers on example traces representing all of the target types for which they need to be deployed.

PT\Dataset	*OfficeDataset*	*CloudDataset*	*WifiDataset*
obfs3	5,281 (0.43%)	14,714 (0.20%)	34,726 (0.65%)
obfs4	730 (0.06%)	16,257 (0.22%)	24,221 (0.46%)
FTE	6,437 (0.53%)	23,432 (0.31%)	19,857 (0.37%)
meekA	2,065 (0.17%)	787 (0.01%)	1,024 (0.02%)
meekG	837 (0.07%)	3 (0%)	2 (0%)
Total	0.98%	0.70%	1.40%

Table 2.5: False positives of classifiers on the campus network datasets. The value in the parentheses is the false-positive rate of the selected classifier on a given campus network dataset. Recall that the number of flows tested for *OfficeDataset*, *CloudDataset*, and *WifiDataset* are 1.22 M, 7.48 M, and 5.32 M respectively.

Protocol\PT	obfs3	obfs4	FTE	meekA	meekG
HTTP	20,415 (0.20%)	11,267 (0.11%)	43,680 (0.43%)	1,414 (0.01%)	0
SSL/TLS	30,482 (1.84%)	23,388 (1.41%)	3,784 (0.23%)	1,727 (0.10%)	391 (0.02%)
SSH	33 (1.94%)	0	33 (1.94%)	0	0
SMTP	198 (0.94%)	13 (0.06%)	73 (0.35%)	0	0
Unknown	3,593 (0.15%)	6,540 (0.28%)	2,156 (0.09%)	735 (0.03%)	451 (0.02%)
Total	54,721	41,208	49,726	3,876	842

Table 2.6: Breakdown of the numbers of flows from our campus traces incorrectly labeled by our ML classifiers as the indicated obfuscator. The values in the parentheses are the percentage of flows labeled by Bro as the indicated protocol that represent false positives (e.g., 0.12% of HTTP flows are mislabeled as obfsproxy3 by our classifier). "Unknown" means Bro fails to identify the protocol of the flow. The total number of false positives across all protocols is shown in the final row.

False positives in campus traces. We finally turn to analyzing false-positive rates of a classifier trained with the methodology and data above when faced with real, not synthetic, network traces. We first combine the *TorA*, *TorB* and *TorC* datasets and use the combination to train and validate a classifier. We perform stratified randomized splits of 30% for training and 70% for validation, train a classifier for each obfuscator, and compute the PR-AUC on the validation set. We repeat this procedure 20 times with fresh random splits, and select the classifiers with the highest PR-AUC. The PR-AUC of all the classifiers are greater than 0.90 (ranging from 0.91 - 0.99). The true-positive rates (false-positive rates) of the best/selected classifiers are 97.3% (0.08%),

98.3% (0.08%), 97.2% (0.13%), 97.0% (0.19%), and 97.8% (0.06%) for meek-amazon, meek-google, obfsproxy3, obfsproxy4, and FTE, respectively.

Next, for each TCP flow in the campus datasets, we extract the features using the first 30 packets, label it as negative, and then test it with each obfuscator classifier. A summary of the results is in Table 2.5. The total false-positive rate of the classifiers across all three datasets is 0.98%. As broken down by dataset, we see as low as 0.70% (*CloudDataset*) and as high as 1.40% (*WifiDataset*). Note that the total rates are not equal to the sum of the individual obfuscator false-positive rates, as some traces are falsely labeled by multiple classifiers.

Compared to the classifiers for other obfuscators, the classifier for meek-google produces a relatively small number of false positives, which is a total of 842 false positives out of 14 M flows. That the wireless network exhibits the largest number of false positives may be due to their noisier nature [89, 90]. For instance, when there are multiple TCP retransmissions in a flow, most of the packets we examined in the first 30 packets could be identical, hindering classification.

The false positives are associate with 12,551 distinct hosts, a small fraction of all monitored hosts, in the monitored networks. Meanwhile, they are only associated with 6,239 distinct destination IPs outside the campus networks. Less than 30% of these hosts or destination IPs are associated with more than 90% of the false positives. We find that a single IP outside the campus networks can contribute to as high as 4.6% of the false positives (as high as 1.2% for a single source IP inside the networks). This suggests that specific server-side or client-side settings could be the reasons for false positives.

We also determined the protocols of the false-positive flows. As shown in Table 2.6, most of the false positives are HTTP, SSL/TLS or *unknown* flows, according to Bro. The false positives of the meek-amazon classifier have a diversity in their protocols. We examine some of the mislabeled flows, and realize they may use a patched version of protocols such as SSH with encrypted handshakes [91].

We use nDPI to examine the flows labeled as unknown by Bro. nDPI fails to identify 13.5% of the unknown flows, and reports 45 protocols found in the remainder. Of these, 21 are built atop HTTP or SSL/TLS. These protocols account

for 78.2% of the unknown flows. Whether an ML approach can be enhanced to reduce false positives even further remains an open question.

2.6.1 Discussion

Our results show that trained classifiers using traffic-analysis-type techniques are effective at detecting the meek family of obfuscators. Our classifiers can distinguish the traffic produced by meek (and other obfuscators) from all other traffic. The true-positive rates are high, and the false-positive rates are relatively small.

When attempting to detect the use of any Tor pluggable transports, one could run all of our tests in parallel (one targeting each obfuscation method) and if any of them returns "positive" then the meta-decision would be that Tor with some pluggable transport is in use.

Though the training process of the ML-based approaches are complex, the decision trees emitted by training are, themselves, actually quite simple: evaluating the trees requires between 6 and 13 integer comparisons. These comparisons use per-flow state including just a small number of integer counters, for up to the first 30 packets. We therefore believe the trees themselves will be relatively easy to deploy, while the trickier issue will be the lack of portability we observed. This specifically implies that building good decision trees requires careful training in, ideally, the local network environment to which they will be deployed.

2.7 Entropy-Based Attacks

We explore other possible ways to detect target obfuscators and develop more efficient, entropy-based attacks against obfsproxys and FTE.

Entropy-based analyses have been used for various purposes such as randomness testing, network anomaly detection, and traffic classification [92, 93, 94, 95]. As traffic generated by the obfuscators are encrypted, one expects them to have noticeably higher entropy than conventional, unencrypted protocols (e.g., HTTP). Prior works show that entropy tests can be effectively used to detect and cull en-

crypted or compressed packets from network streams [96]. Their goal was to speed up DPI analyses by avoiding such opaque packets. One might expect that similar techniques could work here, but we will need to adapt them to our setting of obfuscator detection.

The obfsproxy methods directly apply encryption to every transmitted message, thereby “randomizing” even the first messages sent. On the other hand, conventional encryption protocols like TLS (or HTTPS) use handshake messages that typically contain unencrypted data from small, fixed sets of strings. Thus, the entropy of initial messages may provide a reliable way to distinguish obfsproxy messages from normal traffic. FTE also employs encryption from the start, although ciphertexts are designed not to look randomized. Nonetheless, the initial messages produced by the FTE obfuscators (as currently deployed in the TBB) are HTTP GET messages containing URIs that directly surface random-looking bytes from an underlying encryption scheme. So both the obfsproxy methods and FTE may admit detection by entropy-based tests on their initial messages. We explore this conjecture in a moment. First, let us explain the statistics we will use.

Shannon-entropy estimator. Let $X = x_1 x_2 \cdots x_L$ be a string of L bytes, i.e., each $x_i \in \{0, 1, \ldots, 255\}$ (where we map between bytes and integers in the natural way). Let n_j be the number of times that the value j appears in X, and let $p_j = n_j/L$. Then our estimate of the (byte-oriented) Shannon entropy is computed as $H(X) = -\sum_{i=0}^{255} p_i \log_2 p_i$. Notice that the maximum value of $H(X)$ is $8 = \log_2 256$, and this occurs when $p_i = 1/256$ for all i. That means that the payload string X contains every symbol in $\{0, 1, \ldots, 255\}$ an equal (positive) number of times. Furthermore, we note that a string with only printable ASCII characters will never have an entropy more than $6.6 = \log_2 95$.

The entropy-distribution test. The traditional Kolmogorov-Smirnov (KS) two-sample test provides a tool for deciding whether or not two sets of samples were drawn from the same distribution. We use it to help us detect high-entropy byte sequences. As above, let X be a string of L bytes. Fix an integer *block size* $k > 0$, define $B = \lfloor L/k \rfloor$, and write $X = S_0 \,\|\, S_1 \,\|\, \cdots \,\|\, S_{B-1}$, where each block S_i is length k bytes

(or possibly less for S_{B-1}). We will treat the sequence $H(S_0), H(S_1), \ldots, H(S_{B-1})$ as B samples from some unknown distribution D_k over entropy values. The second distribution we consider is the distribution of entropy values when the underlying strings are uniformly random. We denote this distribution by $\tilde{D}_k$, and a method for calculating $\tilde{D}_k$ is given in [96]. As shown, the distributions under consideration are parameterized by the block size k, whose value we will address later.

We compute the two-sample KS-test[3] for $H(S_0), H(S_1), \ldots, H(S_{B-1})$ for our *entropy-distribution test*, and an equal-length sequence of samples from $\tilde{D}_k$, and observe the p-value of the test. If the p-value is smaller than our chosen significance-level of $\alpha = 0.1$, then the null hypothesis that $D_k = \tilde{D}_k$ is rejected, and we conclude that the bytes are not uniformly distributed. Otherwise, the test does not provide statistically significant evidence for rejecting the null hypothesis, and we conclude that the bytes are uniformly distributed.[4] Our choice of $\alpha = 0.1$ was based on it being commonly used as a less restrictive (than 0.5) significance level in hypothesis testing in other settings. Higher α would label flows as uniform less often, lower α the opposite. Looking ahead, our experiments show that many choices of α around 0.1 would achieve the same obfuscator detection accuracy.

2.7.1 Detecting obfsproxy

We start by showing how to use the entropy-distribution test to detect that a network flow corresponds to one of the two obfsproxy methods. (Our tests currently make no effort to distinguish between obfsproxy3 and obfsproxy4.) As discussed above, we will make use of the observation that randomizers like obfsproxy3 and obfsproxy4 encrypt all bytes in the first message of a flow. Conventional encryption protocols such as TLS and SSH do not: they have fixed plaintext headers. We therefore apply the entropy-distribution test to the first 2,048 bytes of the first client-to-server packet's application-layer contents. If the length of a first-seen content is less than

[3]Using `scipy.stats.ks_2samp`[97].

[4]Technically, accepting the null hypothesis in this way should only be performed when a test has sufficient power. As we will see using the test in this way performs well, and so we dispense with a power analysis.

2,048 bytes, we will just test on all the bytes of the content. If the entropy-distribution test applied to these bytes indicates that they are uniformly distributed we flag the flow as obfsproxy.

First, based on the protocol specifications, the minimum payload length for obfsproxy3 is 193 bytes. For obfsproxy4 it is 149 bytes. Thus, we will check payload lengths for the first packet and require that the length be at least 149 bytes to be labeled as obfuscated. We will couple this check with an entropy-based test.

We experiment with the entropy distribution test described above for block sizes $k \in \{4, 8, 16, 32, 64\}$. We also considered two kinds of alternative tests. First is the byte uniformity test, in which we use a KS test with null hypothesis that the payload byte values are uniformly distributed (which is different than the entropy distribution test described above, where the null hypothesis is that the distribution of entropy values match closely those that one would get form a uniform distribution). We also consider the truncated sequential probability ratio test (SPRT) from [96] applied to payload byte values. This was used by [96] to detect "opaque" traffic (encrypted or compressed), and was shown to be more powerful than the (plain) entropy-test in their setting. Please see [96] for details of that test. We consider it for block sizes of 8, 16, and 32 bytes. This gives us a total of nine tests, the entropy distribution test with the five distinct block sizes, the byte uniformity test, and the SPRT tests with three different block sizes. In all cases we apply the payload length checks mentioned already.

To evaluate these tests we proceeded as follows. We use stratified random sampling (in our case we sample the same number of traces of each of the type of traffic) to select a 30% fraction of the traces from *HandShakeDataset* to construct a validation set and use the remaining 70% as a test set. Recall that *HandShakeDataset* consists of 5,000 handshake messages for each of obsproxy3, obfsproxy4, SSL/TLS, HTTP, and SSH flows, meaning that we have both obfuscated and un-obfuscated traces represented. We discuss false positives on the university data sets below.

We use the validation set to pick the best performing of the nine tests, and then analyze the test's performance on the test set. We repeat this procedure 10 times with fresh random splits of the test sets. In all 10 iterations the best-performing test

on the validation set is the entropy distribution test with block size $k = 8$. Table 2.7 shows the results broken down by protocol types and including for comparison the performance of some of the other tests on the test set. The results are strong: the entropy distribution test with $k = 8$ had only one false positive and zero false negatives in the worst case. We note that the payload length checks are critical here, we showed that SSL/TLS flows had a false-positive rate of roughly 23% on average when one omits the check. With the length checks in place, however, the p-values computed during the entropy distribution test for SSL, HTTP and SSH handshake messages are always near zero (a definitive rejection of the null hypothesis), whereas the median p-values for the flows declared to be obfsproxy is 0.99. Thus we consider the tests decisive.

False positive analysis using campus traces. The above analysis of false positives was in a closed-world setting with just the three protocol types (SSH/HTTP/TLS). We take the best test from above and evaluate false positives against the network traces seen from the university data sets. Specifically, we apply the entropy distribution test (with $k = 8$) on the first packet of every TCP flow in *OfficeDataset*, *CloudDataset*, and *WifiDataset* (assuming the packet is non-empty). We found 3,998 (0.33%), 19,247 (0.25%), and 12,786 (0.24%) false positives, respectively.

According to Bro, the 36,031 false-positive flows were distributed as follows: 18,939 were SSL/TLS, 9,873 were HTTP, and the remaining 7,219 flows were reported as "unknown". We applied nDPI[98], an open source DPI that can detect hundreds of application protocols, to the unknown flows. It was able to classify 1,673 (23.1%) of these as follows: 1,275 were Real Time Messaging Protocol (RTMP), 92 were SMTP, 14 were SSH, and the remaining were generated by other various applications.

The false-positive SSL/TLS flows account for 1.22% of all SSL/TLS flows examined, while the false-positive HTTP flows account for 0.10% of all HTTP flows examined. We find the SSL/TLS false positives are associated with 1,907 unique destination IPs, and 90% of these false positives are associated with only 685 unique destinations. Therefore, a large fraction of these false positives could be caused by specific server-side settings. Upon further examination of the destinations in the

	FNR		FPR		
	Obfs3	**Obfs4**	**SSL**	**HTTP**	**SSH**
Entropy dist. test	0	0	0.02	0	0
Uniform byte test	5.1	5.0	0	0	0
SPRT (8-byte)	60.2	58.3	30.6	0	0
SPRT (16-byte)	24.4	24.8	0.06	0	0
SPRT (32-byte)	6.6	6.9	5.3	0	0

Table 2.7: A comparison of the average false-positive/false-negative rates (in percentage) of different tests for obfsproxy handshake message detection, across ten rounds of randomized validation/test splits. While the entropy distribution test (only shown for $k = 8$) was chosen in all ten rounds, we show the test-set performance of the other style of tests for comparison. All these tests are performed with the payload length check.

latter mislabeled HTTP flows, we find just 357 unique servers. The URLs of these servers contain only printable ASCII characters but have the same entropy profile as a random string. The false-positive SSH flows all have encrypted handshakes, which suggests they may use obfuscated SSH [91]. The handshake message of RTMP consists of a 1-byte protocol version and a 1536-byte random string, causing our embellished entropy test to label all RTMP flows as positive.

2.7.2 Detecting FTE

As currently implemented in the Tor Browser Bundle, the first message in an FTE flow is an HTTP GET request. The URIs in these requests also have large Shannon entropies because their characters are direct encodings of ciphertext bits into bytes. Estimating the entropies of all URIs produced by FTE in 30% of traces (randomly sampled) in *HandShakeDataset*, we find a relatively narrow range of 5.5 to 5.8 bits. In *HandShakeDataset*, for non-FTE HTTP GET messages, the URIs always have entropies of less than 5.1 bits. This leads to a simple *URI-entropy test*: if the URI in the first GET request of a HTTP flow has entropy of at least 5.5 bits, then we label the flow FTE.

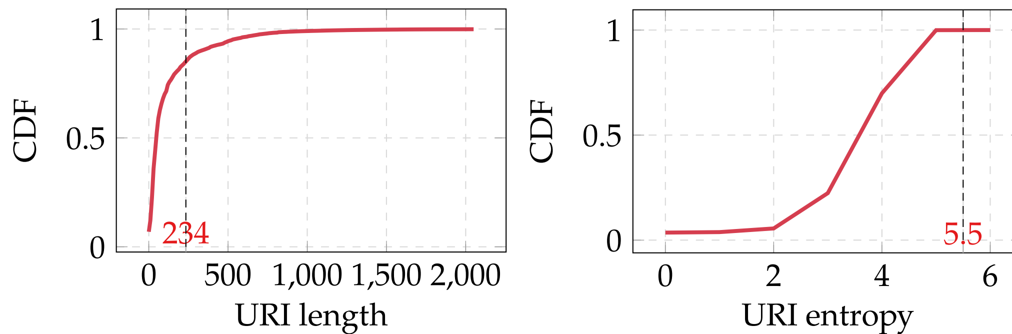


Figure 2.3: CDF of lengths (**left**) and entropies (**right**) of all URIs extracted from three campus network datasets.

Unsurprisingly, this test produces zero false negatives and zero false positives against *HandShakeDataset*. We also evaluated the URI-entropy test against *OfficeDataset*, *CloudDataset*, and *WifiDataset*. For an HTTP flow that contains GET requests, we extract the URI from the first of these, and perform the URI-entropy test on it. We found a total of 420,795 URIs are mislabeled as FTE, or about 4% of the roughly 10 M URIs examined.

To sharpen our test, we observe that the URIs in the HTTP requests generated by FTE have a constant length of 239 bytes. So we embellish our URI-entropy test so that positively labeled flows must also have a URI length of exactly 239 bytes. This significantly reduces the false-positive rate: only 55, 167, and 42 non-FTE flows are mislabeled as FTE in the *OfficeDataset*, *CloudDataset*, and *WifiDataset* datasets, respectively. That is a total of just 264 false-positives out of around 10 M samples, giving the embellished test a very small false-positive rate.

Note that the above tests can in fact be implemented without message reconstruction, since the tests only involve data in the first packet at fixed locations.

We note that [99] suggests using FTE URI length alone for detection. But we find that a length-only test causes about a 15% false-positive rate over the same 10 M flows. The CDFs of URI lengths and entropies are given in Figure 2.3.

2.7.3 Discussion

Our results show that entropy-based tests, embellished with simple length heuristics, can accurately detect obfsproxy3/4 and FTE with relatively low false-positive rates on real network traffic. White et al. [96] show that slightly simplified versions of our entropy-based tests can be implemented on commodity hardware, making them applicable at enterprise scale. The main simplification is that their tests make use of a fixed-size string, whereas ours use up to one full TCP payload. We have not yet considered the efficacy of our tests using truncated payloads.

We note that our entropy-based tests do not perform well at detecting meek-generated traffic, because meek tunnels over HTTPS. To detect meek one will need more sophisticated attacks, as shown in §2.5. Finally, we point out that the entropy-based tests in this section are far less resource-intensive than the ML-based attacks of §2.5. Those attacks in most cases require flow reconstruction, and these do not.

2.8 Estimating the Impact of False Positives

As we have emphasized, a critical component of successful obfuscator detection is achieving low false-positive rates and our work is the first that assesses them. Here we discuss the impact of false positives and whether the rates achieved by our detection approaches are sufficient for a censor's purpose.

Let us fix a particular representative scenario from our data, other scenarios can be analyzed similarly. Say a nation-state censor deploys the combined ML-based classifiers for each of the five obfuscators in the office environment, and assume a true-positive rate of 99% (as measured with *TorA*) and the false-positive rate of about 1% (as measured using the *OfficeDataset* dataset). For the foreseeable future, the base rate of obfuscator traffic will continue to be a tiny fraction of the overall traffic. Suppose, pessimistically for our analyses, that one out of every one billion flows is actually obfuscated traffic. Because of the low base rate, only about 1 in 10,000,000 flows marked as obfuscated will, in fact, be obfuscated. (Not 1 in 100 of marked

flows, as one falling victim to the base rate fallacy might assume.) This is likely to result degradation of Internet performance, as perceived by "legitimate" users.

Aggressive censors may be willing to cause such degradation. Of the blocked connections in this scenario, about 34.9% will be encrypted connections (TLS, SSH), and some censors have already demonstrated a willingness to block these. Iran, for example, has been known to block all TLS connections, or at least degrade their performance [100]. (Degrading bandwidth of such flows will tend to force Tor to fail, but not non-Tor TLS.) Other countries have mandated that ISPs prevent or limit use of encrypted protocols [101, 102]. Even so, the majority of flows erroneously labeled by our techniques as obfuscated are plain, unencrypted HTTP. It seems reasonable that blocking these will be less palatable for censors with sensitivity to collateral damage.

Some sophisticated censors have moved to a two-stage detection pipeline to improve accuracy. China's so-called Great Firewall first uses DPI to flag flows as potentially resulting from a censorship circumvention tool. Measurements suggest that they flag any TLS connection that has the same list of allowed ciphers as a Tor client as a possible Tor connection [64, 103]. The destination IP addresses of all such flows are then submitted to a distributed system that asynchronously performs an active probe of the remote IP address using a Tor client. If the remote IP completes a Tor handshake, then the IP is added to a blacklist. See [64, 103] for a detailed discussion of this behavior.

When using our obfuscator detection approaches in such a two-stage pipeline, the false-positive rate of the deployed approach will dictate the load on the more expensive, but more accurate, second-stage. (Perfectly accurate, active attack on Tor that was just described.) We refer to this second-stage as the DPI's slow path, and now turn to analyzing the slow-path load that would result from using our tests using the collected traces for simulation.

Table 2.8 shows a summary of the load seen by the DPI in terms of active flows per second as broken down by the various traces. For our purposes, a flow is active from the time stamp of its first packet to the time stamp of its last packet. We also report statistics regarding the load on the slow-path. A flow is labeled as going

	OfficeDataset	*CloudDataset*	*WifiDataset*
Avg. DPI load	14.6	138.5	182.2
Max. DPI load	362	1,042	1,580
Avg. slow-path load	0.08	0.60	0.77
Max. slow-path load	29	24	54
Slow-path active time	4.7%	38.2%	39.1%

Table 2.8: Summary statistics for DPI load (number of flows per second) and the slow-path load—the number of flows per second flagged as any obfuscator by our best-performing ML classifiers.

to the slow path whenever the combined ML-based classifier labels the traffic as obfuscated (i.e., any of the individual classifiers gives a positive label to the flow). As is shown, the maximum number of new slow-path flows per second is modest – in the worst case 54 are active in any given second– and the average is less than one. This suggests that, with minimal investment in the slow-path infrastructure, a censor could easily keep up with the false-positive rate of our obfuscation detectors. Of course, nation-state censors deal networks even larger than those considered in our work. Hence, caution should be exercised when extrapolating results from our setting to others.

2.9 Conclusion

We present the first comprehensive analysis of detectability of in-use network protocol obfuscators, as they are deployed in Tor. Our analyses reveal machine learning-based attacks can reliably detect meek, a state-of-the-art cloud-based tunneling obfuscator, and fast entropy-based tests for randomizer protocols and FTE (which is mostly randomized). We also show that some semantics-based detection tests suggested in the literature are less effective than a censor might like, due to the inherent long tail of non-standard network traffic. This suggests that future development of semantics-based tests should necessarily perform false positive analyses. Towards helping future similar types of research, we develop

a novel analysis platform for other researchers to systematically evaluate traffic analysis attacks in more realistic settings.

It is important to note that the detection techniques we explore can be, in turn, easily circumvented in almost all cases with simple updates to the obfuscator. This suggests that with the current state-of-the-knowledge on building practical obfuscators, anti-censorship tools will only have the advantage when censors remain ignorant of (or choose to ignore knowledge of) the details of their design. Building more robust, future-proof obfuscators that cannot be blocked by future, efficient DPI algorithms with knowledge of the obfuscator design remains an open question.

3 Attacks on Shared Search Indexes in Multi-Tenant Cloud Services

Full-text search systems, such as Elasticsearch and Apache Solr, enable document retrieval based on keyword queries. In many deployments these systems are multi-tenant, meaning distinct tenants' documents reside in, and their queries are answered by, one or more shared search indexes. Large deployments may use hundreds of indexes across which user documents are randomly assigned. The results of a search query are filtered to remove documents to which a client should not have access.

In this chapter, we show the existence of exploitable side channels in modern multi-tenant search. The starting point for our attacks is a decade-old observation that the TF-IDF scores used to rank search results can potentially leak information about other tenants' documents. To the best of our knowledge, no attacks have been shown that exploit this side channel in practice, and constructing a working side channel requires overcoming numerous challenges in real deployments. We nevertheless develop the first attacks, called STRESS (Search Text RElevance Score Side channel) attacks, and in so doing show how an attacker can map out the number of indexes used by a service, obtain placement of a document within each index, and then exploit co-tenancy with all other tenants to (1) discover the terms in other tenants' documents or (2) determine the number of documents (belonging to other tenants) that contain a term of interest. We demonstrate the attacks on

popular services such as GitHub in controlled experiments, and conclude with a discussion of countermeasures.

3.1 Background

3.1.1 Ranked keyword search

A fundamental information-retrieval task is finding relevant text documents for keyword search queries. Let D denote a corpus of text documents. For our purposes, a document consists of a bag-of-words representation; each word is a string that we refer to as a *term*. Our concern will be search systems that expose an API allowing keyword search, i.e., a client can execute a remote procedure call $\textsc{Search}(t)$ for a term t that returns an ordered list of documents $d_1, \ldots, d_n$ for some (typically fixed, small) n, sorted from most to least relevant. In addition to the list, many APIs also return relevance scores $s_1, \ldots, s_n$ for which $s_i \in \mathbb{R}$ indicates the estimated relevance of d_i to the query. A higher score indicates stronger relevance, and so $s_i \geqslant s_j$ for $i < j$. Many search routines allow more complex queries such as disjunctions and/or conjunctions of keywords, but we will primarily focus on single-term search.

This work will only consider unstructured document search in which documents have no semantic relationships. This distinguishes it from settings such as web or social network search.

In our unstructured search context, the most prevalent way of ranking is via term frequency/inverse document frequency (TF-IDF) scoring [32, 33]. Let D denote the document corpus, $N = |D|$ the number of documents, t be any term, and d be an arbitrary document in D. Define $\text{df}(t, D) = |\{d \in D \mid t \in d\}|$ to be the number of documents in D containing term t. This is referred to as the *document frequency (DF)*. We define the *term frequency* $\text{tf}(t, d)$ as the number of times the term t appears in the document d. We define the *inverse document frequency* by

$$\text{idf}(t, D) = 1 + \log \frac{N}{\text{df}(t, D) + 1} \,.$$

The TF-IDF score for the relevance of document d to the single-term query t is

$$\mathrm{score}_{\text{tf-idf}}(t, d, D) = \mathrm{tf}(t, d) \cdot \mathrm{idf}(t, D) \tag{3.1}$$

The TF-IDF score for a multi-term query $q = (t_1, \ldots, t_m)$ is

$$\mathrm{score}_{\text{tf-idf}}(q, d, D) = \sum_{i=1}^{m} \mathrm{score}_{\text{tf-idf}}(t_i, d, D) \tag{3.2}$$

We note that the $\mathrm{idf}(t, D)$ term is independent of the document d, and it is intuitively used to weight terms for multi-keyword queries.

There are many variants of the basic TF-IDF score that include other parameters and normalizing terms, and also alternative definitions of term frequency and inverse document frequency. Indeed, the live systems we experimented on used more complicated variants of TF-IDF, but we will use this simple formulation for the time being.

To implement TF-IDF scoring and search, a system generates an *inverted index*. For each potentially-searched keyword t, one stores $(t, \mathrm{idf}(t, D))$ at the head of a list of $(d, \mathrm{tf}(t, d))$ pairs. This allows fast computation of the TF-IDF and the documents that should be returned in response to the query.

TF-IDF scoring has many advantages and has intuitive probabilistic and geometric interpretations (c.f., [33]). However, in applications it is often useful to account for other factors in determining relevance, like the length of a document compared to the average length of all documents in the index. The BM25 scoring method incorporates this additional information [34]. As our eventual attacks will focus on TF-IDF, we omit the details and note that our attacks should extend to use of BM25.

3.1.2 Multi-user indexes and the DF side channel

More than 10 years ago, Büttcher and Clarke [37] pointed out a potential side channel when using TF-IDF scoring on multi-user indexes. A multi-user index is simply one generated over a document corpus D that includes files from different

users with different permissions. To perform a search on behalf of a user u, one uses the index to compute a ranked list of documents $(d_1, \ldots, d_n)$ with scores $(s_1, \ldots, s_n)$. Then one post-processes the list to redact documents (and their scores) not accessible by u, resulting in a smaller list $(d'_1, \ldots, d'_n)$ with scores $(s'_1, \ldots, s'_n)$ that are returned to u.

Büttcher and Clarke pointed out that systems like Apple's filesystem search service Spotlight are multi-user. While permissions models can be rather complex, we will focus our attacks on settings in which users should only be able to read the files they own, and no others.

In this context, Büttcher and Clarke show that $\mathrm{idf}(t, D)$ forms a potentially exploitable side channel that violates document confidentiality, even if a search index properly filters out search results on documents not owned by the user performing the search. This channel will allow an adversary to learn partial information about *document frequency*, so we call this the *DF side channel*.

To demonstrate their observation, consider an adversarial user Eve that wants to determine the number of documents that contain a term t^*. For example, it may be that Eve wants to learn whether another user Alice has a document $d_A = \{t^*\}$ stored on the system. Then, there is a simple attack exploiting the scoring function as a side channel.

Eve generates two documents $d_1 = \{t^*\}$ and $d_2 = \{r\}$ where r is some random term of length sufficient to ensure that it will not appear in any user document. Then Eve issues two search queries: First for $\textsc{Search}(t^*)$, which returns document d_1 with score s_1, and then for $\textsc{Search}(r)$ which returns document d_2 with score s_2. Even though $\textsc{Search}$ only returns results related to documents owned by Eve, Eve can anyway use s_1 and s_2 to infer information about other users' documents. By construction Eve has arranged that $\mathrm{tf}(t^*, d_1) = \mathrm{tf}(r, d_2) = 1$ and $\mathrm{df}(r) = 1$. Thus referring back to (3.1), Eve knows that

$$
\begin{aligned}
s_1 &= \mathrm{tf}(t^*, d_1) \cdot \mathrm{idf}(t^*, D) = 1 + \log \frac{N}{\mathrm{df}(t^*, D) + 1} \\
s_2 &= \mathrm{tf}(r, d_2) \cdot \mathrm{idf}(r, D) = 1 + \log N/2 \,.
\end{aligned}
$$

Thus Eve now has two equations in two unknowns and can solve for N and $df(t^*, D)$. The latter reveals how many documents in D contain t^*. Under the assumption that t^* would only appear, if at all, in d_A (e.g., because it is rather high entropy), then Eve can conclude that Alice's document contains t^*.

The attack as described requires scores, but Büttcher and Clarke detail another attack that uses only the order of documents returned by a multi-term search to approximately bound $df(t^*, D)$. They also mention that their techniques could be used to perform brute-force attacks, repeatedly using the side channel for different possible values for the target term t^*.

Büttcher and Clarke conjecture that this side channel could be used to recover information from real multi-user search indexes, but they do not demonstrate any working attacks. So while the DF side channel has been known to exist in theory since 2005, we are unaware of any investigation into its exploitability in practice, despite the widespread deployment of multi-user indexes. As we will discuss in the next section, there appear to be inherent challenges to building real attacks, including some noted by Büttcher and Clarke and others that we uncover related to distributed system design.

3.1.3 Other storage system side channels

Other side channels on search indexes and databases have been developed. Gelernter and Herzberg [104] show how to exploit a cross-site timing side channel to test for the presence of terms in a target search index. Our attack does not require malicious code injection, but does enable term extraction from a search index. Futoransky et al. use a timing side channel on insertions into MySQL and MS SQL databases to extract private information [105]. They observe that insertions take longer if a new virtual memory page is written, and use a divide-and-conquer approach to learn private terms. Their side channel is much harder to exploit than ours because it requires fairly high-precision timing measurements.

3.2 Survey of Multi-tenant Search Side channels

The basic DF side channel has only been discussed in theory, and it is unknown what search systems, if any, are vulnerable. We therefore begin by surveying existing open-source multi-tenant search systems, and experimentally confirm that the DF side channel exists in every setting we consider.

Elasticsearch. There are a few prominent systems for implementing full-text search on unstructured documents. Lucene [106] is a Java library which implements the building blocks of a search index, including functionality such as document tokenizers and query parsers. It also implements common data structures used for indexing. Elasticsearch (ES) and Solr are two libraries that implement sharding and cluster management for Lucene indexes. ES and Solr are widely used in industry due to their efficiency and scalability.

An architectural diagram of a canonical ES deployment is depicted in Figure 3.1. We assume a multi-tenant setting, in which multiple distinct user accounts have their documents indexed. In large deployments, a single server is insufficient to handle search queries, and so one instead builds separate indexes across multiple *shards*. A common way of load balancing across shards is to assign users at random to a shard, meaning all their files will be in that shard. Should individual users have many files, it may be needed to have more granular load balancing. For example, one can assign each individual document to a shard randomly when the document is uploaded, or there may be other logical groupings of documents. For instance, users may have multiple git repositories in GitHub, and as we will see later GitHub load balances across shards at the granularity of repository.

Lucene, Solr, and ES are all open-source projects, and typical configurations for the ranking function can be found online in forums [107, 108]. The default ranking used by Lucene (and so, in turn, by ES and Solr) is a variant of TF-IDF given by the equation

$$\text{score}_{\text{es}}(q, d) = \sum_{t \in q} \frac{\rho_{q,d} \cdot \beta_t \cdot \text{tf}(t, d) \cdot \text{idf}(t, D)^2}{\sqrt{|d|} \cdot \sum_{t \in q} \text{idf}(t, D)^2} \tag{3.3}$$

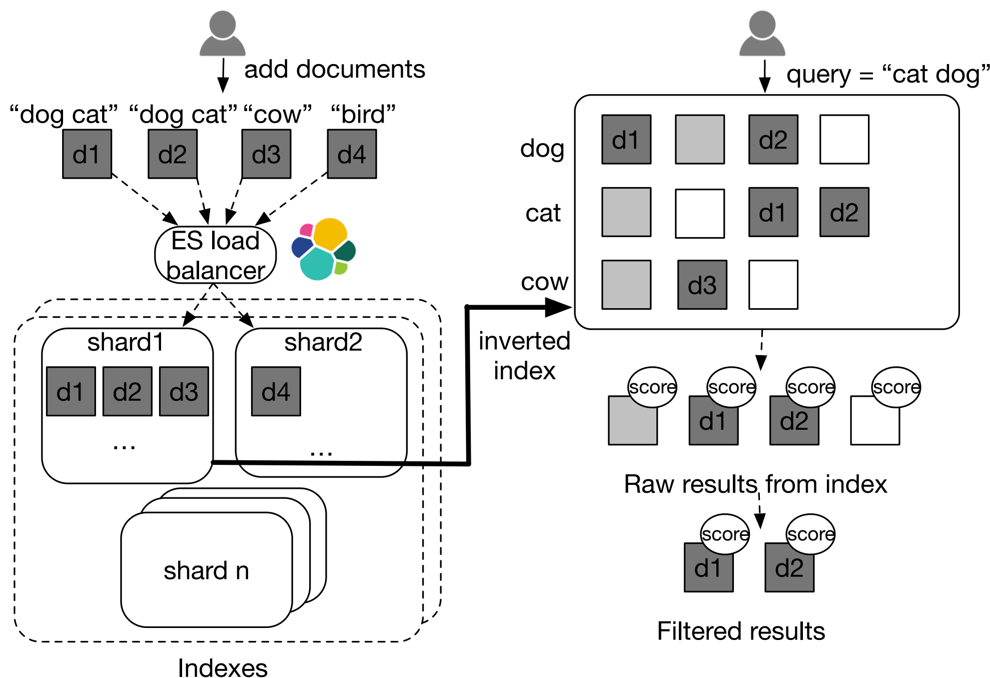


Figure 3.1: A typical multi-tenant ES deployment consisting of several shards, and an example of inverted indexes and query filtering in ES. Documents from different users are in different colors.

The *query coordination factor* $\rho_{q,d} = \sum_{t \in q} \mathbb{I}(t \in d)/|q|$ boosts documents that contain more terms matched by the query. It counts the number of query terms matching the document and divides by the total number of terms in the query $|q|$. The *per-term boost function* β_t allows customization of scores based on important application-specific terms. The division by $\sqrt{|d|}$ is what's referred to as the *field-length norm*, and it simply acts to normalize relative to the size of the document. In some configurations, the field-length norm is combined with an index-time field-level boost, which for our purposes would simply change β_t.

An attacker that retrieves a score $\text{score}_{\text{es}}(q, d, D)$ on their document d will know most of the terms in the right hand side of (3.3), with the only unknowns being the value N, $\text{df}(t, D)$ for each $t \in q$, and, if the configuration is unknown, the

boost function and other factors. When the configuration is known, this is just a (log-linear) equation in two unknowns. In this case the attack applies as in §3.1.

To test if ES has the DF side channel, we set up a local installation of ES version 2.3.4, and configure it to use one shard with zero replicas. We leave the other configuration options default. Following the suggestions provided by ES [36], we adopt the shared index strategy, create two tenants *alice* and *bob*, and add a *tenant-id* field to a document data structure to specify the document owner. A document data structure is a piece of JSON data that consists of three fields: a *tenant-id* field, a *name* to store the document name, and a *content* field to store the document content.

We implement and test two common search filtering mechanisms to enforce access control: filtering on *tenant-id* in the query [109] and filtered index alias [110]. The former excludes the documents that fail to meet the filtering conditions, e.g., excluding documents whose *tenant-id* ≠ *alice* for queries issued by the tenant *alice*. The latter works in the same way as the former, but it makes search filtering easier by allowing a user to create an alias name for a set of filtering conditions.

We first generated a unique term t, added a document $d_a = \{t\}$ as *alice* (the *tenant-id* of d_a is set to *alice*), and got a score $s_a = \text{score}_{\text{es}}(t, d_a)$. Then, we added a document $d_b = \{t\}$ with *tenant-id* = *bob*, and measured $s'_a = \text{score}_{\text{es}}(t, d_a)$. We observed that $s'_a < s_a$, and s'_a decreases as more documents that contain t are added by the tenant *bob*. Finally, we deleted all the documents associated with *bob*, measure again as *alice* to get $s''_a = \text{score}_{\text{es}}(t, d_a)$, and saw s''_a is the same as s_a. We observed the same results under different filtering mechanisms. These observations strongly suggest that one can infer if there are other documents containing a term by examining relevance scores; therefore, the DF side channel exists in ES.

MySQL. We set up a MySQL 5.6 server using its default configurations. In MySQL-based multi-tenant applications, a common design is multi-tenant-per-table, that is, storing all tenant's data in the same table, with a *tenant-id* field to distinguish each tenant's records [111]; then, to get the records only associated with a tenant *alice*, one can issue SQL queries with a condition *tenant-id* = *alice*. We achieve multi-tenancy in MySQL based on this design pattern. Our simple multi-tenant application uses one table, each record in which corresponds to a document. A record has the same

three fields as the documents in the ES tests. To enable full-text search in MySQL, we build a `FULLTEXT` index on the *content* field [31]. As a result, all the tenants share the same index.

We conducted the same tests as we do for ES, and observed the same result: the relevance score of a document for a given term will be affected by the documents that contain the same term, even if these documents are owned by other tenants. The result also suggests that the DF side channel exists in MySQL.

Other vulnerable systems. We found five vulnerable cloud-based search services using the similar methodology as in ES and MySQL. A cloud-based search service aims to provide scalable, easy-to-manage full-text search for web or mobile applications. An application can use it to build and maintain indexes on its data, and handle search requests. Such services usually charge the applications based on the amount of storage used or the number of requests processed. All of the systems we considered provide RESTful APIs and reveal relevance scores. Four of the services are built on ES (i.e., hosted-ES services), including AWS Elasticsearch [112], AWS CloudSearch [113], Searchly [114] and bonsai [115]. It's easy to confirm that they inherit the DF side channels from ES. We investigated these four due to their popularity, but there are many other hosted-ES services that could also have the vulnerability. One vulnerable system called Swiftype implements its own search engine [116].

Note that even if the side channel exists in a hosted search service, an application built atop that service will not necessarily have the DF side channel. For example, an application could conceivably assign each of their users to an independent index. However, due to the costs of cloud-based search services, application developers would typically prefer to use shared indexes. In Swiftype, a basic plan ($299 per month) only provides one index for usage, while a business plan ($999 per month) provides up to three indexes. In Searchly, a professional plan ($99 per month) offers 13 indexes. So if a multi-tenant application is built atop the service, the application's users will share the same indexes and might be vulnerable to information leakage. Looking at the case studies advertised by Swiftype [117], we realized some of them are indeed multi-tenant applications. We also noticed that Heroku uses Swiftype

and Searchly as its search add-on [118], suggesting the DF side channel might be inherited by Heroku-based applications.

Non-vulnerable systems. We also investigated PostgreSQL [119], CouchBase [120], crate.io [121], Searchify [122] (not to be confused with Search*ly* above), and Google App Engine [123]. Our experimentation suggests that these systems do not exhibit the DF side channel, primarily because they appear to use independent indexes for different tenants.

3.3 DF Side Channels in Enterprise Systems

In the controlled or partially-controlled settings above, we verified that the DF side channel was present. Enterprise search systems however introduce a number of complications, and it is at first unclear if the DF side channel can be exploited. In this section we discuss the major issues that must be addressed in understanding if such a system is vulnerable in practice.

Hidden relevance formula. An adversary may not know which TF-IDF variant is being used. The space of TF-IDF variants is large, with several different possible choices for $\mathrm{tf}(t, d)$ and $\mathrm{idf}(t, D)$ other than what we defined above, as well as different formulas for combining them to compute a score. These choices may use more features than we specified above, such as the length of the document. Scores for multi-term queries may also be computed via more complicated formulae and have constants that can be hand-tuned for a given application. Finally, we found that some services implemented scoring via ad hoc methods that took into account last-touched time or the order of terms in a query (i.e., treating the query (t_1, t_2) differently from (t_2, t_1)).

When the adversary does not know the scoring function it can no longer implement the algebraic attack from the previous section. This issue was cited by Büttcher and Clarke as preventing them from carrying out the attack on Spotlight. Instead, other techniques must be developed that are robust to variations in the scoring function.

Sharding. As mentioned above, enterprise search systems perform load balancing by dividing the document corpus into shards, which are essentially independent indexes. Sharding may be done per-document, per-collection, per-user, or via some other metric like creation time. Replica shards (i.e., copies of shards) are used to increase query throughput.

A side channel will only exist when scores are computed as a function of private documents, which usually means that an adversary's document must be on the same shard as victim data that it hopes to extract. Since search system interfaces do not expose information about sharding, this poses a further challenge for an adversary, who will need to arrange for its documents to be *co-sharded* with victim data, and also not be misled when documents are placed on shards without victim data.

Noise. The production search systems we experimented with displayed noisy behaviors that make attacks more difficult. For instance, in all of our experiments on live systems we observed that relevance scores constantly changed, and issuing the same search multiple times will result in different relevance scores on almost every query (see Figure 3.2 in §3.5).

Some of the noise is likely due to variations in the value N, the number of documents in the shard, which is changing constantly as many users write data to the index. This foils the algebraic attack of Büttcher and Clarke, because obtaining two scores computed with same value of N may be difficult or impossible, and anyway one cannot tell when this is the case.

Consistency and deletions. We also observed occasional larger changes in relevance scores likely due to other systems behavior. ES and similar systems have complex mechanisms for propagating newly written data into shards which maintain some form of consistency as segments of data are merged into shards. However, they do not maintain consistent relevance scores when data are merged, causing further difficulties for attacks that depend on fine-grained measurements in score changes. In some services it took up to two minutes for a change in a document to result in a change in relevance scores, slowing possible attacks. We also noticed

that searches may be issued in quick succession yet return greatly differing scores, likely due to a segment merge in between the queries.

Deletions are implemented lazily by marking documents for deletion and later expunging them via a background process (c.f.,[124]). The DF values are incremented quickly (i.e., after a minute) but apparently only reduced after expunging. Thus an adversary who hopes to delete documents as part of an attack is required to wait until its documents have been expunged before it can observe a change in the score function. Further complication arises because an adversary will not know which shard its document was present on and deleted from.

API restrictions. Search interfaces are rate-limited, both in terms of queries per time period (e.g., 5,000 queries per hour on GitHub) and their total size (e.g., 128 bytes to describe the keywords in the query). A very weak side channel may be mitigated if it requires an infeasible number of queries, or large queries.

Tokenizers and API interfaces often strip special characters, treating them as whitespace. So, for example, a hyphenated number XXXX-YYYY may be tokenized into two terms XXXX and YYYY, which have independent DFs. This affects the information available via the DF side channel.

Bystander data. An adversary who targets a victim or victims will need to carry out the attack in the presence of a possibly large number of *bystander* users whose data are uninteresting to the adversary. These data are not known to the adversary but will be used in relevance score calculations using a formula also unknown to the adversary. These bystanders are also actively writing and deleting data in the index.

The primary effect of bystander data in our work is their effect on *false positives*, which we return to below. The essential issue is that an adversary is *only* able to compute the DF of a term on a given shard. It is thus impossible to distinguish with certainty between cases where a victim document or a bystander document contains the term (and causes its DF to be non-zero). In some cases, we will argue that it is possible to use contextual clues, like the presence of other terms in the same shard, to limit false positives.

3.4 Attack Goals and Notation

We consider services that allow an adversary to write data and also use a search interface to retrieve relevant documents with scores. We assume that access control is implemented properly, meaning that other users' private documents are not returned to the adversary, or otherwise leaked directly through the interface.

Our adversary's intermediate goal will be to determine the DF of some given terms $t_1, \ldots, t_q$. (Later we will discuss attacks built on this capability.) We start from simplest case, where each $\mathrm{df}(t_i, D)$ is either 0, meaning no one has a document containing t_i, or is positive, meaning that it appears at least once. The document set D is changing constantly due to bystander activity, but we assume that the terms t_i are not written or deleted in a short period for the attack, and also that the size of the shard does not change dramatically.

In a system with a single shard, the DF of a term is well-defined. But in a multi-shard system, each term will have a shard-specific DF. To define the attack, we model the document set D as being partitioned into sets $D_1, \ldots, D_{n_{\text{SHRDS}}}$, where n_{SHRDS} is the number of shards. In this case, our adversary should determine, for each shard, the tuple $(\mathrm{df}(t_i, D_j))_{i=1}^{q}$ where the shard holds documents D_j. That is, on each of the shards, it should determine an estimate of DF of each term on that shard. We note that this attack will allow an adversary to detect that, say, t_1 and t_2 happen to occur together on the same shard, which is stronger than simply detecting that they occur somewhere in the larger system.

We fix a term-sampling algorithm RNDTERM that outputs a fresh random term that is assumed to never appear in bystander documents. In our experiments choosing a uniformly random 16-character alphabetic string was sufficient.

We also fix some notation for documents, terms, and the interface into the search service. Documents will be treated as sets of terms in our notation. In reality they are strings of text but the order of terms does not matter for scoring. We assume the service provides the ability to write documents, which we formalize as $\text{WRITE}(d, S)$, where d is a reference to a document and S is a set of terms. This operation will overwrite the entire document to consist of exactly S. Next, we will write $\mathrm{score}(t, d)$

to mean the score of document d returned by the service for a search for the term t (note that multiple calls to $\text{score}(t, d)$ may return different scores, and we are not fixing a document set D — $\text{score}(t, d)$ is defined according to the service's response).

3.5 Subroutines of STRESS Attacks

All of our attacks will be built on a fundamental property of all in-use relevance scoring functions we are aware of: As a term t becomes more common (i.e. its DF increases), the term weight decreases. In the case of basic TF-IDF, the term weight is $\text{idf}(t, D)$, but this is true for all of the variants we have encountered. Indeed it is intentional: A more common term should be given less weight in multi-term queries. An adversary that does not know the scoring function can still take advantage of this property.

3.5.1 Score-dipping

We first build what we call the *score-dipping* attack to determine if a term t appears somewhere in a shard of a system that uses an unknown scoring function. For now, assume that our attack owns a document d on the shard of interest. The attack first writes two terms t and r to d by invoking $\textsc{write}(d, \{t, r\})$, where r is a long random term (not present in another document). Then it requests searches for t and r. The search for t returns only d with some score s, and the search for r returns only d with some score s'. The attack checks if $s < s'$, and if so it guesses that t is present in the shard. If t was not present in the system originally, then after writing to d, we have both $\text{df}(t, D) = \text{df}(r, D) = 1$ (where D is the document set in the shard) and the searches should return the same score. But if t was already present, then $\text{df}(t, D) > 1$ and thus s should be lower.

To work on a real system this attack must be extended to tolerate noise in the scores returned. Due to bystander activity, we will observe differences in s and s' even when the terms t, r have the same DF. (Indeed just searching for the

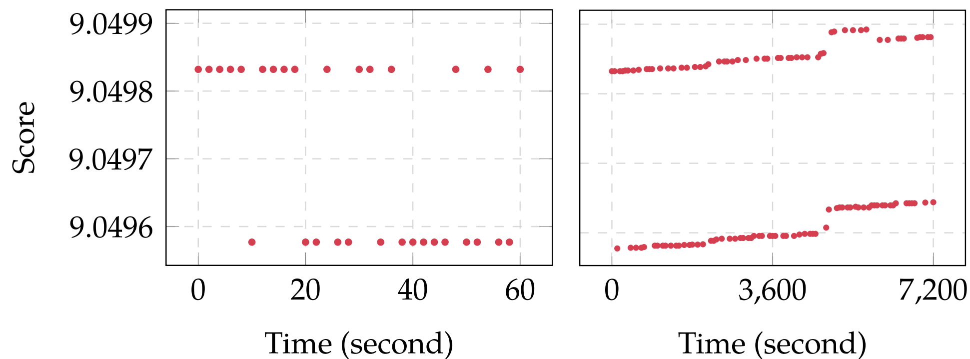


Figure 3.2: Example relevance scores returned by the GitHub API when searching for the same term several times. **Left** shows the score variations in 60 seconds, and **right** shows the score variations in 2 hours. The time intervals for **left** and **right** are 2 s and 60 s respectively. *Y-axis does not start from zero*.

same term twice will produce different scores. See Figure 3.2.) Bystander activity that incidentally decreases s or increases s' may cause the attack to output a false positive.

To mitigate this effect, we observed that the effect of changing a DF from 0 to 1 (or some larger number) caused a noticeably larger change in the relevance score than background bystander activity. In our attack, we perform several measurements on a shard to compute the typical variation when searching for terms with DF equal to 1 and 2. We can then determine a threshold for when the score is small enough to indicate that a term's DF is larger than 0.

3.5.2 Plan for the attacks

We now begin building towards an attack on a multi-shard system. In all multi-shard systems, the mapping of documents to shards is handled by some load balancing strategy that is not directly exposed in the interface. (To an outside user, sharding is meant to be transparent, though it does result in variation of relevance scores for the same query across shards.) Thus an adversary cannot directly see in which shards its documents reside, or directly control the shard on which a newly-created document is placed.

The hidden layer of load-balancing creates several difficulties. If we try to repeat the score-dipping attack several times without considering which shard we are on in each run, we will not know when we have explored all of the shards. Some systems might have hundreds of shards, and it may take a minute or more for a write to possibly change a relevance score. API rate-limiting can further slow naive attacks.

A more serious problem is that naively repeating the single-shard attack is not even a correct strategy when a service processes deletions lazily, meaning it only reduces DFs when expunging. In this setting, naive repetition will *detect its own documents*, which artificially increase the DF of terms of interest, during the attack. Concretely, suppose one stage of the attack writes a document $d = \{t, r\}$ containing the term of interest, and that deleting d, or removing t from d does not reduce the DF of t in the shard holding d. Then later stages of the attack that happen to return to the same shard will detect that the DF of t is non-zero, but this will be due to d and not victim documents.

Below we show how to mitigate the difficulty of attacking without deleting via a technique we call *shard mapping* that reverse-engineers the number of shards in the service and also places an adversary-controlled document on each shard. In addition to giving interesting information about a backend, shard mapping helps avoid the issues above, and also improves the efficiency of attacks.

We build two families of attacks using shard mapping: First we show how to quickly test for the presence of terms in other users' documents, allowing for what we call *brute force term extraction*. Second, we present a totally different approach called *DF prediction* that learns the scoring function on a shard and then attempts to predict DFs using the learned function.

3.5.3 Co-Shard testing

We first build a tool that we will use a sub-routine: *co-shard testing*. This will efficiently determine if an adversary-owned document d_1 resides on the same shard as another adversary-owned document.

Algorithm 1: COSHARDTEST

Input : Document d_1 and document set M
Output : True iff $\exists d \in M$: d_1, d on same shard
Parameter: Integer $\delta > 0$

1 $r \leftarrow$ RNDTERM; $r' \leftarrow$ RNDTERM;
2 WRITE$(d_1, \{r, r'\})$;
3 **foreach** $d \in M$ **do** WRITE(d, r);
4 SLEEP;
5 $s \leftarrow \text{score}(r, d_1)$;
6 $s' \leftarrow \text{score}(r', d_1)$;
7 **if** $(s' - s) > \delta$ **then** return True;
8 **else** return False;

Our strategy uses a technique similar to the score-dipping attack and the details are given in Algorithm 1. The routine COSHARDTEST takes as input references to document d_1, and a set of documents M (not containing d_1), and determines if d_1 was co-sharded with any documents in M (but not which one). It also uses a service-specific constant δ that we set by hand (once for each service). The attack starts by selecting two random terms r and r'. Then it writes r and r' to d_1, and only r to the other documents. After waiting for the writes to propagate, the algorithm issues two searches for r and r', and records the score of d_1 in the searches as s and s' respectively. Finally it outputs true if s' is greater than s by more than δ.

This attack works based on the principles described. If d_1 was not co-sharded with any other document, then we have $\text{df}(r, D_j) = \text{df}(r', D_j) = 1$, where D_j is the shard containing d_1. But if d_1 and one (or more) $d \in M$ are on the same shard D_j, then $\text{df}(r, D_j) \geqslant 2$ and $\text{df}(r', D_j) = 1$, resulting in a noticeable change in the score.

In this algorithm, most of the time is spent in SLEEP on line 4. This is why we have chosen a fast version of co-shard testing, where we can test if a new document d_1 was co-sharded with some $d \in M$ without spending extra time to determine *which* document it was.

The final detail is fixing δ, which must be set so that we distinguish larger changes in the score from random variation. We experimented with each service

by repeating several queries over a period of time, and setting δ to more than the maximum observed variation (see Figure 3.2 for an example of observed random variation).

Co-shard testing on stable services. On some services we noticed that searching for two terms with DF exactly 1 would return results with exactly the same score. On such stable services we can save time when co-shard testing many documents via the following strategy: create many documents, all containing the same random term r. Then request a search for r, which returns all of the created documents, and partition the documents returned by their scores. If the service is stable, the documents on the same shard will have the same scores, and otherwise their scores will likely differ. Thus our test can immediately filter to a subset of documents that are likely co-sharded, and then perform the co-shard test to verify correctness.

3.5.4 Shard mapping

Our multi-shard attacks will start with a pre-computation phase that we call *shard mapping*, which aims to place exactly one adversary-owned document on each shard in the system. We call a set M of documents with this property a *shard map*, and the goal of this subsection is to compute a shard map efficiently. Recall that this is non-trivial since the mapping of documents to shards is hidden by the interface. After this somewhat slower pre-computation set the adversary will be able to build attacks efficiently as we describe below.

Our method for computing a shard map M is as follows: Initialize a set M consisting of a single document d_1 (on some shard). Then create another document d_2, and use the co-shard test to check if d_1 and d_2 are on the same shard. If they are, then the attack discards d_2. If d_1 and d_2 are on different shards, then it adds d_2 to the map M. The attack continues creating further documents, except this time it tests for co-sharding with its documents in M before deciding that it has found a new shard and adding the new document to S. After some large number of runs that do not find a new shard, the adversary concludes that the set S consists of exactly one document on each shard of the system.

We denote our method by MapShards and it is given in detail in Algorithm 2. We repeatedly create a new document and test if it has landed in an "unmapped" shard using CoShardTest. If not, we discard the document. If, on the other hand, the document is on a new shard, then we add it to the map M.

Run-time analysis. In Algorithm 2 we assume a service-specific constant n_{MAX} has been fixed. We want to pick an n_{MAX} large enough to ensure that we eventually find every shard without wasting too much time in the attack, since each co-shard test requires a costly sleep to propagate writes.

To analyze the run-time we assume that each newly created document is assigned a uniformly random shard out of n_{SHRDS} possibilities. (Note that the actual shard assignment strategy being used by a target service could be more complex, so n_{SHRDS} estimated by MapShards would only be a lower bound.) Then the expected number of iterations before we have a document on every shard is given by the well-known *coupon collector problem* with n_{SHRDS} coupons (see for example [125]). A classic analysis tell us that the expected number of tries is close to $n_{SHRDS} \cdot (\ln(n_{SHRDS}) + 1.6)$, with tight tail bounds on deviations from the expectation.

Thus one can set n_{MAX} to be slightly larger than the coupon-collector prediction when one knows n_{SHRDS}, say, from technical information the service has released. Alternatively, one can simply guess n_{SHRDS} and run the attack until many iterations fail to find a shard. Let n_{FIND} be the number of shards found after k iterations, and n_{FAIL} be the number of consecutive iterations fail to find a shard after the kth iteration. The probability of seeing n_{FAIL} iterations of failing can be calculated as $(n_{FIND}/n_{SHRDS})^{n_{FAIL}}$. Then, one can stop if the probability is smaller than a certain threshold. We took the latter approach in our attacks.

Optimizations and multi-mapping. We also implemented a slightly more complicated, but faster, variant of shard mapping. In each iteration of the main loop on line 3, we changed the algorithm to create *two* new empty documents $d_j^{(1)}, d_j^{(2)}$ instead of one. We then execute a version of CoShardTest to test if either $d_j^{(1)}, d_j^{(2)}$ (or both) landed on new shards. If neither did, we discard them both. If exactly one did, then we keep it and discard the other. If both landed on new shards, then

Algorithm 2: MapShards

Parameter : Integer $n_{max} > 0$
Output : Shard map M

```
1  Create an empty document d_1;
2  M ← {d_1};
3  for j = 2, ..., n_MAX do
4  |  Create new empty document d_j;
5  |  if CoShardTest(d_j, M) = False then
6  |  |  M ← M ∪ {d_j};
7  |  else
8  |  |  Discard d_j;
9  |  end
10 end
11 return M
```

we must test if they landed on the same new shard via another co-shard test. In principle this could be run with more than two new documents in each iteration but we found that mapping was fast enough with two.

A second optimization is to apply the faster co-shard test when the service returns stable scores. On some services this will increase the speed of the mapping attack substantially.

Later we show that some attacks can be sped up using multiple shard maps $M_1, \ldots, M_m$, where the first documents of all shard maps lie on the same shard, and second lie on the same shard, etc. On some services like GitHub this will be easy to construct due to their sharding policy, which places all files from a repository on the same shard. On others we can run shard mapping multiple times, and then use co-shard testing again to find one document from each map that is on each shard.

3.6 STRESS Attacks

3.6.1 Attack 1: brute-force term extraction

We now build our brute-force term extraction attack using a pre-computed shard map M. Our attack will use M to quickly determine the DF of given terms on every shard of the system. More precisely, let B be a (potentially large) set of terms that we are interested in testing for, in a system with n_{SHRDS} shards. Our attack will return a tuple $(B_1, \ldots, B_{n_{\text{SHRDS}}})$ of sets of terms, where $B_i \subseteq B$ consists of the terms from B that are in the i-th shard of the system.

Our attack is given in Algorithm 3. It starts by initializing the sets B_i to be empty, and then iterates over each document in the shard map, writing a random term and all of the terms in B to the document. After waiting for the writes to propagate, it then tests for the presence of each $t \in B$ on the shards using score-dipping again with some threshold δ. This approach minimizes the number of costly sleep times by writing many terms to each file.

This technique crucially depends on the shard map to avoid incorrectly dipping the score for a term t with the attacker's own write operations. Also we note that if we have multiple shard maps then we can partition B and run independent instances of TermExtract in parallel.

3.6.2 Attack 2: DF prediction via score extrapolation

Natural extensions of our first attack to estimate DFs appeared to work correctly but were slow, as they had to measure and test if the DF was $0, 1, 2, 3, \ldots$ before finding the correct value. Our second attack estimates how many documents contain a given term on each shard of a search service (that is, we estimate $\text{df}(t, D_j)$ for each shard document set D_j). We call this *DF prediction* which is denoted as DFPred.

At a high level, DF prediction works by collecting data on the behavior of the score function when the DF of a term is known, and then training a model that predicts DF from relevance scores alone. In our attacks we can speculatively guess

Algorithm 3: TermExtract

```
    Input  : Shard map M = {d_1, ..., d_{n_SHRDS}}, term set B
    Output : (B_1, ..., B_{n_SHRDS})
    Param  : δ > 0
 1  Initialize all B_i ← ∅;
 2  for i = 1, ..., n_SHRDS do
 3  |   r_i ← RNDTERM;
 4  |   WRITE(d_i, {r_i} ∪ B)
 5  end
 6  SLEEP;
 7  foreach t ∈ B do
 8  |   for i = 1, ..., n_SHRDS do
 9  |   |   s_i ← score(r_i, d_i);
10  |   |   s'_i ← score(t, d_i);
11  |   |   if (s'_i − s_i) > δ then
12  |   |   |   B_i ← B_i ∪ {t}
13  |   |   end
14  |   end
15  end
16  return (B_1, ..., B_{n_SHRDS})
```

the class of scoring functions based on knowledge of common implementations, but we still assume that constants and custom modifications to the function are hidden.

The algorithm DFPred is described in Algorithm 4. It assumes it is given input several documents on the same shard of the service (either from several shard maps or from some other method). Then it performs a data collection step in the loop that estimates the score of a search when a term has DF equal to $1, \ldots, n_{\text{DFE}}$, where n_{DFE} is a parameter of the system (see Figure 3.3 for example data). After this step it uses a training algorithm to fit a curve f (from some class) that maps integers to reals. This f intuitively is a guess for the mapping from DFs to relevance scores induced by the system.

After computing f we can apply it in attacks. Given a term t of interest, an attack can write t to the document on the target shard and then search and record the

Algorithm 4: DFPRED

```
    Input   : Documents d_1, ..., d_{n_DFE} on same shard
    Output  : Score-to-DF model f
    Params  : n_DFE, training algorithm TRAIN
 1  L_scrs ← φ;
 2  for i = 1 ... n_DFE do
 3  |  r ← RNDTERM;
 4  |  for j = 1 ... i do WRITE(d_j, r);
 5  |  SLEEP;
 6  |  s_i ← Σ_{j=1}^{i} score(r, d_j)/i;
 7  |  Append {⟨i, s_i⟩} to L_scrs
 8  end
 9  f ← TRAIN(L_scrs);
10  return f
```

score as s. Finally, we produce an estimated DF by computing

$$[f^{-1}(s)] - 1$$

where $[x]$ denotes the closest integer to x. We subtract 1 to account for the document added by the attack that contains t.

Comparison to brute-force term extraction. Once we have computed the model f we can also use it for brute-force term extraction to get an attack with essentially the same complexity by using f to predict when terms have DF equal to zero. We opted for the first attack above because it does not require the training phase. Note that DF prediction actually recovers more, as it guesses the DF of a term rather than only detecting if the DF is non-zero. As mentioned above, using TERMEXTRACT to decide the DF of a term would be slow.

Algorithm 5: ROTermExtract

```
   Input    : Documents d1, d2, term set B
   Output   : Terms B′ ⊆ B present on the shard.
 1 foreach t ∈ B do
 2   | r ← RNDTERM
 3   | WRITE(d1, t)
 4   | WRITE(d2, r)
 5   | SLEEP
 6   | R ← ROSearch({t, r_i})
 7   | if d1 is ranked below d2 in R then
 8   |   | B′ ← B′ ∪ {t}
 9   | end
10 end
```

3.6.3 Attack 3: Rank-only term extraction

Our attacks above assumed that the search interface returns relevance scores. Some services however only return the list of ranked results without scores, and here we sketch how to adapt our techniques to this case.

We assume that the service supports multi-term search queries, and that the relevance scoring function assigns weights to terms that decrease with their DF. For now, we also assume there is no noise in the scores on a shard.

When there is no noise in scores, scores will often result in ties, and we start by reverse-engineering how the service breaks ties. In our experience this was done by sorting on the document name, creation time, or some other easily-noticed property of the documents.

Our rank-only term extraction attack is given in Algorithm 5. It takes as input two documents d_1 and d_2 on the same shard, and a target term set B. Without loss of generality we assume that d_1 is ranked higher than d_2 in the case of a tie.

The algorithm will compute the subset $B' \subseteq B$ of terms present on the shard with d_1 and d_2. The algorithm iterates over each $t \in B$. It writes t into the document d_1, and it writes fresh random terms r_i into the document d_2.

After waiting for the writes to propagate to the index, the attacker issues a two-term search query for $\{r_i, t\}$, which returns a ranked list R of two results. This list is either $d_1 > d_2$ or $d_2 > d_1$. If it is the latter, the algorithm infers that t is on the i-th shard and adds it to B'.

To see why this attack works, we consider the cases where $\mathrm{df}(t, D)$ is zero or is positive before the attack starts (where D is the document set on the shard). If it is zero, then d_1 and d_2 will have the same score and hence d_1 will be ranked higher. If however $\mathrm{df}(t, D)$ is positive, then d_2 will have a higher idf since the DF of r is exactly 1 and the DF of t is at least 2 after we write the files. Thus, d_2 has a higher score and be ranked first.

This attack can be generalized to test for several terms in each iteration of the main loop instead of 1 (thus reducing the number of sleep operations). This version requires several documents $d_1, \ldots, d_m$ on the same shard, and we assume that ties are broken in the order $d_1 > d_2 > \cdots > d_m$. The attack writes r into the last document d_m, and the terms of interest $t_1, \ldots, t_{m-1}$ in the first $m - 1$ documents $d_1, \ldots, d_{m-1}$. Then it issues a search query for $\{t_1, \ldots, t_m, r\}$ and looks at the position of d_m in the list. If a document d_i appears below d_m, then the attack infers that t_i appears on the shard for the same reasons as before.

3.7 Case Studies of Modern Services

In this section we discuss how an adversary might abuse the attacks we constructed in the previous section. Then we explore three services against which the score-based attacks are effective: GitHub, Orchestrate.io and Xen.do. We report on the performance of our attacks on the services, such as how long they took, how much they would cost to mount at scale, and how often they might fail. Finally, we examine rank-only attacks against GitHub and Orchestrate.io, who provide web interfaces for performing multi-term search without returning relevance scores.

3.7.1 Scenarios

To understand possible threats let us abstract the ability that is implied by our brute force term extraction and DF prediction attacks. Let the document sets stored on the shards of the service be $D_1, \ldots, D_{n_{\text{SHRDS}}}$. Term extraction gives us abstractly an oracle $\mathcal{O}_{\text{TE}}$ that takes input (t, i) and returns 0 if $\text{df}(t, D_i) = 0$ and otherwise returns 1. DF estimation provides a richer oracle $\mathcal{O}_{\text{DF}}$ that takes the same inputs (t, i) but returns (approximately) $\text{df}(t, D_i)$ itself.

In this view any abuse will have some fundamental limitations. Short terms are likely to appear by chance in documents, so the first oracle will likely return 1 most of the time. Also, since terms are extracted by a tokenizer, if some text happens to contain periods or hyphens (like a URL, SSN, phone number), then the text will be separated into small terms which may have high false positive rates. Neither of these oracles allows one to test for substrings of terms, so very high-entropy terms like cryptographic keys are, without some side information about them, intractable to guess. We nevertheless identify two types of attack scenarios that are possible within these limitations.

Medium-entropy terms. The first is brute-forcing *medium-entropy terms* that are rare enough to avoid false positives yet drawn from a brute-forcible space. As examples of sensitive medium-entropy data that may be stored within a search service, consider SSNs and phone numbers in the United States. In these cases, and assuming no hyphenation is used (which is desirable for search), an adversary could in principle use the first oracle $\mathcal{O}_{\text{TE}}$ to produce a list of all such numbers and SSNs stored in the shards of the service. This is already a severe violation of the confidentiality expected by users.

A second type of medium-entropy data are (relatively strong) passwords. Note that very weak passwords such as "123456" are likely to generate false positives. An attacker may test a dictionary of common passwords (or their hashes) using $\mathcal{O}_{\text{TE}}$ to determine which ones occur in the services' document set of terms. Passwords could be stored in search services when used as application backends, and there have also been well-publicized incidents of passwords being stored on GitHub

repositories. In either case, an attacker could use access to $\mathcal{O}_{\mathrm{TE}}$ to filter the password dictionary to a smaller set that it then uses for online password guessing attacks.

Medium-entropy data targets may also arise when an adversary has partial knowledge on an *a priori* piece of high-entropy data. For instance, someone may store documents that contain terms with adversary-known high-entropy prefixes followed by lower entropy suffixes. The prefixes will lower or remove false positives, allowing for brute-forcing of the rest via the oracle $\mathcal{O}_{\mathrm{TE}}$.

A final type of medium-entropy data may occur when high-entropy data is tokenized into medium-entropy terms. Consider a hypothetical 24-character API key that consists of four 6-character chunks separated by hyphens. These may be tokenized into 6-character terms that could then be found via the oracle $\mathcal{O}_{\mathrm{TE}}$, along with some false positives. This would vastly reduce the space of possible API keys for an attacker who only needs to try keys formed by combinations taken from the set of 6-character terms found in the index.

Term trending. A second class of attacks uses the richer ability of the $\mathcal{O}_{\mathrm{DF}}$ oracle to estimate DFs rather than simply detect if they are positive. Unlike the previous settings, an attack may profitably query $\mathcal{O}_{\mathrm{DF}}$ for even low-entropy terms to learn about how commonly they are included in documents. For instance, on GitHub, one can use the side channel to learn about the popularity of certain libraries or packages. Or, if separate source code documents include unique identifiers associated to a particular victim (e.g., AWS account IDs), then the $\mathcal{O}_{\mathrm{DF}}$ oracle can be used to count the number of documents in that victim's private repositories. Since we are able to extract per-shard DF estimations, an adversary may be able to guess if it has found the shard of a particular user by looking for a shard that contains, with high DFs, terms associated with that user. One can then focus searches on that shard in order to reduce false positive rates in, e.g., a brute-force attack.

3.7.2 Performing Responsible Experiments

We would like to validate the feasibility of attack scenarios as discussed above. However, the nature of the side channel is such that we could, if careless, end up

spying on actual user data in these services (e.g., if we simply started querying for passwords). We therefore took care to ensure that our experiments would not expose private information about their users or otherwise cause undue burdens on services.

Our experiments will only target simulated victims, i.e., accounts under our control with documents that we generate. This will give us ground truth. Except for estimating false positive rates, we apply the DF side channel only to long, random unstructured terms that are exponentially unlikely to appear in any bystander's document (given the number of such terms we use the side channel upon). Put another way, we explicitly avoid learning anything about other users' data from the side channel. We refer to users other than our simulated attack and victim users as bystanders (i.e., everyone is a bystander except for our accounts).

False positive rates caused by bystander data are important for understanding the efficacy of possible attacks, as we expect false positives to be a significant limitation to the attacks in practice. The rates however depend on bystanders' potentially private data. We therefore perform carefully limited false positive measurements in which we infer only whether or not we get the right answer from our side channel for random terms of given lengths. Even here we minimize any perceived risk to other users, only searching for random unstructured data with no semantic value. We only report summary statistics and never what random values may have resided in one or more bystander's documents, and we will not make these false-positive datasets public.

Attacks could involve making a large number of queries to the service. We rate-limit our queries appropriately and show how to extrapolate from our experiments to attackers with no qualms about submitting as many queries as possible per unit time.

3.7.3 GitHub

GitHub is one of the most popular source code hosting platforms, with 14 million users and 35 million repositories as of April 2016 according to Wikipedia. GitHub

has two types of repositories: public repositories and private repositories. Users can register for a free plan and set up unlimited numbers of public repositories, but no documents or repositories can be marked as private. To enable use of private repositories, one can choose a 7 USD per month plan. In a private repository, documents can be accessed and searched by their owner or authorized users. Non-authorized users should not be able to learn anything about the repository's contents, such as the number and type of documents, the contents of those documents, etc.

GitHub search API and basic experiments. GitHub uses ES (hosted by Elastic.co) as its search engine for full-text search [126]. A user could use a web-based interface or RESTful APIs to search for a term of interest. A search request will return with all the documents containing this term in both the public repositories as well as in private repositories to which the requesting user has access. The RESTful search API returns relevance scores to facilitate application development, which our attacks will exploit, while the web-interface returns ranked results without scores (we discuss attacking this setting in the Appendix). Based on public documentation [127, 126], we know that GitHub load balances across shards at the granularity of an individual repository: at the time the repository is created it is assigned to a shard. All documents in that repository are indexed within the assigned shard.

We first performed some manual experimentation using our score-dipping attack to both confirm the DF side channel and reverse engineer some undocumented aspects of the GitHub search service. We found that public repositories and private repositories use the same indexes. This means that, looking ahead, a malicious user could use (free) public repositories and the search API to extract sensitive terms from a victim's private repositories. We also observed that the index update time, i.e., the time between inserting a document into a repository and it being added to an index, is less than 1 minute in most cases.

Search queries emanating from a particular user account are limited to 5,000 per hour. There is a public interface for search as well, which does not require an account, and only searches public documents (which suffices for our attacks should an attacker use public repositories). This is rate limited to 60 per IP per hour. The GitHub search API allows queries with size less than or equal to 128

bytes. In our experiments, we primarily used private repositories for our simulated attacker, and found that pausing at least two seconds between two consecutive API requests avoids triggering rate limits. Therefore, in all our experiments, we pause for 2 seconds after search query and 60 seconds after creating/updating a document. Using longer pause times might be better for handling outliers (i.e., the index update time can be up to 2 minutes in rare cases), but would significantly increase the experiment running time.

Shard mapping. As mentioned GitHub hosts millions of repositories across many users, and therefore uses a large number of ES shards. We apply our shard mapping tool to determine how many, and to place an attacker document on each of the discovered shards.

We ran the shard mapping algorithm variant as described in §3.6, creating two new repositories each with one document over 513 rounds for a total of 1,026 repositories. The δ in CoShardTest was set to 0.05. We stopped after 50 consecutive rounds (100 repositories) failed to find a new shard. It took 104 hours and we discovered 191 shards. We might have missed some small number of shards. For example, assuming random assignment of repositories to shards, the probability of 100 consecutive failings if there were in fact 200 shards is 1%. Nevertheless, our shard map ends up sufficient for all experiments — all subsequent simulated victims ended up on one of the 191 shards that we discovered. We note an Elastic.co use-case description states that GitHub has 128 shards [127], suggesting this information is out of date.

After creating one repository on each shard at GitHub, we can generate many shard map sets $M_1, M_2, \ldots$ simply by creating one document on each repository.

We note that using shard mapping it would seem possible to track, over time, the number of shards used by GitHub. This could already be a hypothetical confidentiality issue for services that want to keep their infrastructure configuration secret.

Note that the consistency issues mentioned in §3.3 might produce false positives in CoShardTest; i.e., the difference in scores for two documents is greater than a threshold even though the documents are not on the same shard. To handle this issue, we double check after CoShardTest returns a positive result: we run

CoShardTest again and accept the result if both rounds of tests give positive results. We adopt this false positive identification method in the CoShardTest on all examined services.

DF prediction. As mentioned above, the documents in the same repository are assigned to the same shard. Doing more manual tests, we confirmed this, and leverage it in the design of our experiments. We use one account *AcctA* as the account for a simulated attacker and another account *AcctV* for a simulated victim.

We tested the accuracy of DF prediction as follows. For a given value of n_{DFE}, we create n_{DFE} training documents in a repository under the attacker's account *AcctA* and run DFPred. During the training, we use OriginLab [128] to test the data against all the functions provided, and find without exception the best-fit function is in the form of $f(x) = a - b * \ln(x + c)$, where x is the variable representing the unknown DF and a, b, c are coefficients. This function is consistent with the standard Elasticsearch scoring function in [129].

Using *AcctV* we generate n_{DF} victim documents in a single repository, each document containing the single term $\{t^*\}$ which is chosen as a random 16-byte alphabetic string. We vary n_{DF} and test the accuracy of the attack. We run the CoShardTest attack to place a document $d = \{t^*\}$ from *AcctA* on the same shard with the documents of *AcctV*. We then measure and record the score $\text{score}(t^*, d)$ by making a search query from *AcctA*. Then, we calculate $\text{df}_{\text{est}}(t^*) = f^{-1}(\text{score}(t^*, d)) - 1$ as an approximation of $\text{DF}(t^*)$ and measure the *relative error rate* (in percentage) and *absolute error* in order to evaluate estimation accuracy. The relative error rate is calculated as $|\text{df}(t^*) - \text{df}_{\text{est}}(t^*)|/\text{df}(t^*) * 100$ and the absolute error is calculated as $|\text{df}(t^*) - \text{df}_{\text{est}}(t^*)|$.

We perform experiments for each $n_{\text{DFE}}, n_{\text{DF}}$ pair for $n_{\text{DFE}} \in \{1, 5, 10, \ldots, 250\}$ and $n_{\text{DF}} \in \{0, 1, \ldots, 999\}$. Figure 3.3 shows the changes of the relevance score of $\text{score}(t^*, d)$ as $\text{DF}(t^*)$ increases from 1 to 1,000.

As shown in Table 3.4, the average relative errors (across all n_{DF}) for any n_{DFE} are all less than 0.5%, and average absolute errors are less than 3. We find that when $n_{\text{DFE}} \geqslant 50$, the average relative errors and the average absolute errors under different n_{DFE} are similar, i.e., the estimations do not become more accurate as we

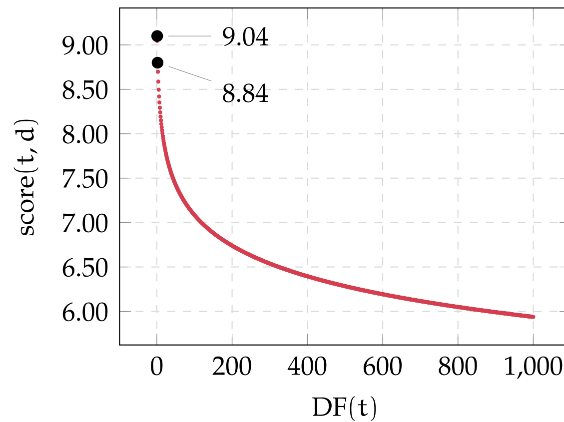


Figure 3.3: The changes of $\text{score}(t, d)$ as $\text{df}(t, D)$ increases. The scores when $\text{df}(t, D) = 1$ and $\text{df}(t, D) = 2$ are highlighted. Y-axis does not start from zero.

	Relative error			**Absolute error**		
	Min	**Avg**	**Max**	**Min**	**Avg**	**Max**
Alphabetic	0.07%	0.38%	0.53%	0.52	1.93	2.83
Numerical	0.13%	0.43%	0.58%	0.59	2.15	3.03

Figure 3.4: An overview of the average relative and absolute errors for DF prediction for all n_{DFE} on GitHub. The first row targets estimation for a random 16-byte alphabetic string and the second row is for random 16-byte number.

use more data points during regression analysis. Figure 3.5 shows a histogram of the errors for the 1,000 experiments (for the 1,000 different n_{DF} values) and for $n_{\text{DFE}} = 50$. As can be seen, the performance of the DF prediction is very good: about 10% of the estimations are correct; less than 14 of the estimations have absolute errors of 5. We note that the attack performs differently on alphabetic and numeric terms, likely due to boosting in the score function.

We repeat the experiments again on two further shards and get similarly small error rates: when $n_{\text{DFE}} = 50$, the average relative errors are 0.65% and 0.27%, and the average absolute errors are 3.9 and 1.2, respectively.

One important factor that can affect the estimation accuracy is the time we wait between updating a document and relevance score measurement. We find if

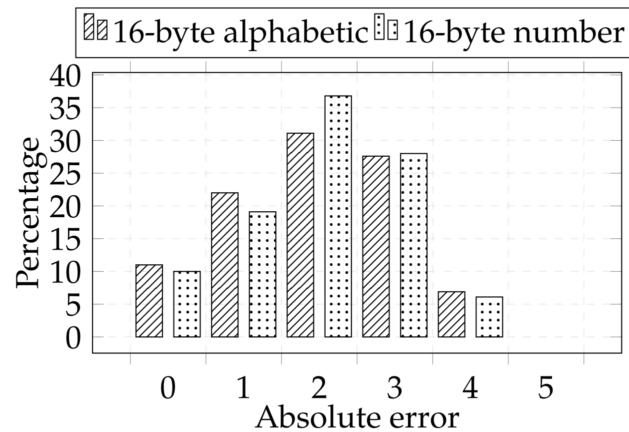


Figure 3.5: The distribution of the absolute errors when $n_{\text{DFE}} = 50$ (GitHub).

waiting only 30 seconds, there is so much noise in the data that we cannot even do a reasonable curve fit to the scoring function. However, sometimes 60 seconds might still not be long enough for an index to reach a stable state: we indeed observed unusual score variations during data collection. While the DF estimation already works well despite this noise, we believe the performance could be improved further with more effort on data collection and processing.

Term extraction attacks. We start by confirming that term extraction works correctly in a controlled setting. We generate a set of 50 victim terms B and a set of 50 control terms B'. We create a victim document $d = B$, and then run our term extraction attack on all the terms in $B \cup B'$ to see if it can properly identify the victim terms. We repeated the experiment 50 times.

To save time, once TermExtract finds the target shard containing d (i.e., a shard containing any of the terms form $B \cup B'$) we ignore the other shards and only do term exaction on the target shard. The average time for finding the target shard is 1,149 seconds, while the minimum time is 698 seconds and the maximum time is 1,607 seconds. The median number of tries (i.e., number of shards examined before finding the target index) is 98.

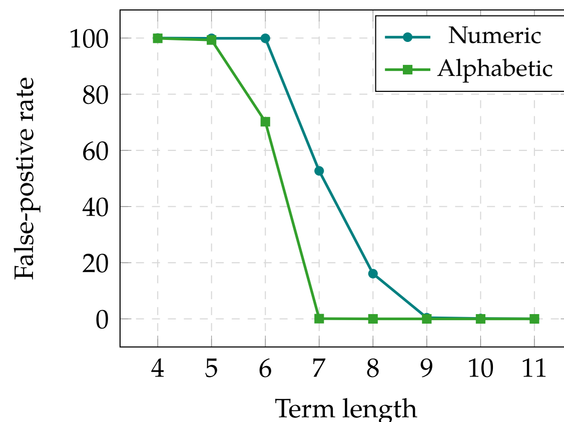


Figure 3.6: The average false-positive rates for different lengths of alphabetic-character-only term and numeric-character-only terms across three shards in GitHub and Orchestrate.io.

The attack achieves a true-positive rate of 100% and a false-positive rate of 0%. Since we also chose long random terms, excluding any noise due to bystanders, we conclude that the attack solved the experiment perfectly.

False positives on GitHub. The brute-force term extraction attack will encounter false positives due to bystander data. To understand how often terms happen to be contained on GitHub, we estimate the false-positive rate associated with low-entropy terms. We also test two types of terms: alphabetic-character-only terms and numeric-character-only terms. For a given length ℓ, we generate 20,000 terms of length ℓ ($\ell \geqslant 5$) and of a given type (10^4 terms for numeric-character-only term when $\ell = 4$) to construct B, and randomly select 5% of these terms as B'. We set ℓ to each of $4, 5, ..., 16$. We repeat the test on three different shards, and report on the average false-positive rates across 3 rounds in Figure 3.6. We can see when $\ell = 4, 5$, the false-positive rates are 100% or near 100% in both services. The false-positive rates are relatively high even when $\ell = 8$, but drops to a very small value ($< 0.5\%$) when $\ell \geqslant 9$ and zero when $\ell \geqslant 11$. We can also clearly see that numeric-character-only terms involve more false positives than alphabetic-character-only terms.

Feasibility of brute-force attacks. According to GitHub, developers sometimes leave CCN information in source code [41], and users might also store their own personal information on GitHub [130]. We argue that it is sometimes feasible for an attacker with partial information to harvest this (and other) information via the DF side channel.

Recall that in GitHub one account can send 5,000 requests per hour. Our brute-force attack will write large files containing terms to test and then issue one API call per term. Since writing the file requires a wait time for propagation to the index, one would pipeline the writes while performing the search queries. Assuming this is implemented, in the limit our term extraction needs one API request per term guess (using a modified version of TermExtract that uses one random term r_i to generate a score s_i that is then compared against the scores of many victim terms). Each guess checks if the term is on a particular shard. This gives a rough estimate of 120,000 guesses on a shard per day with one account. Creating n additional accounts increases the brute-forcing power by a factor n as the guessing algorithm can be run in parallel.

For a concrete example, if one knows the BIN (bank identification number) and last four digits of a CCN then there are about 10^6 possible CCNs. If an attack has focused on a particular shard the rest of the CCN could be brute-forced with one account in under a day. If the attacker is unsure of the shard, it could create one free account per shard and execute the attack in parallel (which, nicely, would be perfectly load-balanced on GitHub's backend).

3.7.4 Orchestrate.io

Orchestrate.io is a database-as-a-service platform for developing web and mobile applications. The information stored on Orchestrate.io is likely different than in GitHub since it is a generic key-value database and is being used to store all types of data. It seems likely that application backends store sensitive customer information in Orchestrate.io.

According to Orchestrate.io's official blog, it uses ES as its search engine [131], and has made efforts to secure its search API. However, we found its search API also expose the relevance scores of returned documents. Further tests suggested that the DF side channel also exists in Orchestrate.io.

The Orchestrate.io API does not restrict the number of operators in the query but enforces a maximum query size of 6 KB. The service does not have a specific rate-limiting policy but will throttle a user if her API requests affect their servers' performance. The index update time on Orchestrate.io is faster than GitHub. To avoid burdening on the target server, we decide to pause 30 seconds after creating/updating a document and 2 seconds after each search query.

Attack results. In Orchestrate.io, we use the same experiments as in GitHub to test MapShards and TermExtract. MapShards collects 50 shards in 12 hours, using 128 rounds with 256 documents being created. The δ in CoShardTest was set to 0.08. In TermExtract, the average time for locating the target shard is 324 seconds and the median number of tries is 15. The term extraction attack also achieves a true-positive rate of 100% and a false-positive rate of 0%.

We also conduct the same false-positive tests in Orchestrate.io. The average false-positive rates across three rounds are shown in Figure 3.6. As the term length increases, the false-positive rates drop to zero more quickly than on GitHub; when $7 \leqslant l \leqslant 9$, we only find very few false positives (1 to 3) for a given length.

To perform DFPred, we need to put multiple documents on the same shard. Unfortunately, unlike GitHub, no features in Orchestrate.io directly facilitate creating documents on the same shard. One solution is to create many documents and use CoShardTest to discover the documents that are on the same shard with a target document. However, this is very time-consuming. To speed this process up, we use the aforementioned ad-hoc score-based co-shard test in §3.5. More specifically, we create 30,000 documents that have the same content in $AcctV$, which is a unique 16-byte term, and measure the relevance scores of these documents. We group the documents by their relevance scores, and keep 500 documents from the largest group. To eliminate false positives, we use CoShardTest to confirm these

documents are indeed on the same index. We repeat these procedures in *AcctA* and keep 100 documents.

We perform experiments for each n_{DFE}, n_{DF} pair for $n_{DFE} \in \{1, 5, 10, \ldots, 100\}$ and $n_{DF} \in \{0, 1, \ldots, 499\}$. The scoring function in Orchestrate.io is still in the form of $a - b * \ln(x + c)$. The average relative and absolute error rates decrease as n_{DFE} increases. When $n_{DFE} = 100$, the average relative errors are about 2.2% and the average absolute errors are less than 6.0 for terms being tested. As $df(t^*)$ increases, the estimations become less accurate. The maximum absolute errors are 15. However, when $df(t^*) \leqslant 250$, the attack still performs well, with the maximum absolute error less than or equal to 2.

Feasibility of brute-force attacks. In Orchestrate.io, a free-plan user can only send 50,000 requests every month. So to search 10^9 terms, the attacker needs 20,000 accounts. Though this sounds costly, the process can be automated due to the fact that the account registration is very simple — the attacker just needs to fill in an email address and a password — and no captchas are being used.

Another choice is to use Orchestrate.io's professional plan, which is $499 per month, that allows one to send 5 M requests per month and pay $0.01 for 10 K additional requests. Sending 10^9 requests costs an attacker $1,500, but the gain of the attacks could be more than the cost. Of course, smaller spaces can be brute forced much more cheaply and quickly.

3.7.5 Xen.do

Xen.do is a hosted search service which aggregates data from a user's accounts on multiple third-party services, builds full-text indexes over the data, and provides interfaces to search the aggregated data. Xen.do supports more than 35 services, including, but not limited to, Google Apps (Gmail, Contacts, Drives, etc.), cloud storage services (Dropbox, OneDrive, etc.), customer relationship management (CRM) systems (Salesforce, ZohoCRM, etc.), and other services (Evernote, Office 365, etc.).

Sensitive information harvesting is particularly threatening on Xen.do since the data are collected from users' personal accounts. Xen.do makes an best effort to guarantee data security and privacy, and has received high ratings in various security tests such as Skyhigh Networks CloudTrust [132]. Unfortunately, we also find the DF side channel in Xen.do. We found the all the supported services in Xen.do share the same multi-tenant indexes. Therefore, a malicious user can extract the sensitive terms in other users' documents from different sources at the same time.

For Xen.do, its API access is not public and the API key can be only obtained on request. We only obtain a 30-day trial to the beta-test version of the API, which currently only provides basic operations such as full-text search and authentication. One operation — connecting Xen.do to a service — in the attacks must be done manually via the web interface.

The indexes updates on Xen.do are very slow, often taking about 20 to 30 minutes. In our attacks, after creating or updating a document, we query every 10 minutes to see if the document has been indexed. We still pause 2 seconds after each API request.

Attacks results. Using CoShardTest, we confirm that all the services supported by Xen.do are using the same set of shards. We create a document d_1 on a service $serv1$ (e.g., Gmail), and connect *AcctA* to $serv1$; then, we create a document d_2 on another service $serv2$ (e.g., Dropbox), and connect *AcctV* to $serv2$. We then use CoShardTest to test if d_1 and d_2 are on the same shard. If not, we disconnect *AcctA* from $serv1$ and reconnect it again, which forces Xen.do to assign d_1 to a new shard. We did two tests: (1) randomly chose 5 different pairs of $serv1$ and $serv2$, and (2) fix $serv1$ as Dropbox, and 17 different $serv2$. In both tests, CoShardTest usually succeeded in between 4 and 10 tries. The success of CoShardTest indicates that Xen.do uses the same set of shards for all services. The δ in CoShardTest was set to 0.08.

In MapShards, we stop the attack if we can't find more shards in 10 rounds. After 20 rounds, we found 4 shards.

Due to the restrictions of the Xen.do API and slow index propagation, we only collected a small amount of data. We use the ad-hoc score-based method again to put 50 documents on the same shard. Since the index updates are slow, it took us longer to run DFPRED (dominated by waiting). We had the best results fitting the scoring function to a curve of the form $f(x) = a - b * \ln(x + c)$. We use first the 15 data points to approximate the scoring function and the other data points for evaluation. The absolute errors of 40%, 49%, and 14% of the estimations are 0, 1, and 2, respectively. This preliminary assessment suggests that our attacks will work on Xen.do.

3.7.6 Rank-only Attacks on GitHub and Orchestrate.io.

We briefly checked if our rank-only attack works correctly against GitHub and Orchestrate.io, who provide web interfaces for performing multi-term search without returning relevance scores. On GitHub we started with a control experiment with an empty victim document and two attack documents on the same shard (recall that creating several co-resident documents is easy because GitHub shards based at the repository level). Using the web interface for GitHub search (which ranks but does not report scores), we observed that our attack returned a true negative (i.e. the order of the two attacker documents did not change). Next, we added the term t (in this case a long random string) to the victim document and re-ran the attack, which swapped the order of the attack documents in the web interface, confirming that the attack works.

Interestingly our attack failed on Orchestrate.io. This appears to be due to their using a non-standard scoring function for multi-term queries. We found that for multi-term queries, Orchestrate.io computed relevance scores that weight terms based on their order in the query. So, for instance, "t_1 t_2" will give different term weights from "t_2 t_1" while TF-IDF and common variants will treat these terms equivalently.

3.7.7 Conclusions

The results demonstrate that our score-based attacks can work on the three targets and can be used to extract sensitive data from other tenants' documents. Without relevance scores, one can still exploit the DF-side channel using rank-only attacks. All the services we tested claim protecting data security and data privacy as a priority. Indeed, they make efforts to secure their physical infrastructures, systems, and APIs. However, the DF side channels, hidden in their underlying search engines for years, make the services vulnerable to sensitive data leakage via side-channel attacks.

3.8 Countermeasures

Previously proposed countermeasures. The most obvious idea for a countermeasure is to simply not return relevance scores in response to searches, instead just providing an ordered list of documents. This might be a hindrance to applications that make use of the API's relevance scoring. But more importantly, while removing relevance scores would prevent our score-based attacks, as shown in §3.7, it does not prevent exploitation of the DF side channel via rank-only attacks.

One can remove the side channel by isolating each users' documents within independent indexes. Received wisdom suggests this approach is unsuitable for large-scale systems with many users due to poor performance [35].

Elastic.co suggests that services in some cases can disable scoring and ranking if the resulting functionality loss is acceptable. Another approach is to put sensitive terms in the fields that are not used for ranking, an approach suggested by Alex Brasetvik of Elastic.co. This will prevent the side channel being exploited for those terms, though some services might find reliably identifying sensitive information within tenants' data challenging.

All the approaches discussed above seem to have inherent limitations which might impede their usability in large-scale multi-tenant search indexes. Below we outline two approaches that eliminate DF side channels more efficiently. We

also implement and evaluate one approach. In both approaches, the searches are no longer scored strictly according to TF-IDF. Instead, the relevance score of a document d is computed as a function *only* of the public documents and of d. In particular, it is no longer a function of other private documents, whether or not d is public or private.

Public-corpus DFs. The first approach is called *public-corpus DFs*. The idea is to train a DF model using public data. In GitHub, for example, this would mean computing a DF model on a subset of public repositories. The model itself would be stored as an auxiliary index in Elasticsearch, enabling nodes to efficiently fetch the current public DF value for a term they have not seen. A default DF (of one) could be used for terms which do not appear in the public data. In settings where there is no notion of "public" and "private" information, this approach will not work with data on the service. Instead, one could train on suitable public file corpuses, should they exist.

Blind DFs. We call the second approach *blind DFs*. Recall that the search system we consider stores an inverted index that consists of per-term postings lists. At the head of each list, the DF is stored to speed up searching. Typically the DF value is equal to the length of the list.

To implement blind DFs, one augments each posting entry to contain a binary attribute indicating if the document is public (i.e. world readable) or not. We then modify the mechanisms for adding and deleting to maintain a count that we call the *blind DF*, which is now the number of *public* postings in the postings list. This can be achieved, say, by only incrementing or decrementing a posting lists' DF when adding or deleting a public document. Of course one may also store the (true) DF for purposes other than relevance scoring. This metadata must be stored for each document so document deletions can properly decrement the DF.

To process a (public or private) search with blind DFs, one modifies the system to use the blind DFs in place of true DFs, but otherwise leaves it unchanged. In particular, one could compute relevance scores exactly as before, but with a blind DF. To enforce access control one could use the post-processing filtering mechanism

as is currently deployed. Since the relevance scores are not a function of private documents, the DFs will contain only public information.

Evaluation. Both approaches increase the amount of storage space needed for the index. For blind DFs, the amount of extra space required is on the order of the number of documents in the index, since the public/private attributes for each document must be stored. For public-corpus DFs, the amount of extra space needed is only on the order of the number of unique terms in the index. The amount of space needed for public-corpus DFs does not change as more documents are added to the index, whereas the space overhead of blind DFs does increase over time. We believe public-corpus DFs will likely be better for large-scale search systems like Github's due to its low space complexity.

Here we report on initial experiments to assess the potential practicality of the countermeasure. All experiments were performed on an Ubuntu 16.04 desktop, using Lucene 6.3.0 and Java 8. The machine was equipped with a 512 GB NVMe SSD and 16 GB of DRAM. Microbenchmarks revealed a small latency increase of about 1% due to the countermeasure. We therefore focus our evaluation of public-corpus DFs on search quality.

We will use a standard methodology from information retrieval: queries with human-labeled relevance judgments. This measures search quality for a set of synthetic queries on a standard corpus by using human judges to label documents as relevant or non-relevant for each query, then evaluating a search engine's performance in retrieving relevant documents. We built our experiment by modifying Ian Soboroff's `trec-demo` project [133].

We use two datasets for the experiment. The first dataset was a pre-built Lucene index consisting of volumes 4 and 5 from NIST's Text Research Collection. These two volumes contain about 0.53 M total documents and 4.1 M unique terms. The index was not stemmed, but common stopwords were removed. We used the relevance judgments from the "ad hoc" track of the sixth, seventh, and eighth sessions of NIST's Text Retrieval Conference (TREC). There were 150 labeled queries in total. The second dataset is the Enron email dataset [134], which consists of 0.54 M documents and 0.6 M terms.

		Real DFs	Enron DFs
TF-IDF	MAP	0.17	0.17
	P@5	0.43	0.43
	P@20	0.31	0.31
	P@100	0.17	0.17
BM25	MAP	0.17	0.17
	P@5	0.44	0.43
	P@20	0.31	0.31
	P@100	0.17	0.17

Figure 3.7: Results of search quality experiment. MAP is "mean average precision". P@n is the precision only considering the top n documents returned for the search, averaged across all queries. Higher scores are better.

Our experiment consisted of a few concrete steps. We performed queries on two versions of the NIST index: an unmodified 'insecure" one which used the actual DFs of the NIST corpus and one which used public DFs from the Enron corpus. For both, we recorded the top 1,000 most relevant documents returned for each query. Finally, with the relevance judgments as ground truth, we computed quality metrics to measure the degradation in quality (if any) caused by our countermeasure.

We used two metrics from information retrieval: "precision" and "mean average precision". Intuitively, precision is the fraction of returned documents that were relevant to the query. If the number of relevant documents returned for a query is r and the total number of returned documents is s, the precision is defined as r/s. The metric P@n is defined as the precision when only considering the top n documents returned for the search. The numbers given in Figure 3.7 are averaged over all 150 queries.

Mean average precision is a related metric, defined simply as the mean over all queries of the per-query "average precision". The average precision is, importantly, *not* simply the average of an arbitrary set of precision scores. Average precision is defined in our case by measuring the precision at every cutoff point (i.e. n in P@n above) from 1 to 1,000, then summing and dividing by the number of relevant documents.

The results of the experiment are in Figure 3.7. The results using relevance scores computed with the default implementation is in the column of Figure 3.7 labelled "Real DFs". The results with the public-corpus DFs countermeasure enabled (using the Enron email corpus) are in the column labeled "Enron DFs". As we can see, using the Enron DFs in place of the real DFs for the corpus has negligible effect on the precision of the searches. Most values are identical when rounded to the hundredths place. This is true both for TF-IDF and the more modern BM25 scoring function.

3.9 Conclusion

We presented STRESS attacks. These demonstrate that the industry-standard method for multi-tenant search leads to an exploitable side channel, even in complex distributed systems. We developed efficient attacks on live services (GitHub, etc.). Using our side channel we estimated the time and cost required to extract information like phone and credit card numbers from private files stored in these services.

Our case studies only hint at the scope of affected systems. As mentioned, we also confirmed that following best practice guides for building multi-tenant search on top of AWS ElasticSearch, AWS CloudSearch, Searchly, bonsai, and Swifttype would lead to a DF side channel. Some of these search-as-a-service systems are in turn used by other cloud services, such as Heroku, which may therefore inherit any side channels. We have not yet performed in-depth experimentation with applications using these services, so it may be that noise or other subtleties prevent, e.g., brute-force term recovery attacks or accurate DF estimation. That said, services would do well to revisit their use of shared search indexes in order to prevent STRESS attacks.

Along another dimension, we have focused on attacks whose search queries include a single term. But many search services allow more sophisticated queries such as phrase or wildcard queries. We began thinking about how to exploit these,

but have not yet seen how they could provide attacks better than our single-term ones. Future work may do better.

Based on our experiments we recommend that the implementations move away from the simple filter-based approach to multi-tenancy. We suggested possible countermeasures, such as using document frequencies taken only from public documents, and our preliminary evaluation suggests this approach will be very practical for deployments.

4 Characterizing Serverless Computing Platforms

Serverless computing is an emerging paradigm in which an application's resource provisioning and scaling are managed by third-party services. Examples include AWS Lambda, Azure Functions, and Google Cloud Functions. Behind these services' easy-to-use APIs are opaque, complex infrastructure and management ecosystems.

Taking on the viewpoint of a serverless tenant, we conduct the largest measurement study to date, launching more than 50,000 function instances across these three services, in order to characterize their architectures, performance, and resource management efficiency. In this chapter, we explain how the platforms isolate the functions of different accounts, using either virtual machines or containers, which has important security implications. We characterize performance in terms of scalability, coldstart latency, and resource efficiency, with highlights including that AWS Lambda adopts a bin-packing-like strategy to maximize VM memory utilization, that severe contention between functions can arise in AWS and Azure, and that Google had bugs that allow customers to use resources for free.

4.1 Background

Serverless computing. In serverless computing, an application usually consists of one or more ***functions*** — standalone, small, stateless components dedicated to handle specific tasks. A function is most often specified by a small piece of code

written in some scripting language. Serverless computing providers manage the execution environments and backend servers of functions, and allocate resources dynamically to ensure their scalability and availability.

In recent years, many serverless computing platforms have been developed and deployed by cloud providers, including Amazon, Azure, Google, and IBM. We focus on Amazon AWS Lambda, Azure Functions and Google Cloud Functions.[1] In these services, a function is executed in a dedicated container or other type of sandbox with limited resources. We use ***function instance*** to refer to the container/sandbox a function runs on. The resources advertised as available to a function instance varies across platforms, as shown in Table 4.1. When the function is invoked by requests, one or more function instances (depending on the request volume) will be launched to execute the function. After the function instance(s) have processed the requests and exited or reached the maximum execution time (see "Timeout" in Table 4.1), the function instance(s) becomes idle. They may be reused to handle subsequent requests to avoid the delay of launching new instances. However, idle function instances can also be suddenly terminated [135]. Each function instance is associated with a non-persistent local disk for temporarily storing data, which will be erased when the function instance is destroyed.

One benefit of using serverless services is that tenants do not pay for resources consumed when function instances are idle. Tenants are billed based on resource consumption only during execution.[2] In common across platforms is charging for aggregated function execution time across all invocations. Additionally, the price varies depending on the pre-configured function memory (AWS, Google) or the actual consumed memory during invocations (Azure). Google further charges different rates based on CPU speed.

Related work. Many serverless application developers have conducted their own experiments to measure coldstart latency, function instance lifetime, maximum idle

[1]We use AWS, Azure and Google to refer to these services.

[2]Azure Functions offers two types of function hosting plans. *Consumption Plan* manages resources in a serverless-like way while *App Service Plan* is more like "container-as-a-service". We only consider Consumption Plan in this paper.

	AWS	Azure	Google
Memory (MB)	64 * k (k = 2, 3, ..., 24)	1536	128 * k (k = 1, 2, 4, 8, 16)
CPU	Proportional to Memory	Unknown	Proportional to Memory
Language	Python 2.7/3.6 Nodejs 4.3.2/6.10.3 Java 8, and others	Nodejs 6.11.5, Python 2.7, and others	Nodejs 6.5.0
Runtime OS	Amazon Linux	Windows 10	Debian 8*
Local disk (MB)	512	500	> 512
Run native code	Yes	Yes	Yes
Timeout (second)	300	600	540
Billing factor	Execution time Allocated memory	Execution time Consumed memory	Execution time Allocated memory Allocated CPU

Table 4.1: A comparison of function configuration and billing in three services. (*: We infer the OS version of GCF by checking the help information and version of several Linux tools such as `APT`.)

time before shut down, and CPU usage in AWS Lambda [47, 55, 56, 57, 52, 49, 50]. Unfortunately, their experiments were ad-hoc, and the results may be misleading because they did not control for contention by other instances. A few research papers report on measured performance in AWS. Hendrickson et al. [136] measured request latency and found it had higher latency than AWS Elastic Beanstalk (a platform-as-a-service system). McGrath et al. [61] conducted preliminary measurements on four serverless platforms, and found that AWS achieved better scalability, coldstart latency, and throughput than Azure and Google.

A concurrent study from Lloyd et al. [60] investigated the factors that affect application performance in AWS and Azure. The authors developed a heuristic to identify the VM a function runs on in AWS based on the VM uptime in `/proc/stat`. Our experimental evaluation suggests that their heuristic is unreliable (see §4.3.5), and that the conclusions they made using it are mostly inaccurate.

In our work, we design a reliable method for identifying instance hosts, and use systematic experiments to inspect resource scheduling and utilization.

4.2 Methodology

We take the viewpoint of a serverless user to characterize serverless platforms' architectures, performance, and resource management efficiency. We set up vantage points in the same cloud provider region to manage and invoke functions from one or more accounts via official APIs, and leverage the information available to functions to determine important characteristics. We repeated the same experiment under various settings by adjusting function configuration and workloads to determine the key factors that could affect measurement results. In the rest of the paper, we only report on the relevant factors affecting the experiment results.

We integrate all the necessary functionalities and subroutines into a single function that we call a *measurement function*. A measurement function performs two tasks: (1) collect invocation timing and function instance runtime information, and (2) run specified subroutines (e.g., measuring local disk I/O throughput, network throughput) based on received messages. The measurement function collects runtime information via the proc filesystem on Linux (`procfs`), environment variables, and system commands. It also reports on execution start and end time, invocation ID (a random 16-byte ASCII string generated by the function that uniquely identify an invocation), and function configurations to facilitate further analysis.

The measurement function checks the existence of a file named *InstanceID* on the local disk, and if it does not exist, creates this file with a random 16-byte ASCII string that serves as the function instance ID. Since the local disk is non-persistent and has the same lifetime as the associated function instance, the *InstanceID* file will not exist for a fresh function instance, and will not be modified or deleted during the function instance lifetime once created.

The regions for functions were us-east-1, us-central-1, "EAST US" in AWS, Google and Azure (respectively). The vantage points were VMs with at least 4 GB RAM and 2 vCPUs. We used the software recommended by the providers and

follow the official instructions to configure the time synchronization service in the vantage points. [3]

We implemented the measurement function in various languages, but most experiments used Python 2.7 and Nodejs 6.* as the language runtime (the top 2 most popular languages in AWS according to Newrelic[137]). We invoked the functions via synchronous HTTP requests. Most of our measurements were done from July–Dec 2017.

Ethical considerations. We built our measurement functions in a way that should not cause undue burden on platforms or other tenants. In most experiments, the function did no more than collecting necessary information and sleeping for a certain amount of time. Once we discovered performance issues we limited our tests to not DoS other tenants. We only conducted small-scale tests to inspect the security issues but did not further exploit them.

4.3 Serverless Architectures Demystified

We combine two approaches to infer the architectures of AWS Lambda, Google Cloud Functions, and Azure Functions: (1) reviewing official documents, related online articles and discussions, and (2) measurements — analyzing the data collected from running our measurement functions many times ($>$ 50,000) under varying conditions. This data enables partially reverse engineering the architectures of AWS, Azure, and Google.

4.3.1 Overview

AWS. A function executes in a dedicated function instance. Our measurements suggest different versions of a function will be treated as distinct and executed in different function instances (we discuss outliers in §4.4.3). The `procfs` file system

[3] AWS: `http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/set-time.html`; Google: `https://developers.google.com/time/`; Azure does not offer instructions so we use the default NTP servers at `http://www.pool.ntp.org/en/use.html`

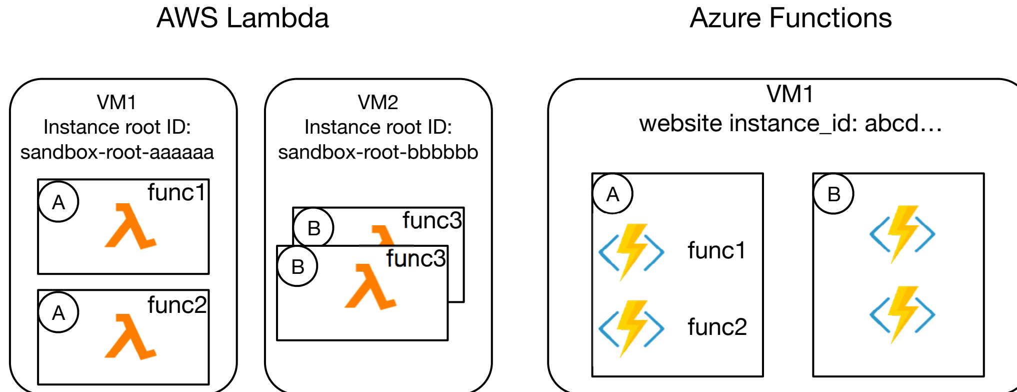


Figure 4.1: VM and function instance organization in AWS Lambda and Azure Functions. A rectangle represents a function instance. A or B indicates different tenants.

exposes global statistics of the underlying VM host, not just a function instance, and contains useful information for profiling runtime, identifying host VMs, and more. From `procfs`, we found host VMs mostly have 2 vCPUs and 3.75 GB physical RAM (same as EC2 c4.large instances).

Azure. Azure Functions uses *Function Apps* to organize functions. A function app, corresponding to one function instance, is a container that contains the execution environments for individual functions [138]. The environment variables in the function instance contain some global information about the host VM. The environment variables collected suggest the host VMs can have 1, 2 or 4 vCPUs.

One can create multiple functions in a function app and run them concurrently. In our experiments, we assume that a function app has only one function.

Google. Google isolates and filters information that can be accessed from `procfs`. The files under `procfs` only report usage statistics of the current function instance. Also, many system files and syscalls are obscured or disabled so we cannot get much information about runtime. The `/proc/meminfo` and `/proc/cpuinfo` files suggest a function instance has 2 GB RAM and 8 vCPUs, which we suspect is the configuration for VMs.

1. Set up N distinct functions $f_1, ..., f_N$ that run the following task upon receiving a `RUN` message: record `/proc/diskstats`, write 20 K – 30 K times to a file (1 byte each time), and record `/proc/diskstats` again.
2. Invoke each function once without `RUN` message to launch N function instances.
3. Assuming the instances of $f_1, ..., f_k$ (k instances) share the same instance root ID, invoke $f_1, ..., f_k$ once each with the `RUN` message and examine I/O statistics of each function instance.

Figure 4.2: I/O-based coresidency test in **AWS**.

4.3.2 VM identification

Associating function instances with VMs enables us to perform richer analysis. The heuristic for identifying VMs in AWS Lambda proposed by Lloyd et al., though theoretically possible, has never been evaluated experimentally [60]. Therefore, we looked for a more robust method.

AWS. The `/proc/self/cgroup` file has a special entry that we call *instance root ID*. It starts with “sandbox-root-” followed by a 6-byte random string. We found it can be used to reliably identity a host VM. Using the I/O-based coresidency tests (shown in Figure 4.2), we confirmed that the instances sharing the same instance root ID are on the same VM, as the difference in the total bytes written between two consecutive invocations, for f_i and f_{i+1} respectively, is almost the same as the number of bytes written by f_i. Moreover, we can get the same kernel uptime (or memory usage statistics) from the instances when reading `/proc/uptime` (`/proc/meminfo`) at the same time.

We call the IP obtained via querying IP address lookup tools from an instance *VM public IP*, and the IP obtained from running `uname` command *VM private IP*. Function instances that share the same instance root ID have the same VM public IP and VM private IP.

Azure. The `WEBSITE_INSTANCE_ID` environment variable serves as the VM identifier, according to official documents [139]. We refer to it as *Azure VM ID*. We used Flush-Reload via shared DLLs to verify coresidency of instances sharing the same Azure VM ID [140]. The results suggest Azure VM ID is a robust VM identifier.

Google. We could not find any information enabling us to identify a host. Using I/O-based coresidency did not work as `procfs` contains no global usage statistics. We tried to use performance as a covert-channel (e.g., performing patterned I/O operations in one function instance and detecting the pattern from I/O throughput variation in another) but found this is not reliable, as performance varied greatly (See §4.5.2).

4.3.3 Tenant isolation

Prior studies showed that co-located VMs in AWS EC2 allow attacks [4, 7, 141]. With the knowledge of instance-VM relationship, we examined how well tenants' primary resources — function instances — are isolated. We assume that one tenant corresponds to one user account, and only consider VM-level coresidency.

AWS. The functions created by the same tenant will share the same set of VMs, regardless of their configurations and code. The detailed instance placement algorithm will be discussed in §4.4.1. AWS assigns different VMs to each tenant, since we have never seen function instances from different tenants in the same VM. We conducted a cross-tenant coresidency test to confirm this assumption. The basic principle is similar to Figure 4.2: in each round, we create a new function under each of the two accounts at the same time, write a random number of bytes in one function, and check the disk usage statistics in another function. We ran this test for one week, but found no VM-coresidency of cross-tenant function instances.

Azure. Azure Functions are a part of the Azure App service, in which all tenants share the same set of VMs according to Azure [142]. Hence, tenants in Azure Functions should also share VM resources. A simple test confirmed this assumption: we invoked 500 functions in each of two accounts and found that 30% of function instances were coresident with a function instance from the other account, executing

in a total of 120 VMs. Note that as of May 2018, different tenants no longer share the same VMs in Azure. See §4.4.1 for more details.

4.3.4 Heterogeneous infrastructure

We found the VMs in all the considered services had a variety of configurations. The variety, likely resulting from infrastructure upgrades, can cause inconsistent function performance. To estimate the fraction of different types of VM in a given service, we examined the configurations of the host VMs of 50,000 unique function instances in each service.

In AWS, we checked the *model_name* entry and the processor numbers in the `/proc/cpuinfo`, and the *MemTotal* entry in the `/proc/meminfo`, and found five types of VMs: two E5-2666 vCPUs (2.90 GHZ), two E5-2680 vCPUs (2.80 GHZ), two E5-2676 vCPUs (2.40 GHZ), two E5-2686 vCPUs (2.30 GHZ), and **one** E5-2676 vCPUs. These types account for 59.3%, 37.5%, 3.1%, 0.09% and 0.01% of 20,447 distinct VMs.

Azure shows a greater diversity of VM configurations. The instances in Azure report various vCPU counts: of 4,104 unique VMs, 54.1% use 1 vCPU, 24.6% use 2 vCPUs, and 21.3% use 4 vCPUs. For a given vCPU count, there are three CPU models: two Intel and one AMD. Thus, nine (at least) different types of VMs are being used in Azure. Performance may vary substantially based on what kind of host (more specifically, the number of vCPUs) runs the function. See §4.5 for more details.

In Google, the *model_name* is always "unknown", but there are 4 unique model versions (79, 85, 63, 45), corresponding to 47.1%, 44.7%, 4.2%, and 4.0% of selected function instances.

4.3.5 Discussion

Being able to identify VMs in AWS is essential for our measurements. It helps to reduce noise in experiments and get more accurate results. For the sake of comparison, we evaluated the heuristic designed by Lloyd et al. [60]. The heuristic assumes that different VMs have distinct boot times, which can be obtained from

/proc/stat, and group function instances based on the boot time. We sent 10 – 50 concurrent requests at a time to 1536 MB functions for 100 rounds, used our methodology (instance root ID + IP) to label the VMs, and compared against the heuristic. The heuristic identified 940 VMs as 600 VMs, so 340 (36%) VMs were incorrectly labeled. So, we conclude this heuristic is not reliable.

None of these serverless providers completely hide runtime information from tenants. More knowledge of instance runtime and the backend infrastructure could make finding vulnerabilities in function instances easier for an adversary. In prior studies, procfs has been used as a side-channel [143, 144, 145]. In the serverless setting, one actually can use it to monitor the activity of coresident instances; while seemingly harmless, a dedicated adversary might use it as a steppingstone to more sophisticated attacks. Overall, accesses to runtime information, unless necessary, should be restricted for security purposes. Additionally, providers should expose such information in an auditable way, i.e., via API calls, so they are able to detect and block suspicious behaviors.

4.4 Resource Scheduling

We examine how instances and VMs are scheduled in the three serverless platforms in terms of instance coldstart latency, lifetime, scalability, and more.

4.4.1 Scalability and instance placement

Elastic, automatic scaling in response to changes in demand is a main advertised benefit of the serverless model. We measure how well platforms scale up.

We created 40 measurement functions of the same memory size $f_1, f_2, \ldots, f_{40}$ and invoked each f_i with $5i$ concurrent requests. We paused for 10 seconds between batches of invocations to cope with rate limits in the platforms. All measurement functions simply sleep for 15 seconds and then return. For each configuration we performed 50 rounds of measurements.

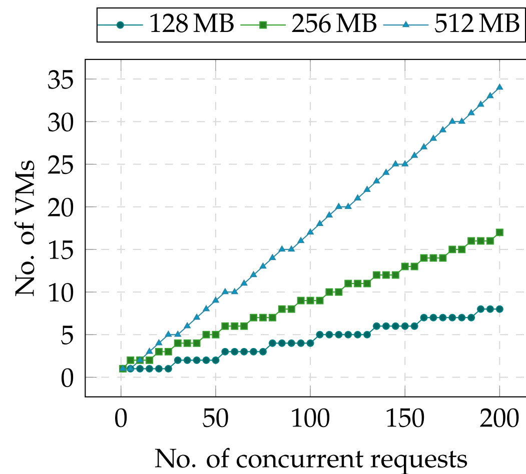


Figure 4.3: The total number of VMs being used after sending a given number of concurrent requests in **AWS**.

AWS. AWS is the best among the three services with regard to supporting concurrent execution. In our measurements, N concurrent invocations always produced N concurrently running function instances. AWS could easily scale up to 200 (the maximum measured concurrency level) fresh function instances.

We observed that **3,328 MB** was the maximum aggregate memory that can be allocated across all function instances on any VM in AWS Lambda. AWS Lambda appears to treat instance placement as a bin-packing problem, and tries to place a new function instance on an existing active VM to maximize *VM memory utilization rates*, i.e., the sum of instance memory sizes divided by 3,328. We invoked a single function with sets of concurrent requests, increasing from 5 to 200 with a step of 5, and recorded the total number of VMs being used after each number of requests. A few examples are shown in Figure 4.3. The number of active VMs are close to the "expected" number if AWS maximizes VM memory utilization. Quantitatively speaking, more than 89% of VMs we got in the test achieved 100% memory utilization. Sending concurrent requests to different functions resulted in the same pattern, indicating placement is agnostic to function code.

In a further test we sent 10 sets of random numbers of concurrent requests to randomly-chosen functions of varied memory sizes over 50 runs. AWS's placement still worked efficiently: the average VM memory utilization rate across VMs in the same run ranged from 84.6% to 100%, with a median of 96.2%.

Azure. Azure documentation states that it will automatically scale up to at most 200 instances for a single Nodejs-based function and at most one new function instance can be launched every 10 seconds [146]. However, in our tests of Nodejs-based functions, we saw at most 10 function instances running concurrently for a single function, no matter how we changed the interval between invocations. All the requests were handled by a small set of function instances. None of the concurrently running instances were on the same VM. So, it appears that Azure does not try to co-locate function instances of the same function on the same VMs.

We conducted a single-account coresidency test to examine how function instances are placed on VMs of different numbers of vCPUs. We invoked 100 different functions from one account at a time until we had 1,000 concurrent, distinct function instances running. We then checked for co-residency, and repeated the entire test 10 times.

We observed at most 8 instances on a single 1/2/4-vCPU VM. Co-resident instances tend to be on 1-vCPU VMs (presumably because there are more 1-vCPU VMs for Azure Functions). We show the breakdown of co-residency results in Table 4.2. In general, co-residency is undesirable for users wanting many function instances, as contention between instances on low-end VMs will exacerbate performance issues.

We further conducted a cross-account coresidency test in a more realistic scenario where an attacker wants to place her function instances on the same VM with the instances of a target victim. In each round of this test, we launched either 5 or 100 function instances from one account (the *victim*) and 500 simultaneous function instances from another account (the *attacker*). On average, 0.12% (3.82%) of the 500 attacker instances were coresident with the 5 (100) victim instances in each round (10 rounds in total). So, it was possible to achieve cross-tenant coresidency even for a few targets. In the test with 100 victim instances, we were able to obtain

#vCPU	Total	1	2	3	4	>4
1	61.3	16.6	24.6	13.7	4.9	1.5
2	19.5	7.3	7.1	3.3	1.4	0.4
4	19.2	7.6	6.2	3.9	1.3	0.2
All	100	31.5	37.9	20.9	7.6	2.1

Table 4.2: The average (over 10 runs) probabilities (as percentages) of getting N-way single-account coresidency (for $N \in \{1, 2, 3, 4, \}$ and $N > 4$, when launching 1,000 function instances in **Azure**. Here $N = 1$ indicates no coresidency among the functions.

up to 5 attacker instances on the same VM. Security implications will be discussed in §4.4.4.

We repeated the coresidency tests in May 2018 but could not find any cross-tenant coresident instances, even in the test in which we tired 500 victim instances. Therefore, we believe that Azure has fixed the cross-tenant coresidency issue.

Google. Google failed to provide our desired scalability, even though Google claims HTTP-triggered functions will scale to the desired invocation rate quickly [147]. In general, only about half of the expected number of instances, even for a low concurrency level (e.g., 10), could be launched at the same time, while the remainder of the requests were queued.

4.4.2 Coldstart and VM provisioning

We use *coldstart* to refer to the process of launching a new function instance. For the platform, a coldstart may involve launching a new container, setting up the runtime environment, and deploying a function, which will take more time to handle a request than reusing an existing function instance (*warmstart*). Thus, coldstarts can significantly affect application responsiveness and, in turn, user experience.

For each platform, we created 1,000 distinct functions of the same memory and language and sequentially invoked each of them twice to collect its coldstart and warmstart latency. We use the difference of invocation send time (recorded by the vantage point) and function execution start time (recorded by the function) as an

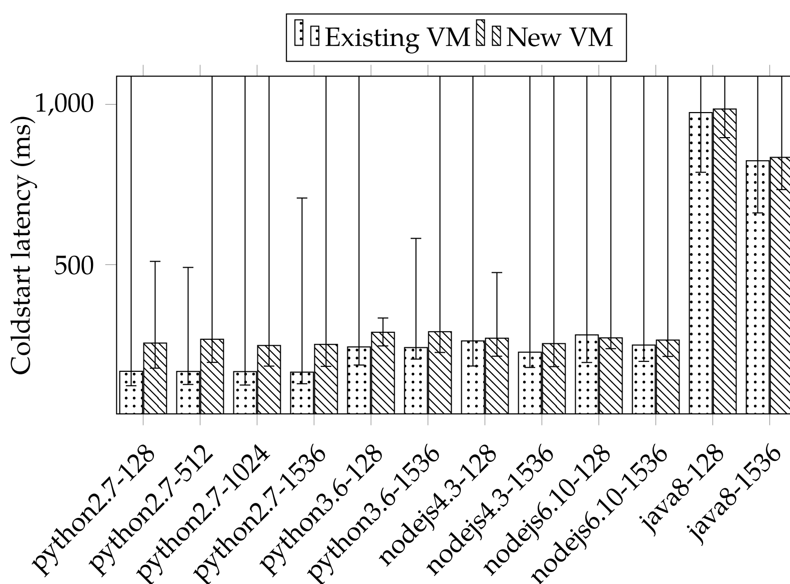


Figure 4.4: Median coldstart latency with min-max error bars (across 1,000 rounds) under different combinations of function languages and memory sizes in **AWS**. *Y-axis is truncated at 1,000 ms*.

estimation of its coldstart/warmstart latency. As baselines, the median warmstart latency in AWS, Google, and Azure were about 25, 79 and 320 ms (respectively) across all invocations.

AWS. We examine two types of coldstart events: a function instance is launched (1) on a new VM that we have never seen before and (2) on an existing VM. Intuitively, case (1) should have significantly longer coldstart latency than (2) because case (1) may involve starting a new VM. However, we found case (1) was only slightly longer than (2) in general. The median coldstart latency in case (1) was only 39 ms longer than (2) (across all settings). Plus, the smallest VM kernel uptime (from `/proc/uptime`) we found was 132 seconds, indicating that the VM has been launched before the invocation. So, AWS has a pool of ready VMs. The extra delays in case (1) are more likely introduced by scheduling (e.g., selecting a VM) rather than launching a VM.

Provider-Memory	Median	Min	Max	STD
AWS-128	265.21	189.87	7048.42	354.43
AWS-1536	250.07	187.97	5368.31	273.63
Google-128	493.04	268.5	2803.8	345.8
Google-2048	110.77	52.66	1407.76	124.3
Azure	3640.02	431.58	45772.06	5110.12

Table 4.3: Coldstart latencies (in ms) in AWS, Google, and Azure using Nodejs 6.* based functions for comparison.

Our results are consistent with prior observations: function memory and language affect coldstart latency [55], as shown in Figure 4.4. Python 2.7 achieves the lowest median coldstart latencies (167–171 ms) while Java functions have significantly higher latencies than other languages (824–974 ms). Coldstart latency generally decreases as function memory increases. One possible explanation is that AWS allocates CPU power proportionally to the memory size; with more CPU power, environment set up becomes faster (see §4.5.1).

A number of function instances may be launched on the same VM concurrently, due to AWS's instance placement strategy. In this case, the coldstart latency increases as more instances are launched simultaneously. For example, launching 20 function instances of a Python 2.7-based function with 128 MB memory on a given VM took 1,321 ms on average, which is about 7 times slower than launching 1 function instance on the same VM (186 ms).

Azure and Google. The median coldstart latency in Google ranged from 110 ms to 493 ms (see Table 4.3). Google also allocates CPU proportionally to memory, but in Google memory size has greater impact on coldstart latency than in AWS. It took much longer to launch a function instance in Azure, though their instances are always assigned 1.5 GB memory. The median coldstart latency was 3,640 ms in Azure. Anecdotes online [148] suggest that the long latency is caused by design and engineering issues in the platform that Azure is both aware of and working to improve.

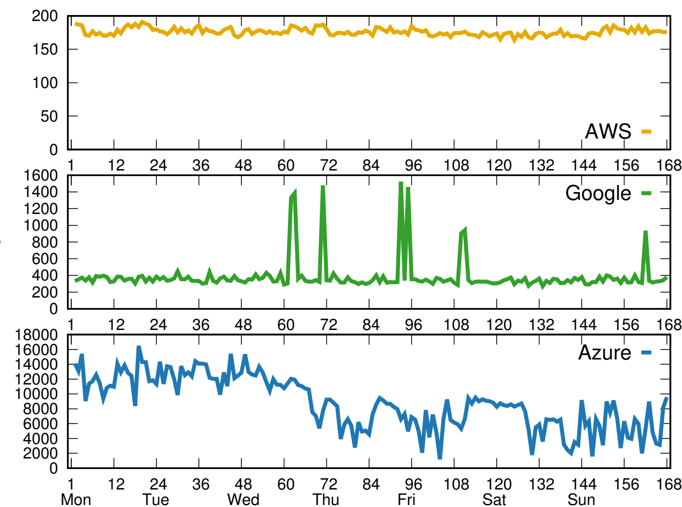


Figure 4.5: Coldstart latency (in ms) over 168 hours. All the measurements were started at right after midnight on a Sunday. Each data point is the median of all coldstart latencies collected in a given hour. For clarity, the y-axes use different ranges for each service.

Latency variance. We collected the coldstart latencies of 128 MB, Python 2.7 (AWS) or Nodejs 6.* (Google and Azure) based functions every 10 seconds for over 168 hours (7 days), and calculated the median of the coldstart latencies collected in a given hour. The changes of coldstart latency are shown in Figure 4.5. The coldstart latencies in AWS were relatively stable, as were those in Google (except for a few spikes). Azure had the highest network variation over time, ranging from about 1.5 seconds up to 16 seconds.

We repeated our coldstart measurements in May 2018. We did not find significant changes in coldstart latency in AWS. But, the coldstart latencies became 4x slower on average in Google, probably due to its infrastructure update in February 2018 [149], and 15x better in Azure. This result demonstrates the importance of developing a measurement platform for serverless systems (similar to [10] for IaaS) to do continuous measurements for better performance characterization.

4.4.3 Inconsistent function usage

Tenants expect the requests following a function update should be handled by the new function code, especially if the update is security-critical. However, we found in AWS there was a small chance that requests could be handled by an old version of the function. We call such cases *inconsistent function usage*. In the experiment, we sent $k = 1$ or $k = 50$ concurrent requests to a function, and did this again without delay after updating one of the following aspects of the function: IAM role, memory size, environment variables, or function code. For a given setting, we performed these operations for 100 rounds. When $k = 1$, 1%–4% of the tests used an inconsistent function. When there were more associated instances before the update ($k = 50$), 80% of our rounds had at least one inconsistent function. Looking across all tests from all rounds, we found that 3.8% of instances ran an inconsistent function. Examining the cases, we found two situations: (1) AWS launched new instances of the outdated function (2% of all the cases), and (2) AWS reused existing instances of the outdated function. Inconsistent instances never handle more than one request before terminating (note that max execution time is 300 s in AWS), but still, a considerable faction of requests may fail to get desired results.

As we waited for a longer time after the function update to send requests, we found fewer inconsistent cases, and eventually zero cases with a 6-second waiting time. So, we suspect that the inconsistency issues are caused by race conditions in the instance scheduler. The results suggest coordinating function update among multiple function instances is challenging as the scheduler cannot do an atomic update.

4.4.4 Discussion

We believe our results motivate further study on designing more efficient instance scheduling algorithms and robust schedulers to further improve VM resource utilization, i.e., to maximize VM memory usage, reduce scheduling latency, and promptly propagate function updates while guaranteeing consistency.

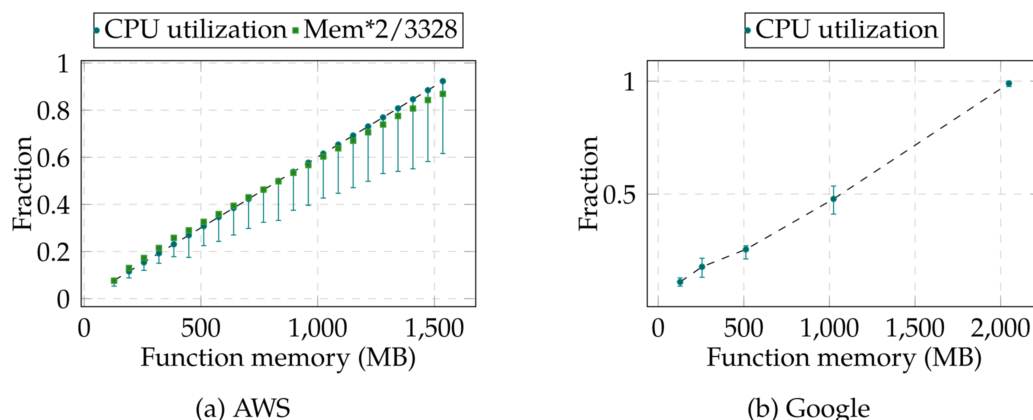
Loading modules or libraries could introduce high latency during coldstart [148, 150]. To reduce coldstart latency, providers might need to adopt more sophisticated library loading mechanisms, for example, using library caching to speed up this process, and resolving the library dependence before deployment and only loading required libraries.

Cross-tenant VM sharing in Azure plus the ability to run arbitrary binaries in the function instance could make applications vulnerable to many kinds of side-channel attacks [9, 151, 152, 153]. We did not examine how well Azure can tackle the potential threats resulting from cross-tenant VM sharing, and leave the actual security vulnerable as an open question.

AWS's bin-packing placement may bring security issues to an application, depending on its design. When a multi-tenant application in Lambda uses IAM roles to isolate its tenants, function instances from different application tenants still share the same VMs. We found two real services that use this pattern: Backand [154] and Zapier [155]. Both allow their tenants to deploy functions in Lambda in some way. We successfully achieved cross-account function coresidency in Backand in just a few tries, while failing in Zapier due to its rate limits and large user base (1 M+). Nevertheless, we could still observe the changes of `procfs` caused by other Zapier tenants' applications, which may admit side-channels [143, 144, 145]. For these multi-tenant applications to isolate their tenants and achieve better security and privacy, AWS may need to provide a finer-grained VM isolation mechanism, i.e., allocating a set of VMs to each IAM role instead of to each account.

4.5 Performance Isolation

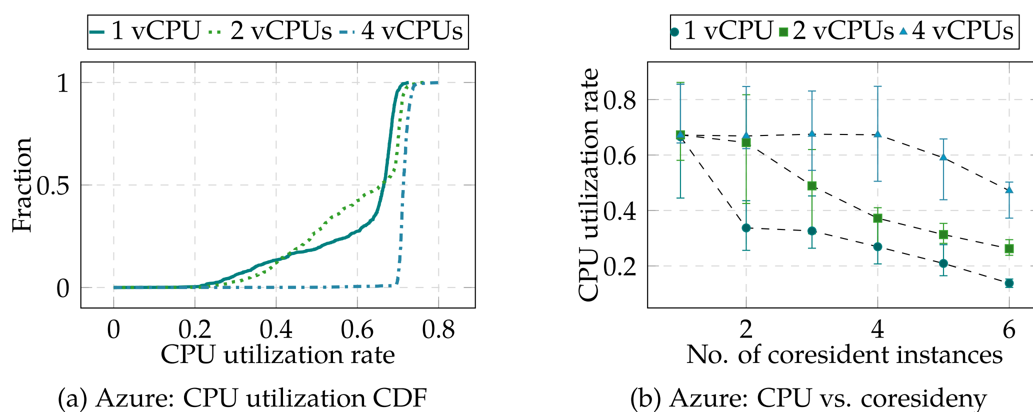
In this section, we investigate performance isolation. We mainly focus on AWS and Azure, where our ability to achieve coresidency allows more refined measurements. We also present some basic performance statistics for instances in Google that surface seeming contention with other tenants.



(a) AWS

(b) Google

Figure 4.6: The median instance CPU utilization rates with min-max error bars in AWS and Google as function memory increases, averaged across 1,000 instances for a given memory size.



(a) Azure: CPU utilization CDF

(b) Azure: CPU vs. coresideny

Figure 4.7: (a) CDFs of CPU utilization rates of instances (1,000 for each type) and (b) the median CPU utilization rates across a given number of coresident instances (50 rounds) in Azure, with min-max error bars.

4.5.1 CPU utilization

To measure CPU utilization, our measurement function continuously records timestamps using `time.time()` (Python) or `Date.now()` (Nodejs) for 1,000 ms. The metric

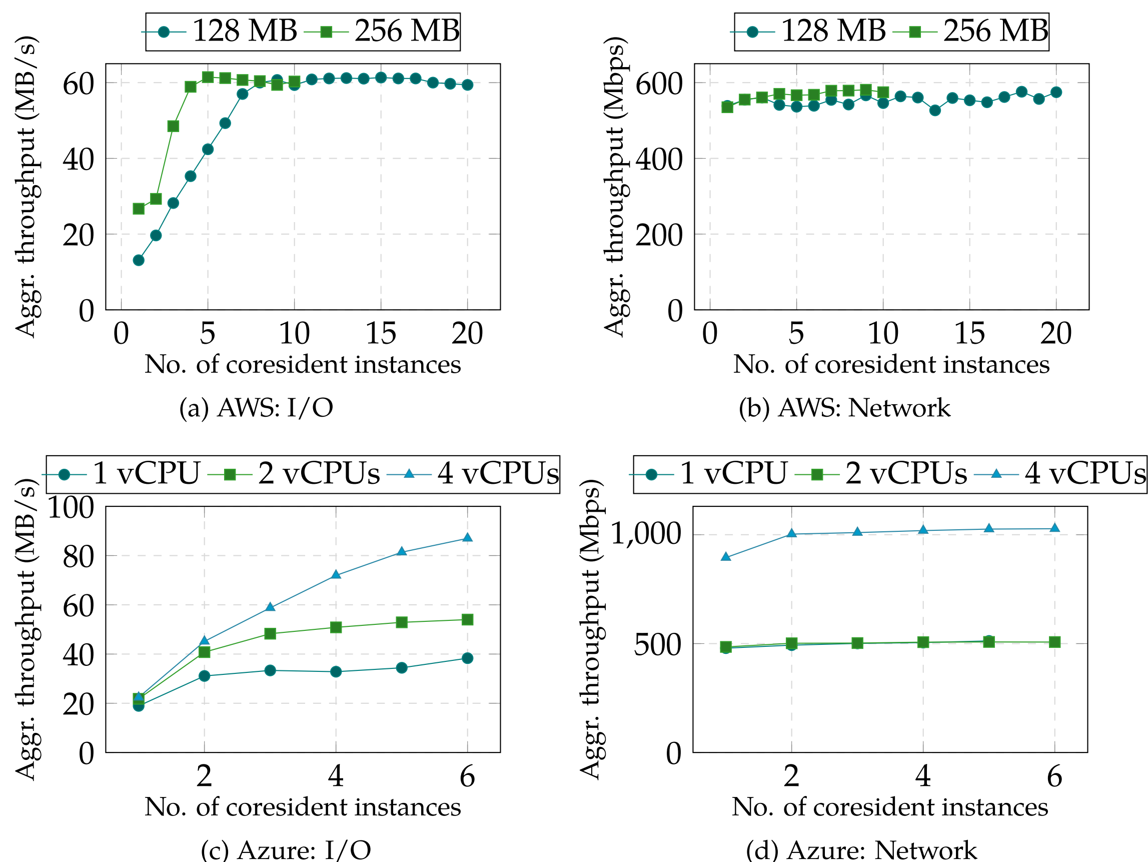


Figure 4.8: Aggregate I/O and network throughput across coresident instances as concurrency level increases. The coresident instances perform the same task simultaneously. The values are the median values across 50 rounds.

instance CPU utilization rate is defined as the fraction of the 1,000 ms for which a timestamp was recorded.

AWS. According to AWS, a function instance's CPU power is proportional its preconfigured memory [156]. However, AWS does not give details of how exactly CPU time is allocated to instances. We measured the CPU utilization rates on 1,000 distinct function instances and show the median rate for a given memory size in Figure 4.6a. Instances with higher memory get more CPU cycles. The median instance CPU utilization rate increased from 7.7% to 92.3% as memory

increased from 128 to 1,536 MB, and the corresponding standard deviations (SD) were 0.7% and 8.7%. When there is no contention from other coresident instances, the CPU utilization rate of an instance can vary significantly, resulting in inconsistent application performance. That said, an upper bound on CPU share is approximated by $2 * m/3328$, where m is the memory size.

We further examine how CPU time is allocated among coresident instances. We let colevel be the number of coresident instances and a colevel of 1 indicates only a single instance on the VM. For memory size m, we selected a colevel in the range 2 to $\lfloor 3328/m \rfloor$. We then measured the CPU utilization rate in each of the coresident instances. Examining the results over 20 rounds of tests, we found that the currently running instances share CPU fairly, since they had nearly the same CPU utilization rate (SD $<$0.5%). With more coresident instances, each instance's CPU share becomes slightly less than, but still close to $2 * m/3328$ (SD $<$2.5% in any setting).

The above results indicate that AWS tries to allocate a fixed amount of CPU cycles to an instance based only on function memory.

Azure and Google. Google adopts the same mechanism as AWS to allocate CPU cycles based on function memory [147]. In Google, the median instance CPU utilization rates ranged from 11.1% to 100% as function memory increased. For a given memory size, the standard deviations of the rates across different instances are very low (Figure 4.6b), ranging from 0.62% to 2.30%.

Azure has a relatively high variance in the CPU utilization rates (14.1%–90%), while the median was 66.9% and the SD was 16%. This is true even though the instances are allocated the same amount of memory. The breakdown by vCPU number shows that the instances on 4-vCPU VMs tend to gain higher CPU shares, ranging from 47% to 90% (Figure 4.7a). The distributions of utilization rates of instances on 1-vCPU VMs and 2-vCPU VMs are in fact similar; however, when colevel increased, the CPU utilization of instances on 1-vCPU VMs drops more dramatically, as shown in Figure 4.7b.

4.5.2 I/O and network

To measure I/O throughput, our measurement functions in AWS and Google used the `dd` command to write 512 KB of data to the local disk 1,000 times (with `fdatasync` and `dsync` flags to ensure the data is written to disk). In Azure, we performed the same operations using a Python script (which used `os.fsync` to ensure data is written to disk). For network throughput measurement, the function used `iperf 3.13` with default configurations to run the throughput test for 10 seconds with different same-region iperf servers, so that iperf server-side bandwidth was not a bottleneck. The iperf servers used the same types of VMs as the vantage points.

AWS. Figure 4.8 shows aggregate I/O and network throughput across a given number of coresident instances, averaged across 50 rounds. All the coresident instances performed the same measurement concurrently. Though the aggregate I/O and network throughput remains relatively stable, each instance gets a smaller share of the I/O and network resources as colevel increases. When colevel increased from 1 to 20, the average I/O throughput per 128 MB instance dropped by 4x, from 13.1 Mbps to 2.9 Mbps, and network throughput by 19x, from 538.6 MB/s to 28.7 MB/s.

Coresident instances get less share of the network with more contention. We calculate the Coefficient of Variation (CV), which is defined as SD divided by the mean, for each colevel. A higher CV suggests the performance of instances differ more. For 128 MB instances, the CV of network throughput ranged from 9% to 83% across all colevels, suggesting significant performance variability due to contention with coresident instances. In contrast, the I/O performance was similar between instances (CV of 1% to 6% across all colevels). However, the I/O performance is affected by function memory (CPU) for small memory sizes ($\leqslant$ 512 MB), and therefore the I/O throughput of an instance could degrade more when competing with instances of higher memory.

Azure. In Azure, the I/O and network throughput of an instance also drops as colevel increases, and fluctuates due to contention from other coresident instances. Even more interestingly, resource allocation is differentiated based on what type of

VM a function instance happens to be scheduled on. As shown in Figure 4.8, the 4-vCPU VMs could get 1.5x higher I/O and 2x higher network throughput than the other types of VMs. The 2-vCPU VMs have higher I/O throughput than 1-vCPU VMs, but similar network throughput.

Google. In Google, both the measured I/O and network throughput increase as function memory increases: the median I/O throughput ranged from 1.3 MB/s to 9.5 MB/s, and the median network throughput ranged from 24.5 Mbps to 172 Mbps. The network throughput measured from different instances with the same memory size can vary substantially. For instance, the network throughput measured in the 2,048 MB function instances fluctuated between 0.2 Mbps and 321.4 Mbps. We found two cases: (1) all instances' throughputs fluctuated during a given period of time, irrespective of memory sizes, or (2) a single instance temporarily suffered from degraded throughput. Case (1) may be due to changes in network conditions, while case (2) leads us to suspect that GCF tenants actually share hosts and suffer from resource contention.

4.5.3 Discussion

AWS and Azure fail to provide proper performance isolation between coresident instances, and so contention can cause considerable performance degradation. In AWS, the fact that they bin-pack function instances from the same account onto VMs means that scaling up a function places the same function on the same VM, resulting in resource contention and prolonged execution time (not to mention a longer coldstart latency). Azure has similar issues, with the additional issue that contention within VMs arises between accounts. The latter also opens up the possibility for cross-tenant degradation of service attacks.

We leave developing new, efficient isolation mechanisms that take the special characteristics of serverless (e.g., frequent instance creation, short-lived instances, and small memory-footprint functions) as considerations for future work.

4.6 Resource Accounting

In the course of our study, we found several resource accounting issues that can be abused by tenants.

Background processes. We found in Google one could execute an external script in the background that continued to run even *after* the function invocation concluded. The script we ran posted a 10 M file every 10 seconds to a server under our control, and the longest time it stayed alive was 21 hours. We could not find any logs of the network activity performed by the background process and were not charged for its resource consumption.[4,5] In contrast, one could run such background script in Azure but Azure logged all the activity. Our observations suggest that: (1) In Azure and Google the function instance execution context will not be frozen after an invocation, as opposed to AWS; and (2) Google does resource accounting via monitoring the Node.js process rather than the entire function instance.

One can exploit the billing issue in Google to run sophisticated tasks at negligible cost. For a function instance with 2 GB memory and 2.4 GHz CPU, one only needs to pay for a few invocations ($0.0000029/100 ms, with 2 M free calls) to get the same computing resources as using a g1-small instance ($0.0257/hour) on Google Cloud Platform.

CPU accounting. In Google, we found there was an 80% chance that a just-launched function instance (of any memory size other than 2,048 MB) could temporally gain more CPU time than expected. Measuring the CPU utilization rates and the completion times of a CPU-intensive task, we confirmed that the instances that one expects to have 8%–58% of the CPU time (see §4.5) had near 100% of the CPU time, the same as that given to 2,048 MB instances. The instance can retain the CPU resources until the next invocation. Note that if one wants to conduct performance measurements in Google, this issue could introduce a lot of noise (we appropriately controlled for it in previously reported experiments).

[4]Google has a free tier of service, but even after that is used up the background process consumption went unbilled.

[5]We have reported this issue to Google and Google has been working on fixing it as of May 2018.

4.7 Conclusion

In this chapter, we provided insights into architectures, resource utilization, and the performance isolation efficiency of three modern serverless computing platforms. We discovered a number of issues, raised from either specific design decisions or engineering, with regard to security, performance, and resource accounting in the platforms. Our results surface opportunities for research on improving resource utilization and isolation in future serverless platform designs.

We have repeated several measurements in May 2018 and highlight in the paper the improvements the providers have made. We noticed that serverless platforms are evolving quickly; nevertheless, our findings serve as a snapshot of the resource management mechanisms and efficiency of popular serverless platforms, provide performance baselines and design considerations for developers to build more reliable platforms, and help tenants improve their use of serverless platforms. More generally, our study provides new measurement techniques that are useful for other researchers.

5 Summary and Future Work

In this work, we examined security and privacy issues in different types of cloud applications via measurement-driven approaches. First, we examined the detectability of meek, a cloud-based network obfuscation tool. We found the way meek using clouds produces unexpected traffic patterns, which can be efficiently captured by machine learning based traffic analysis. We developed a trace analysis framework that uses real network traces as background traffic to simulate real DPI settings to evaluate our attacks, and found the false-positive rates of our attacks are sufficient low; the framework can facilitate future cloud security research that leverage traffic analysis, e.g., detecting the traffic from malicious applications in clouds. We also examined other types of obfuscation tools, and developed reliable attacks against them. Next, we discovered there exist logical side channels, which are inherited from some vulnerable search engines, in the full-text interfaces in many multi-tenant cloud applications. Such side channels allow a malicious tenant to breach the isolation boundary to learn sensitive information from other tenants' private data. We developed the first practical attacks called STRESS that exploit these side channels, and demonstrated our attacks work efficiently on three live services. Finally, we conducted a large-scale measurement to characterize three popular serverless computing platforms in terms of security and performance, and discovered various issues as a result of particular design choices, including, but not limited to, cross-tenant VM sharing, unpredictable performance, and bad

performance isolation. More specifically, tenants may be vulnerable to cross-tenant side channel attacks in Azure, and performance degradation attacks in both Azure and AWS.

Gaining comprehensive visibility into an application is important for improving its security; however, this becomes increasingly challenging because cloud applications and their dependencies usually are blackboxes. Exiting tools are not sufficient for cloud security research, as they are mainly designed for performance measurement, or have been deprecated due to the ever-changing cloud environment, or only work for specific applications. Our work fills this gap, offering a set of measurement tools to facilitate other studies to get insights into cloud applications. We exercised these tools to partially uncover the design, workflows, or architectures of several applications. This information will be useful for studying similar types of applications.

Today's cloud applications often rely on existing cloud services, which usually are blackboxes, to provide certain functionality for ease of development. This common pattern actually poses an enlarged attack surface to cloud applications: as demonstrated by our work, vulnerabilities not only exist in an application's code and logic, but also in the services the application depends on, and the interactions between the application and the services. These are often-ignored angles for discovering vulnerabilities in cloud applications.

Similarly, our work suggests that tenants may share resources in unexpected ways in a multi-tenant application. So instead of only focusing on the physical infrastructures of clouds, researchers can investigate other types of shared resources to look for side channels or other vulnerabilities. This leads to an interesting question: what can introduce side channels in multi-tenant applications? Answering this question is the first step towards understanding the anti-patterns in designing multi-tenant applications, developing better isolation mechanisms, and improving the security of multi-tenant applications.

There are many opportunities for continued research in this topic. We give three future research directions that naturally follow this dissertation as examples:

Inspecting more cloud applications. We can continue to explore the security and privacy issues originated from cloud-specific application design and deployment patterns. Novel cloud applications or services unceasingly come into use, and two types of them are particularly interesting: (1) "X-as-a-Service" services that try to simplify application development and management by delivering commonly used components to applications, such as Storage-as-a-Service and Payment-as-a-Service, and (2) applications that contribute to emerging technologies such as cloud-based self-driving cars, IoT platforms, and machine learning platforms. Such types of applications usually serve as building blocks of other applications, so understanding the security implications of their service models and design is valuable for improving security and privacy in a wide range of cloud applications.

Securing multitenancy cloud applications. We would like to answer the question about how to develop secure multitenancy applications. We realized security isolation is easily broken in multitenancy, especially in nested multitenancy (i.e., multitenancy applications that are built atop other multitenancy applications), and result in side channels. To understand what design patterns for multitenancy applications could be problematic, we will conduct a survey on the popular multitenancy applications of diverse categories to identify the common pitfalls in handling multitenancy. This will give us opportunities to discover new attack vectors that people have paid little attention to before. Then, based on the results in the survey, we plan to develop more general frameworks or mechanisms that can solve the discovered security issues accordingly, comply with the requirements for finer-grained access control in multitenancy, and assist in developing more secure multitenancy applications.

Protecting serverless applications. Serverless applications are mostly written in Python or Node.js. To facilitate development, tenants tend to reuse third-party libraries from pip or NPM, and may unintentionally use vulnerable or even malicious libraries. These libraries can be exploited by attackers to subvert the control flow and data flow of applications to steal sensitive data or perform stealthy operations. It is difficult to understand the expected behaviors of serverless applications, which

essentially are distributed applications, because of their interactions with different services for data externalization, and lack of debugging tools. Several serverless security solutions [157, 158, 159] have been developed, but, unfortunately, they only focus on protecting each individual function rather than the entire application, and ignore the distributed nature of serverless. As a result, they fail to provide desired security guarantee. So, we plan to develop a security framework for serverless applications that (1) facilitates a tenant to understand the control flow of her serverless application and enforce security policies to ensure control flow integrity, (2) provides better sensitive data management in serverless applications, (3) can be easily extended to cope with various security requirements, and (4) is transparent, meaning that it requires little modifications to existing serverless applications.

5.0.1 Lessons learned

Be critical. In the obfuscator detection project (Chapter 2), we started with examining the attacks proposed by some high impact papers. Many works follow those high impact work with regard to evaluation methodology, and so did we at the very beginning. Then, we tried a more realistic setting (i.e., detecting traffic containing a large amount of non-obfuscated flows; see §2.4), and expected to see these attacks still work; however, we only found that they, rather surprisingly, failed to achieve desired performance. We repeated our experiments several times to make sure we hadn't made any mistakes, and confirmed that our results were correct. This led us to think more critically about the previously proposed attacks, and developed a new evaluation methodology. So, prior work should not be treated as "gold standards", even though they are published in prestigious conferences. We should always evaluate prior work critically, during which we may come up with new research ideas.

Clouds are dynamic. Future cloud-related measurement studies should consider the dynamic nature of clouds. As we know, popular cloud providers, such as AWS, Google, and Azure, constantly update their services and infrastructures, which could make measurements invalid or outdated. For instance, just after we finished

the measurements on serverless platforms (Chapter 4), Azure changed its function instance scheduling strategy and eliminated cross-tenant instance coresidency. The worst-case scenario is such major infrastructure updates happen in the middle of measurements because one has to redo all the measurements again and may observe something completely different. One should stay up-to-date with measured targets, and be prepared to change her measurement methodology and iterate the experiments. That said, outdated results are still valuable, as they can provide insights into anti-patterns that should be avoided in the future system design. To dealing with the dynamic nature of clouds, one may consider performing long-term measurement, which can better characterizing cloud applications.

Don't forget research ethics. It is important to conduct research ethically. One must carefully design any experiments to follow the ethical practices (e.g., disclosing discovered vulnerabilities with the affected services before publishing your paper) in the related research areas. One important lesson learned from our research is always allowing targets to opt out of your research. In one measurement project [10], we crawled the top-level webpages hosted on the IPs in EC2 and Azure to find malicious webpages. We left our contact information and the project description in the User-Agent string of the crawler. In a three-month period, we received 84 emails from website operators, and excluded 67 IPs form our crawling as requested by some of the operators. In general, the reasons offered for requesting IPs be excluded were either: (1) the servers are not designed to be public, but with careless configurations, they are accidentally discovered by the crawler; and (2) since cloud providers charge tenants based on traffic, the traffic generated by crawler, though it is very small, is still unwelcome for some tenants. Overall, we cannot safely predict whether people like your research or not, and if it is impossible to collect consent from everyone, providing an opt-out option is always the right thing to do.

Bibliography

[1] Alexa. Alexa. http://alexa.com/, 2014.

[2] Keqiang He, Alexis Fisher, Liang Wang, Aaron Gember, Aditya Akella, and Thomas Ristenpart. Next stop, the cloud: understanding modern web service deployment in EC2 and Azure. In *IMC 2013*, pages 177–190. ACM, 2013.

[3] Cor-Paul Bezemer, Andy Zaidman, Bart Platzbeecker, Toine Hurkmans, and Aad't Hart. Enabling multi-tenancy: An industrial experience report. In *Software Maintenance (ICSM), 2010 IEEE International Conference on*, pages 1–8. IEEE, 2010.

[4] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan 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.

[5] Yunjing Xu, Michael Bailey, Farnam Jahanian, Kaustubh Joshi, Matti Hiltunen, and Richard Schlichting. An exploration of l2 cache covert channels in virtualized environments. In *Proceedings of the 3rd ACM workshop on Cloud computing security workshop*, pages 29–40. ACM, 2011.

[6] Zhenyu Wu, Zhang Xu, and Haining Wang. Whispers in the hyper-space: High-speed covert channel attacks in the cloud. In *USENIX Security symposium*, pages 159–173, 2012.

[7] Venkatanathan Varadarajan, Yinqian Zhang, Thomas Ristenpart, and Michael M Swift. A placement vulnerability study in multi-tenant public clouds. In *USENIX Security*, pages 913–928, 2015.

[8] Mehmet Sinan Inci, Berk Gülmezoglu, Gorka Irazoqui Apecechea, Thomas Eisenbarth, and Berk Sunar. Seriously, get off my cloud! cross-vm rsa key recovery in a public cloud. *IACR Cryptology ePrint Archive*, 2015:898, 2015.

[9] Yinqian Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. Cross-tenant side-channel attacks in PaaS clouds. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 990–1003. ACM, 2014.

[10] Liang Wang, Antonio Nappa, Juan Caballero, Thomas Ristenpart, and Aditya Akella. Whowas: A platform for measuring web deployments on iaas clouds. In *Proceedings of the 2014 Conference on Internet Measurement Conference*, pages 101–114. ACM, 2014.

[11] Tor project. Obfsproxy2. `https://gitweb.torproject.org/pluggable-transports/obfsproxy.git/tree/doc/obfs2/obfs2-protocol-spec.txt`, 2015.

[12] Tor project. Obfsproxy3. `https://gitweb.torproject.org/pluggable-transports/obfsproxy.git/tree/doc/obfs3/obfs3-protocol-spec.txt`, 2015.

[13] Philipp Winter, Tobias Pulls, and Juergen Fuss. Scramblesuit: A polymorphic network protocol to circumvent censorship. In *Proceedings of the 12th ACM workshop on Workshop on privacy in the electronic society*, pages 213–224. ACM, 2013.

[14] Brandon Wiley. Dust: A blocking-resistant internet transport protocol. *Technical rep ort. http://blanu. net/Dust. pdf*, 2011.

[15] Kevin P Dyer, Scott E Coull, Thomas Ristenpart, and Thomas Shrimpton. Protocol misidentification made easy with format-transforming encryption. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 61–72. ACM, 2013.

[16] Zachary Weinberg, Jeffrey Wang, Vinod Yegneswaran, Linda Briesemeister, Steven Cheung, Frank Wang, and Dan Boneh. Stegotorus: a camouflage proxy for the tor anonymity system. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 109–120. ACM, 2012.

[17] Hooman Mohajeri Moghaddam, Baiyu Li, Mohammad Derakhshani, and Ian Goldberg. Skypemorph: Protocol obfuscation for tor bridges. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 97–108. ACM, 2012.

[18] Qiyan Wang, Xun Gong, Giang TK Nguyen, Amir Houmansadr, and Nikita Borisov. Censorspoofer: asymmetric communication using ip spoofing for censorship-resistant web browsing. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 121–132. ACM, 2012.

[19] J. Appelbaum and N. Mathewson. Pluggable transports for circumvention. `https://www.torproject.org/docs/pluggable-transports.html.en`, 2012.

[20] University of Washington. uProxy. https://www.uproxy.org/, 2015.

[21] Tor project. Tor metrics. https://metrics.torproject.org/, 2015.

[22] Chad Brubaker, Amir Houmansadr, and Vitaly Shmatikov. Cloudtransport: Using cloud storage for censorship-resistant networking. In *International Symposium on Privacy Enhancing Technologies Symposium*, pages 1–20. Springer, 2014.

[23] goagent. Goagent. https://github.com/goagent, 2015.

[24] Tor project. Tor meek. https://trac.torproject.org/projects/tor/wiki/doc/meek, 2015.

[25] David Fifield, Chang Lan, Rod Hynes, Percy Wegmann, and Vern Paxson. Blocking-resistant communication through domain fronting. *Proceedings on Privacy Enhancing Technologies*, 2015(2):46–64, 2015.

[26] Amir Houmansadr, Chad Brubaker, and Vitaly Shmatikov. The parrot is dead: Observing unobservable network communications. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 65–79. IEEE, 2013.

[27] Tor project. Obfsproxy3. https://gitweb.torproject.org/pluggable-transports/obfsproxy.git/tree/doc/obfs3/obfs3-protocol-spec.txt, 2015.

[28] Yawning. Obfsproxy4. https://github.com/Yawning/obfs4/blob/master/doc/obfs4-spec.txt, 2015.

[29] Elasticsearch. https://www.elastic.co/products/elasticsearch, 2016.

[30] Apache Solr. http://lucene.apache.org/solr/, 2016.

[31] MySQL full text search. http://dev.mysql.com/doc/refman/5.7/en/fulltext-search.html, 2011.

[32] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, New York, NY, USA, 2008.

[33] Wikipedia. Term frequency-inverse document frequency. https://en.wikipedia.org/wiki/Tf-idf, 2016.

[34] Wikipedia. Okapi BM25. https://en.wikipedia.org/wiki/Okapi_BM25.

[35] Elasticsearch. Discovering the need for an indexing strategy in multi-tenant applications. `https://www.elastic.co/blog/found-multi-tenancy`, 2015.

[36] Elasticsearch: the definitive guide. `https://www.elastic.co/guide/en/elasticsearch/guide/current/shared-index.html`, 2016.

[37] Stefan Büttcher and Charles L. A. Clarke. A security model for full-text file system search in multi-user environments. In *Proceedings of the 4th Conference on USENIX Conference on File and Storage Technologies - Volume 4*, FAST'05, 2005.

[38] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of AES. In *Cryptographers" Track at the RSA Conference*, pages 1–20. Springer, 2006.

[39] Daniel J Bernstein. Cache-timing attacks on AES, 2005.

[40] Colin Percival. Cache missing for fun and profit, 2005.

[41] GitHub. Sensitive data exposure. `https://bounty.github.com/classifications/sensitive-data-exposure.html`, 2016.

[42] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. Occupy the cloud: distributed computing for the 99%. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 445–451. ACM, 2017.

[43] Mengting Yan, Paul Castro, Perry Cheng, and Vatche Ishakian. Building a chatbot with serverless computing. In *Proceedings of the 1st International Workshop on Mashups of Things and APIs*, page 5. ACM, 2016.

[44] Alex Glikson, Stefan Nastic, and Schahram Dustdar. Deviceless edge computing: extending serverless computing to the edge of the network. In *Proceedings of the 10th ACM International Systems and Storage Conference*, page 28. ACM, 2017.

[45] Erik Peterson. Serverless security and things that go bump in the night. `https://www.infoq.com/presentations/serverless-security`, 2017.

[46] Andrew Krug. Hacking serverless runtimes profiling Lambda, Azure, and more., 2017.

[47] Frederik Willaert. AWS Lambda container lifetime and config refresh. `https://www.linkedin.com/pulse/aws-lambda-container-lifetime-config-refresh-frederik-willaert`, 2016.

[48] Security and serverless. https://read.acloud.guru/security-and-serverless-ec52817385c4, 2017.

[49] How does proportional CPU allocation work with AWS Lambda? https://engineering.opsgenie.com/how-does-proportional-cpu-allocation-work-with-aws-lambda-41cd44da3cac, 2018.

[50] The occasional chaos of AWS Lambda runtime performance. https://blog.symphonia.io/the-occasional-chaos-of-aws-lambda-runtime-performance-880773620a7e, 2017.

[51] Lambda CPU relative to which instance type? https://forums.aws.amazon.com/message.jspa?messageID=614558, 2014.

[52] How long does AWS Lambda keep your idle functions around before a cold start? https://read.acloud.guru/how-long-does-aws-lambda-keep-your-idle-functions-around-before-a-cold-start-bf715d3b810, 2017.

[53] Consumption plan scaling issues. https://github.com/Azure/azure-webjobs-sdk-script/issues/1206, 2017.

[54] AWS Lambda performance issues. https://stackoverflow.com/questions/43089879/aws-lambda-performance-issues, 2017.

[55] How does language, memory and package size affect cold starts of AWS Lambda? https://read.acloud.guru/does-coding-language-memory-or-package-size-affect-cold-starts-of-aws-lambda-a15e26d12c76, 2017.

[56] Understanding AWS Lambda performance. https://blog.newrelic.com/2017/01/11/aws-lambda-cold-start-optimization/, 2017.

[57] Understanding AWS Lambda coldstarts. https://www.iopipe.com/2016/09/understanding-aws-lambda-coldstarts/, 2016.

[58] My accidental 3âŁ“5x speed increase of AWS Lambda functions. https://serverless.zone/my-accidental-3-5x-speed-increase-of-aws-lambda-functions-6d95351197f3, 2017.

[59] Comparing AWS Lambda performance when using Node.js, Java, C# or Python. https://read.acloud.guru/comparing-aws-lambda-performance-when-using-node-js-java-c-or-python-281bef2c740f, 2017.

[60] Wes Lloyd, Shruti Ramesh, Swetha Chinthalapati, Lan Ly, and Shrideep Pallickara. Serverless computing: An investigation of factors influencing microservice performance. In *Cloud Engineering (IC2E), 2018 IEEE International Conference on*, pages 159–169. IEEE, 2018.

[61] Garrett McGrath and Paul R Brenner. Serverless computing: Design, implementation, and performance. In *Distributed Computing Systems Workshops (ICDCSW), 2017 IEEE 37th International Conference on*, pages 405–410. IEEE, 2017.

[62] Xueyang Xu, Z Morley Mao, and J Alex Halderman. Internet censorship in China: Where does the filtering occur? In *Passive and Active Measurement*, pages 133–142. Springer, 2011.

[63] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The second-generation onion router. Technical report, DTIC Document, 2004.

[64] Philipp Winter and Stefan Lindskog. How the great firewall of China is blocking tor. *Free and Open Communications on the Internet*, 2012.

[65] Yawning. Obfsproxy4. `https://github.com/Yawning/obfs4/blob/master/doc/obfs4-spec.txt`, 2015.

[66] Daniel Luchaup, Kevin P. Dyer, Somesh Jha, Thomas Ristenpart, and Thomas Shrimpton. LibFTE: A toolkit for constructing practical, format-abiding encryption schemes. In *Proceedings of USENIX Security 2014*, August 2014.

[67] Kevin P. Dyer, Scott E. Coull, and Thomas Shrimpton. Marionette: A programmable network-traffic obfuscation system. In *Proceedings of USENIX Security 2015*, August 2015.

[68] Vern Paxson. Bro: a system for detecting network intruders in real-time. *Computer networks*, 31(23):2435–2463, 1999.

[69] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive: a warehousing solution over a map-reduce framework. *Proceedings of the VLDB Endowment*, 2(2):1626–1629, 2009.

[70] Apache. Hive operators and user-defined functions. `https://cwiki.apache.org/confluence/display/Hive/LanguageManual+UDF`, 2015.

[71] Marc Juarez, Sadia Afroz, Gunes Acar, Claudia Diaz, and Rachel Greenstadt. A critical evaluation of website fingerprinting attacks. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 263–274. ACM, 2014.

[72] Tor project. Stem. `https://stem.torproject.org/`, 2015.

[73] Seleniumhq.org. Selenium - web browser automation. http://www.seleniumhq.org/, 2015.

[74] ISO. 171/sc 2: Iso 32000–1: 2008 document management-portable document format-part 1: Pdf 1.7.

[75] Karl Heinz Kremer. The trouble with the XREF table. http://khkonsulting.com/2013/01/the-trouble-with-the-xref-table/, 2013.

[76] Balachander Krishnamurthy, Jeffrey C Mogul, and David M Kristol. Key differences between HTTP/1.0 and HTTP/1.1. *Computer Networks*, 31(11):1737–1751, 1999.

[77] Microsoft. Invalid content-length header may cause requests to fail through ISA server. https://support.microsoft.com/en-us/kb/300707, 2007.

[78] Hill01. "Weird" utf-8 characters in POST body causing content-length mismatch. https://github.com/strongloop/express/issues/1816, 2013.

[79] Nigel Williams, Sebastian Zander, and Grenville Armitage. A preliminary performance comparison of five machine learning algorithms for practical IP traffic flow classification. *ACM SIGCOMM Computer Communication Review*, 36(5):5–16, 2006.

[80] Mashael AlSabah, Kevin Bauer, and Ian Goldberg. Enhancing tor's performance using real-time traffic classification. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 73–84. ACM, 2012.

[81] John Barker, Peter Hannay, and Patryk Szewczyk. Using traffic analysis to identify the second generation onion router. In *Embedded and Ubiquitous Computing (EUC), 2011 IFIP 9th International Conference on*, pages 72–78. IEEE, 2011.

[82] Andriy Panchenko, Lukas Niessen, Andreas Zinnen, and Thomas Engel. Website fingerprinting in onion routing based anonymization networks. In *Proceedings of the 10th annual ACM workshop on Privacy in the electronic society*, pages 103–114. ACM, 2011.

[83] Xiang Cai, Xin Cheng Zhang, Brijesh Joshi, and Rob Johnson. Touching from a distance: Website fingerprinting attacks and defenses. In *Proceedings of the 2012 ACM conference on Computer and Communications Security*, pages 605–616. ACM, 2012.

[84] Thomas Karagiannis, Konstantina Papagiannaki, and Michalis Faloutsos. Blinc: multilevel traffic classification in the dark. In *ACM SIGCOMM Computer Communication Review*, volume 35, pages 229–240. ACM, 2005.

[85] Laurent Bernaille, Renata Teixeira, Ismael Akodkenou, Augustin Soule, and Kave Salamatian. Traffic classification on the fly. *ACM SIGCOMM Computer Communication Review*, 36(2):23–26, 2006.

[86] Thuy TT Nguyen and Grenville Armitage. Training on multiple sub-flows to optimise the use of machine learning classifiers in real-world ip networks. In *Local Computer Networks, Proceedings 2006 31st IEEE Conference on*, pages 369–376. IEEE, 2006.

[87] Jesse Davis and Mark Goadrich. The relationship between precision-recall and roc curves. In *Proceedings of the 23rd international conference on Machine learning*, pages 233–240. ACM, 2006.

[88] David Cournapeau. Scikit-learn: Machine learning in Python. `http://scikit-learn.org/`, 2007.

[89] Charles Reis, Ratul Mahajan, Maya Rodrig, David Wetherall, and John Zahorjan. Measurement-based models of delivery and interference in static wireless networks. In *ACM SIGCOMM Computer Communication Review*, volume 36, pages 51–62. ACM, 2006.

[90] Shravan Rayanchu, Anadi Mishra, Deepak Agrawal, Simanto Saha, and Suman Banerjee. Diagnosing wireless packet losses in 802.11: Separating collision from weak signal. In *INFOCOM 2008. The 27th Conference on Computer Communications. IEEE*. IEEE, 2008.

[91] Bruce Leidl. Obfuscated-openssh. `https://github.com/brl/obfuscated-openssh`, 2009.

[92] Andrew Rukhin, Juan Soto, James Nechvatal, Miles Smid, and Elaine Barker. A statistical test suite for random and pseudorandom number generators for cryptographic applications. Technical report, DTIC Document, 2001.

[93] George Nychis, Vyas Sekar, David G Andersen, Hyong Kim, and Hui Zhang. An empirical evaluation of entropy-based traffic anomaly detection. In *Proceedings of the 8th ACM SIGCOMM conference on Internet measurement*, pages 151–156. ACM, 2008.

[94] Jing Yuan, Zhu Li, and Ruixi Yuan. Information entropy based clustering method for unsupervised internet traffic classification. In *Communications, 2008. ICC'08. IEEE International Conference on*, pages 1588–1592. IEEE, 2008.

[95] Chaim Sanders, Jacob Valletta, Bo Yuan, and Daryl Johnson. Employing entropy in the detection and monitoring of network covert channels. 2012.

[96] Andrew M White, Srinivas Krishnan, Michael Bailey, Fabian Monrose, and Phillip A Porras. Clear and present data: Opaque traffic and its security implications for the future. In *NDSS*, 2013.

[97] Eric Jones, Travis Oliphant, Pearu Peterson, et al. SciPy: Open source scientific tools for Python. http://www.scipy.org/, 2001.

[98] Ntop.org. nDPI. http://www.ntop.org/products/deep-packet-inspection/ndpi/, 2015.

[99] Alexis Communeau, Paul Quillent, and Alexandre Compain. Detecting FTE proxy. 2014.

[100] Simurgh Aryan, Homa Aryan, and J Alex Halderman. Internet censorship in iran: A first look. *Free and Open Communications on the Internet, Washington, DC, USA*, 2013.

[101] Phobos. Kazakhstan upgrades censorship to deep packet inspection. https://blog.torproject.org/blog/kazakhstan-upgrades-censorship-deep-packet-inspection, 2012.

[102] Eugene. Age of surveillance: the fish is rotting from its head. http://non-linear-response.blogspot.com/2011/01/age-of-surveillance-fish-is-rotting.html, 2011.

[103] Philipp Winter and Jedidiah R Crandall. The great firewall of China: How it blocks tor and why it is hard to pinpoint. 2012.

[104] Nethanel Gelernter and Amir Herzberg. Cross-site search attacks. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1394–1405. ACM, 2015.

[105] Ariel Futoransky, Damián Saura, and Ariel Waissbein. The ND2DB attack: Database content extraction using timing attacks on the indexing algorithms. In *WOOT*, 2007.

[106] Lucene. https://lucene.apache.org/, 2016.

[107] Lucene's practical scoring function. https://www.elastic.co/guide/en/elasticsearch/guide/current/practical-scoring-function.html.

[108] Lucene's scoring function. http://lucene.apache.org/core/3_5_0/api/core/org/apache/lucene/search/Similarity.html.

[109] Elasticsearch. Term Filter query. https://www.elastic.co/guide/en/elasticsearch/reference/current/query-dsl-filtered-query.html, 2016.

[110] Index Aliases. https://www.elastic.co/guide/en/elasticsearch/reference/current/indices-aliases.html#filtered, 2016.

[111] Microsoft. Multi-tenant data architecture. https://msdn.microsoft.com/en-us/library/aa479086.aspx, 2006.

[112] Amazon. Amazon Elasticsearch service. https://aws.amazon.com/elasticsearch-service.

[113] Amazon. Amazon Cloudsearch. https://aws.amazon.com/cloudsearch.

[114] Searchly – Elasticsearch as a service. https://http://www.searchly.com, 2016.

[115] Bonsai – Hosted Elasticsearch. https://bonsai.io, 2016.

[116] Swiftype - site search and enterprise search. https://swiftype.com, 2016.

[117] Swiftype. Customer case studies. https://swiftype.com/customers, 2016.

[118] Add-ons - Heroku Elements. https://elements.heroku.com/addons#search, 2016.

[119] PostgreSQL. https://www.postgresql.org.

[120] Couchbase – NoSQL database. http://www.couchbase.com.

[121] Cratedb. https://crate.io.

[122] Searchify. https://www.searchify.com.

[123] Google. Google app engine. https://cloud.google.com/appengine.

[124] elastic.co. Updating a whole document. https://www.elastic.co/guide/en/elasticsearch/guide/current/update-doc.html, 2016.

[125] Joseph K. Blitzstein and Jessica Hwang. *Introduction to Probability*. Chapman and Hall/CRC, 2014.

[126] Andrew Cholakian. Elasticsearch at GitHub. http://exploringelasticsearch.com/github_interview.html, 2014.

[127] GitHub on Elastic.co case study. https://www.elastic.co/use-cases/github, 2014.

[128] OriginLab. http://originlab.com/, 2016.

[129] Lucene Practical Scoring function. https://www.elastic.co/guide/en/elasticsearch/guide/current/practical-scoring-function.html, 2016.

[130] Vulnerability.ch. Creative commons: Donors data leak. https://vulnerability.ch/tag/github/, 2014.

[131] Orchestrate. How we improved elasticsearch indexing. https://www.ctl.io/developers/blog/post/improved-elasticsearch-indexing, 2014.

[132] Xendo. Xendo security blog. https://help.xen.do/hc/en-us/sections/200689704-Security, 2016.

[133] Ian Soboroff. Information retrieval evaluation demo. https://github.com/isoboroff/trec-demo.

[134] Bryan Klimt and Yiming Yang. The enron corpus: A new dataset for email classification research. In *European Conference on Machine Learning*, pages 217–226. Springer, 2004.

[135] Understanding container reuse in AWS lambda. https://aws.amazon.com/blogs/compute/container-reuse-in-lambda/, 2014.

[136] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Serverless computation with open-lambda. In *Proceedings of the 8th USENIX Conference on Hot Topics in Cloud Computing*, pages 33–39. USENIX Association, 2016.

[137] AWS Lambda in production. https://blog.newrelic.com/2017/11/21/aws-lambda-state-of-serverless/, 2017.

[138] Create your first function in the Azure portal. https://docs.microsoft.com/en-us/azure/azure-functions/functions-create-first-azure-function, 2017.

[139] Azure runtime environment. https://github.com/projectkudu/kudu/wiki/Azure-runtime-environment, 2017.

[140] Yuval Yarom and Katrina Falkner. Flush+ reload: a high resolution, low noise, L3 cache side-channel attack. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 719–732, 2014.

[141] Zhang Xu, Haining Wang, and Zhenyu Wu. A measurement study on co-residence threat inside the cloud. In *USENIX Security Symposium*, pages 929–944, 2015.

[142] Azure app service, virtual machines, service fabric, and cloud services comparison. https://docs.microsoft.com/en-us/azure/app-service/choose-web-site-cloud-service-vm, 2017.

[143] Suman Jana and Vitaly Shmatikov. Memento: Learning secrets from process footprints. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 143–157. IEEE, 2012.

[144] Xiaoyong Zhou, Soteris Demetriou, Dongjing He, Muhammad Naveed, Xiaorui Pan, XiaoFeng Wang, Carl A Gunter, and Klara Nahrstedt. Identity, location, disease and more: Inferring your secrets from android public resources. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 1017–1028. ACM, 2013.

[145] Qi Alfred Chen, Zhiyun Qian, and Zhuoqing Morley Mao. Peeking into your app without actually seeing it: UI state inference and novel android attacks. In *USENIX Security Symposium*, pages 1037–1052, 2014.

[146] Azure Functions scale and hosting. `https://docs.microsoft.com/en-us/azure/azure-functions/functions-scale`, 2017.

[147] Google Cloud Functions quotas. `https://cloud.google.com/functions/quotas`, 2017.

[148] Cold start taking a long time in consumption mode for C# Azure Function. `https://github.com/Azure/azure-functions-host/issues/838`, 2017.

[149] Google cloud functions release notes. `https://cloud.google.com/functions/docs/release-notes`, 2018.

[150] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Sock: Rapid task provisioning with serverless-optimized containers. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATCâŁ™18)*, 2018.

[151] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. S$A: a shared cache attack that works across cores and defies vm sandboxing and its application to AES. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 591–604. IEEE, 2015.

[152] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache template attacks: Automating attacks on inclusive last-level caches. In *USENIX Security Symposium*, pages 897–912, 2015.

[153] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+Flush: a fast and stealthy cache attack. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 279–299. Springer, 2016.

[154] Backand. `https://www.backand.com/`, 2018.

[155] Backand. `https://zapier.com/`, 2018.

[156] Configuring Lambda functions. `https://docs.aws.amazon.com/lambda/latest/dg/resource-model.html`, 2017.

[157] Functionshielde. https://www.puresec.io/function-shield.

[158] Epsagon. https://epsagon.com/.

[159] Intrinsic. https://intrinsic.com/.