How Bad Can It Git? Characterizing Secret Leakage in Public GitHub Repositories

Michael Meli
North Carolina State University
mjmeli@ncsu.edu

Matthew R. McNiece
North Carolina State University
Cisco Systems, Inc.
mrmcniec@ncsu.edu

Bradley Reaves
North Carolina State University
bgreaves@ncsu.edu

***Abstract*—GitHub and similar platforms have made public collaborative development of software commonplace. However, a problem arises when this public code must manage authentication secrets, such as API keys or cryptographic secrets. These secrets must be kept private for security, yet common development practices like adding these secrets to code make accidental leakage frequent. In this paper, we present the first large-scale and longitudinal analysis of secret leakage on GitHub. We examine billions of files collected using two complementary approaches: a nearly six-month scan of real-time public GitHub commits and a public snapshot covering 13% of open-source repositories. We focus on private key files and 11 high-impact platforms with distinctive API key formats. This focus allows us to develop conservative detection techniques that we manually and automatically evaluate to ensure accurate results. We find that not only is secret leakage pervasive — affecting over 100,000 repositories — but that *thousands* of new, unique secrets are leaked every day. We also use our data to explore possible root causes of leakage and to evaluate potential mitigation strategies. This work shows that secret leakage on public repository platforms is rampant and far from a solved problem, placing developers and services at persistent risk of compromise and abuse.**

I. Introduction

Since its creation in 2007, GitHub has established a massive community composed of nearly 30 million users and 24 million public repositories [1], [11], [55]. Beyond merely storing code, GitHub is designed to encourage public, collaborative development of software. The rise in popularity of public, "social" coding also comes at a time where software, perhaps more than ever, relies on external online services for essential functionality. Examples include APIs for maps, credit card payments, and cloud storage, to say nothing of integration with social media platforms. As part of this integration, developers generally have to authenticate to the service, typically by using static random API keys [35], which they must manage securely. Developers may also need to manage cryptographic public and private keys for access control (e.g., SSH) or TLS.

Unfortunately, the public nature of GitHub often comes in conflict with the need to keep authentication credentials private. As a result, these secrets are often — accidentally or intentionally — made public as part of the repository. Secrets leaked in this way have been exploited before [4], [8], [21], [25], [41], [46]. While this problem is known, it remains unknown to what extent secrets are leaked and how attackers can efficiently and effectively extract these secrets.

In this paper, we present the first comprehensive, longitudinal analysis of secret leakage on GitHub. We build and evaluate two different approaches for mining secrets: one is able to discover 99% of newly committed files containing secrets in real time, while the other leverages a large snapshot covering 13% of all public repositories, some dating to GitHub's creation. We examine millions of repositories and billions of files to recover hundreds of thousands of secrets targeting 11 different platforms, 5 of which are in the Alexa Top 50. From the collected data, we extract results that demonstrate the worrying prevalence of secret leakage on GitHub and evaluate the ability of developers to mitigate this problem.

Our work makes the following contributions:

- **We perform the first large-scale systematic study across billions of files that measures the prevalence of secret leakage on GitHub by extracting and validating hundreds of thousands of potential secrets.** We also evaluate the time-to-discovery, the rate and timing of removal, and the prevalence of co-located secrets. Among other findings, we find thousands of new keys are leaked daily and that the majority of leaked secrets remain available for weeks or longer.
- **We demonstrate and evaluate two approaches to detecting secrets on GitHub.** We extensively validate the discovery coverage and rejection rates of invalid secrets, including through an extensive manual review.
- **We further explore GitHub data and metadata to examine potential root causes.** We find that committing cryptographic key files and API keys embedded directly in code are the main causes of leakage. We also evaluate the role of development activity, developer experience, and the practice of storing personal configuration files in repositories (e.g., "dotfiles").
- **We discuss the effectiveness of potentially mitigating practices**, including automatic leakage detectors, requiring multiple secret values, and rate limiting queries on GitHub. Our data indicates these techniques all fail to limit systemic large-scale secret exposure.

Network and Distributed Systems Security (NDSS) Symposium 2019
24-27 February 2019, San Diego, CA, USA
ISBN 1-891562-55-X
https://dx.doi.org/10.14722/ndss.2019.23418
www.ndss-symposium.org

We are not the first to recognize and attempt to measure secret leakage on GitHub [4], [8], [25], [33], [46], [48], [53], [63], [65]. However, much of this work has used techniques that provide superficial conclusions (e.g., merely reporting search query hits), fail to evaluate the quality of their detection heuristics, operate on a scale orders of magnitude smaller than this work, or fail to perform in-depth analyses of the problem. To the best of our knowledge, no peer-reviewed research has addressed this issue apart from a single short paper [53] published in 2015. We believe our findings present significant added value to the community over prior work because we go far beyond noting that leakage occurs, providing a conservative longitudinal analysis of leakage, as well as analyses of root causes and the limitations of current mitigations. Throughout this work, we take conservative approaches to increase confidence in the validity of our results. Consequently, our work is not exhaustive but rather demonstrates a *lower bound* on the problem of secret leakage on GitHub. The full extent of the problem is likely much worse than we report.

The remainder of this paper is organized as follows: Section II analyzes related work; Section III describes our secret detection process and experimental methodology; Section IV contains an ethical statement from the authors; Section V characterizes secret leakage; Section VI evaluates our methodology; Section VII performs data analysis to investigate root cause of leakage; Section VIII considers various case studies; Section IX discusses potential mitigations; Section X acknowledges threats to validity; and Section XI concludes the paper.

II. Related Work

GitHub is the world's most popular site for storing code [2] and thus is a popular place for software engineering research. Researchers have analyzed GitHub data to see how software engineers track issues [7], [10], [37], resolve bugs [54], use pull requests [61], [66], and even investigate gender bias in open-source projects [57]. Due to GitHub's research popularity, researchers have created tools such as GHTorrent [31] and Boa [13] to assist others, and Google maintains a snapshot of open-source repositories in BigQuery [28], [36].

Despite its importance, security-sensitive secret information is regularly leaked. Data breaches regularly compromise users' PII and secrets [26]. A clever Google search can reveal files containing passwords and keys to an attacker [29], [44]. Popular resources such as Docker images and AWS VMs can be full of security issues to the publisher, consumer, and manager of an environment [64]. These images, which are often public, have frequently been found to contain leftover secrets that can be easily obtained by attackers [5] or have numerous vulnerabilities that threaten contained secrets [52].

Tools exist that work to identify secrets in text for secrets of both fixed and variable formats. Since variable format passwords and API keys can have high degrees of entropy [12], [60], one approach for finding secrets is searching for high entropy strings; this technique is employed by tools like TruffleHog [59]. For finding secrets with fixed structures, regular expressions can be effective. Targeted regular expressions have been built to extract API keys from Android applications on Google Play [12], [62] and have recently been added to TruffleHog [59]. Unfortunately, these tools are prone to large numbers of false positives as they use an inaccurate set of regular expressions that often match generic Base64 strings, and they generally have a smaller set of targets than our work. Supervised neural networks have been built to attempt to solve these issues [12], but ultimately fall victim to the same problems due to their limited training data. Essentially, there is no existing tool that can be used to confidently mine GitHub at a large-scale.

Secret leakage via GitHub first gained significant attention in 2013 when people used GitHub's search tool with targeted strings to find thousands of keys and passwords [21], [25], [41]. This problem remains, and leakage cannot easily be fixed with a commit to remove the secret as the secret can be recovered from Git history [50]. Tools exist to automatically recover secrets from history [32], although they cannot be used at a large-scale. Websites have discussed this problem [4], [8], [33], [46], [48], [63], [65], but they use naive techniques that result in shallow or non-validated conclusions.

The work closest to ours is by Sinha et al. [53], which, to our knowledge, is the only peer-reviewed work on GitHub secret leakage. This short paper identified AWS keys in a sample of repositories using regular expressions and light static analysis. This work only investigated Java files in 84 repositories for a single credential type with unvalidated heuristics. In our paper, we develop more accurate techniques to mine for 19 types of secrets at a large-scale. We also examine related issues like root causes and potential mitigations.

III. Secret Detection

In this section, we describe our approach for detecting and validating secrets. We define a "secret" as a cryptographic key or API credential whose privacy must be maintained for security. We briefly outline the overall strategy here before discussing details in the following subsections.

A major issue in detecting secrets is avoiding false positives from non-secret random strings. Naively using tools from prior work, such as scanning for high entropy strings or writing regular expressions matching known secret formats, may result in high numbers of false positives as strings detected by these methods are not guaranteed to be secret. In order to avoid this problem, we developed a rigorous multi-phase process that combined multiple methods to detect candidate secrets and then validate them to obtain high confidence in their sensitivity.

Our multi-phase process is shown in Figure 1. We began in Phase 0 by surveying a large set of API credentials and cryptographic keys to identify any with distinct structures unlikely to occur by chance, giving high confidence in their validity if detected. We then wrote regular expressions to recognize these secrets. *Note that we did not attempt to examine passwords as they can be virtually any string in any given file type, meaning they do not conform to distinct structures and making them very hard to detect with high accuracy.*

Then, in Phases 1a and 1b, we pursued two complementary approaches for locating files that may contain secrets. In Phase 1a, we developed targeted queries for GitHub's Search API to collect "candidate files", which were files likely to contain secrets. We continuously searched this API to identify new secrets as they are committed in real-time. In Phase 1b, we searched for secrets in a snapshot of GitHub maintained as a

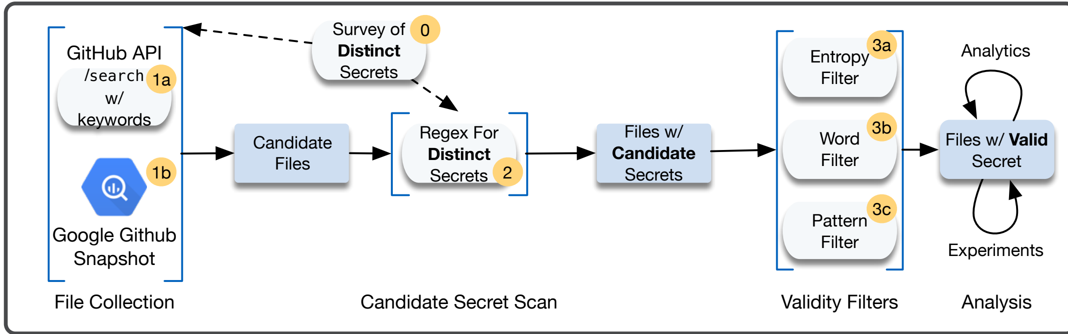


Fig. 1: Our secret collection methodology involves various phases to identify secrets with high confidence

public dataset in Google BigQuery [28]. This provided a large sample of repositories, especially those that may have been updated before we began our continuous search on GitHub. We chose the BigQuery snapshot instead of alternative collections of GitHub data (e.g. GHTorrent) [31] because BigQuery contains searchable file contents. Creating this dataset ourselves by cloning and examining each repository was infeasible due to computational constraints and GitHub rate limits.

In Phase 2, we used the regular expressions developed in Phase 0 to scan the candidate files from Phase 1 and identify "candidate secrets." Candidate secrets were then scanned by three filters in Phases 3a, 3b, and 3c to flag and remove candidate secrets that were unlikely to be valid. After the filtering in Phase 3, we considered the remaining candidates to be "valid secrets" that we then used in later analyses. We note that the secrets classified in Phase 3 as "valid" are not always "sensitive." For example, an RSA key used in an OpenSSL unit test may be valid — because it is, in fact, a key — but is non-sensitive as its secrecy is not required. We examine the impact of this issue on our results in Section V-B.

A. Phase 0: Survey of Popular APIs

Identifying secrets within code or data files can be a difficult task as secrets take on a wide variety of forms depending on their type, application, and platform. As shown by Phase 0 in Figure 1, we worked to identify a set of keys that conform to highly distinct structures. Because of their distinct structures, these keys are unlikely to occur by random chance, and so their detection gives high certainty in their validity. We call these *types* of keys our "distinct secrets." For these distinct secrets, we manually constructed "distinct secret regular expressions" that could be used in a later phase to extract candidate secrets with high confidence from a given input file. In total, we identified 15 API key types and 4 asymmetric private key types that have distinct signatures. While these types are not exhaustive, they represent many of the most popular secrets in use by software developers, and their highly distinct structures allow us to construct a high confidence lower bound evaluation of leakage on GitHub.

1) API Keys: Some popular API services add a distinct signature to their randomly generated values when creating API secrets. For example, all Amazon AWS Access Key ID values start with the string `AKIA` and Google API keys start with `AIza` [53]. Such an approach does not degrade the security of the API secret in terms of its randomness, but it does make searching for leaked keys significantly easier.

We began looking for services with distinct keys by enumerating all websites and services in the Alexa Top 50 Global and United States lists and in an open-source list of popular public APIs [49]. Next, we searched these lists to identify around 50 well-known and commonly-used services that provide a public API and whose key leakage would entail a security risk. We evaluated security risk by analyzing the functional scope of the APIs to determine how the different services could be abused; for example, AWS keys could be used to authorize expensive computation (monetary risk) or to access and modify data in cloud storage (data integrity and privacy). Finally, for each high-risk API, we registered and created 10 unique sets of developer credentials to confirm whether the provided secrets showed a distinct signature and, if so, we manually developed a regular expression that tightly matched those secrets. In total, we were able to compile signatures for 11 unique platforms (e.g., Google) and 15 distinct API services (e.g., Google Drive), of which 5 of the platforms and 9 of the APIs are for websites in the Alexa Top 50 for the U.S. at time of writing. These APIs, their keys, and their respective risks if compromised are shown in Table I. The regular expression we used for each key can be found in Table III in the Appendix.

The listed API keys have varying levels of secrecy and complexity to compromise because they may require additional information to be fully utilized. For example, sensitive Amazon AWS requests require both the Access Key ID, which has a distinct structure, and the Access Key Secret, which does not. Similarly, we note that Google's OAuth ID is often not considered secret, but its presence can allow us to locate an adjacent OAuth secret. In Table I, we distinguish keys that require additional pieces of information as "multi-factor secrets", while keys that are used alone are classified as "single-factor secrets". Importantly, we show in Section V-D that compromising multi-factor secrets is not difficult because we can leverage the distinct secret to identify associated secrets with high probability. Another potential hurdle for compromise is that some platforms allow users to place restrictions on their keys. For example, attacking OAuth flows may be difficult due to restrictions placed on redirect URIs, although compromise may still be possible with misuse and misconfiguration [14], [15], [43], [56]. Despite these considerations, in general, all identified keys lead to information whose leakage would

TABLE I: Keys for many popular APIs have distinct structures whose compromise would result in security risk

Domain	Platform/API	Key Type	Single-factor or Multi-factor	Primary Risks			
				Monetary Loss	Privacy	Data Integrity	Message Abuse
Social Media	Twitter	Access Token	M		X	X	X
	Facebook	Access Token	S		X	X	X
	YouTube [a]	API Key	S	X	X		
		OAuth ID	M		X	X	X
	Picatic	API Key	S		X	X	X
Finance	Stripe	Standard API Key	S	X		X	
		Restricted API Key	S				
	Square	Access Token	S	X		X	
		OAuth Secret	S				
	PayPal Braintree	Access Token	S	X		X	
	Amazon MWS	Auth Token	M	X	X	X	X
Communications	Gmail	(same as YouTube)[a]	(same as YouTube)[a]		X	X	X
	Twilio	API Key	S		X	X	X
	MailGun	API Key	S		X	X	X
	MailChimp	API Key	S		X	X	X
Storage	Google Drive	(same as YouTube)[a]	(same as YouTube)[a]		X	X	
IaaS	Amazon AWS	Access Key ID	S	X	X	X	
	Google Cloud Platform	(same as YouTube)[a]	(same as YouTube)[a]	X	X	X	
Private Keys	RSA	Cryptographic key	M	X	X	X	X
	EC	Cryptographic key	M	X	X	X	X
	PGP	Cryptographic key	M	X	X	X	X
	General	Cryptographic key	M	X	X	X	X

[a] These secrets share the same format as part of the Google platform, but have different risks and are thus considered different

compromise the security of an account, irrespective of difficulty.

2) Asymmetric Private Keys: Asymmetric cryptography is used frequently for many applications. For example, authentication over SSH often uses a private key located in one's `~/.ssh/id_rsa` file, or certificate-based authentication for OpenVPN may include the private key within the `*.ovpn` configuration file. In many cases, the private key will be stored in Privacy-enhanced Electronic Mail (PEM) format, which is identifiable due to its header consisting of the text `-----BEGIN [label]-----`, where `label` may be one of many strings such as `RSA PRIVATE KEY` [38], [42]. We identify 4 commonly leaked types of private keys, including those generated using popular tools such as `ssh-keygen`, `openssl`, and `gpg`, shown in Table I. The regular expression for each can be found in Table IV in the Appendix.

B. Phase 1A: GitHub Search API File Collection

In this section, we describe the first approach for collecting candidate files to be scanned with our distinct secret regular expressions, shown as Phase 1a in Figure 1. GitHub provides a search engine API that allows users to query repositories for code contents, metadata, and activity [22]. We carried out a longitudinal analysis of GitHub by continually querying this API for almost 6 months, from October 31, 2017 to April 20, 2018. Because this API [22] provides results in near real-time as files are pushed to GitHub, *all search results are from actively developed repos*.

The Search API is a flexible, powerful tool, but it does have two limitations that we had to address: no support for regular expressions and set limits on call rate and result count. Querying the Search API requires two parameters: the query string and the sort type. Unfortunately, advanced search techniques such as regular expressions are not supported in the query string [24]. To address this limitation, we first created a set of queries that would identify files likely to contain secrets. These queries on their own are not sufficient to find secrets, but we are able to download the resulting files and then scan them offline with our regular expressions in Phase 2. There were two separate groups of queries that we executed: (1) general queries against any potential secret without targeting a specific platform (e.g., `api_key`) and (2) specific queries created to target the distinct secrets identified in Section III-A derived from their regular expression (e.g., `AKIA` for Amazon AWS keys). These queries are shown in Table V in the Appendix. For the sort type parameter, we always used `sort=indexed` which returns the most recently indexed results, ensuring we received real-time results. We excluded `.gitignore` files from these results as they rarely contained secrets but made up a large percentage of search results[1]. For each query, the API returned a collection of files and their metadata. We then perform another request to the API's content endpoint [18] to get the contents of the file.

GitHub provides stipulations on their search platform, namely that only a maximum of 1,000 results are returned and only files less than 384KB are indexed for search [22], [24]. In addition, GitHub imposes a rate limit; an authenticated user may only perform 30 search queries a minute [23] and a separate total of 5,000 non-search queries an hour [17]. In our experiments, each individual query required at most 10 search requests and 1,000 non-search queries for contents. Only 5 queries could be carried out an hour in this manner. However, since many of the search queries do not generate 1,000 new results per hour, we could only collect files that were new to the dataset to reduce API calls. This way, we can run all queries every thirty minutes within the rate limits using a single API key. We show in Section VI-A that this interval achieves 99% coverage of all files on GitHub containing our queries. *Ultimately, the rate limit is trivial to bypass and is not a substantial obstacle to malicious actors.*

[1] We return to the contents of `.gitignore` files in Section VIII.

C. Phase 1B: BigQuery GitHub Snapshot File Collection

In addition to using GitHub's Search API, we also queried GitHub's BigQuery dataset in Phase 1B. GitHub provides a queryable weekly snapshot of all open-sourced licensed repositories via Google BigQuery [27]. All repositories in this dataset explicitly have a license associated with them, which intuitively suggests that the project is more mature and intended to be shared. This snapshot contains full repository contents and BigQuery allows regular expression querying, so we were able to query against the contents with our distinct secret regular expressions from Section III-A to obtain files containing matching strings. Unfortunately, BigQuery's regular expression support is not fully-featured and does not support the use of negative lookahead or lookbehind assertions, and so the query results were downloaded for a more rigorous offline scan in later Phase 2, similar to in Phase 1A.

While both our file collection approaches queried GitHub data, the two approaches allowed analysis of two mostly non-overlapping datasets. BigQuery only provides a single snapshot view of licensed repos on a weekly basis, while the Search API is able to provide a continuous, near-real time view of all public GitHub. Using both approaches simultaneously gave us two views of GitHub. We collected BigQuery results from the snapshot on April 4, 2018.

D. Phase 2: Candidate Secret Scan

Via Phase 1, we collected a large dataset of millions of files potentially containing secrets. Next, we further scanned these files offline using the distinct secret regular expressions to identify files actually containing secrets and to extract the secrets themselves. This process yielded a set of candidate secrets that could undergo additional validation in a later step. This scanning process is shown in Phase 2 of Figure 1.

Recall that limitations meant the files from the Search API and from BigQuery in Phase 1 were retrieved using methods that could not guarantee they contained a matching distinct secret. These files were downloaded to be evaluated offline against the distinct secret regular expressions from Phase 0. In Phase 2, we performed this offline scan and noted files and strings that match one or more of the regular expressions. Note that each regular expression was prefixed with negative lookbehind `(?<![\w])` and suffixed with negative lookahead `(?![\w])` to ensure that no word characters appeared before or after the regular expression match and improve accuracy. The set of strings that resulted from this scan were classified as "candidate secrets".

E. Phase 3: Validity Filters

It is possible that the candidate secrets provided by Phase 2 were not actually secret, although they matched a regular expression. In Phase 3, we passed the candidate secrets through three independent filters that worked to identify whether a given string should be considered "valid". We define a valid secret as a string that is a true instance of the distinct secret for which it matches. As an example, consider that the regular expression for the Amazon AWS secret, `AKIA[0-9A-Z]{16}`, would match the string `AKIAXXXEXAMPLEKEYXXX`, which is likely not valid, while `AKIAIMW6ASF43DFX57X9` would be.

Unfortunately, it is a non-trivial task to identify a string as being a valid secret for a certain target with complete accuracy, even for a human observer. Intuitively, the best approximation that a human observer could make is whether the candidate secret appears random. We were inspired by and improve upon the algorithms used by TruffleHog [59] and an open-source neural-networked-based API key detector [12] to build our filters. The filters perform three checks against a string: that (1) the entropy of the string does not vary significantly from similar secrets, (2) the string does not contain English words of a certain length, and (3) the string does not contain a pattern of characters of a certain length. A string failing any one of these checks is rejected by the filter as invalid; all others are accepted as valid. The valid secrets were stored in a database and were used for all later analysis.

Entropy Filter While there is no guaranteed mathematical test for randomness, a good estimator is Shannon Entropy. Consider a discrete random variable X. We wish to quantify how much new information is learned when X is observed. The Shannon Entropy formula defines the average amount of information transmitted for X [6], [12]:

$$H(X) = -\sum_{i=0}^{n} P(x_i) \log_2 P(x_i) \quad (1)$$

where X has possible values $x_1, ..., x_n$ and $P(x_i)$ is the probability for X to be the value x_i. Intuitively, a random string should consist of a rare values, giving it a high entropy; on the other hand, English text has been shown to have fairly low entropy—roughly one bit per letter [12], [51].

To apply this stage of the filter, our goal was to eliminate all strings that deviated significantly from the average entropy for a given target secret. To do this, we failed a string for this check if its entropy was more than 3 standard deviations from the mean of all candidate strings for a given target secret, effectively classifying it as an outlier. This approach relied on the candidate set containing almost exclusively valid secrets; we determined that this was the case for almost all targets, and for those where it were not, the other stages of the filter could still be applied.

Words Filter Another intuition is that a random string should not contain linguistic sequences of characters [12]. For this check, we compiled a dictionary of English words of length as least as long as a defined threshold. Then we searched each candidate string for each one of these words and failed the check if detected.

A trade-off exists in choosing this threshold. If it is too small, randomly occurring sequences that happen to create words will create false negatives (marking valid secrets as invalid), but if it is too large, legitimate words will be missed and create false positives (marking invalid secrets as valid). In our experiments, we set the word length threshold to be 5. This threshold was chosen as a best judgment after careful manual review; unfortunately, experimental derivation of this threshold was not possible given limited initial ground truth.

A dictionary of every English word would contain words that would not likely be used as part of a string in a code file and cause high amounts of false negatives. Therefore, we took the intersection of an English dictionary [45] and a dictionary of the

most common words used in source code files on GitHub [40]. The resulting dictionary contained the 2,298 English words that were likely to be used within code files, reducing the potential for false negatives.

Pattern Filter Similar to linguistic sequences of characters, random strings should also not contain mathematical sequences of characters [12]. We identified three such possible patterns to search for: repeated characters (e.g., `AAAA`), ascending characters (e.g., `ABCD`), and descending characters (e.g., `DCBA`). To apply this check, we searched each candidate string for one of these patterns at least as long as a defined threshold and failed the check if detected. We settled on a pattern length threshold of 4, with the same trade-off considerations addressed previously addressed.

IV. Ethics and Disclosure

This paper details experiments collecting over 200,000 leaked credentials to services that could have serious consequences if abused. In this section, we discuss issues related to the ethical conduct of this research.

First and foremost, our institutional review board (IRB) informed us that our data collection and analysis was exempt from review because we only work with publicly available data, not private data or data derived from interaction with human participants.

Second, apart from our search queries, our methodology is passive. All secrets that we collect were already exposed when we find them, thus this research does not create vulnerabilities where they did not already exist. We took a number of steps to ensure that our collected secrets are unlikely to leak further. These include running all experiments from a locked-down virtual machine and ensuring that no secret data leaves our collection database. Access to the database is only possible through public-key-authenticated SSH.

Furthermore, we *never* attempt to use any of the discovered secrets other than for the analytics in this paper, even for innocuous purposes like to merely verify the secret could be used successfully. This prevents the possibility that a secret owner might be negatively impacted by our use of such a secret, for example by causing a billing event or hitting a rate limit cap. It also prevents us from obtaining any sensitive, proprietary, or personal information from the secret owner. Finally, it prevents the remote possibility that a service could be negatively impacted by our testing.

Finally, as of the camera-ready we are currently working to notify vulnerable repository owners of our findings.

V. Secret Leakage Analysis

In this section, we use our collection of discovered secrets to characterize how many projects on GitHub are at risk due to secret exposure. Primarily, our focus is on identifying how many exposed secrets are truly sensitive—we consider a "sensitive" secret as one whose leakage is unintentional and discovery presents security risk to the owner. First, we report high-level statistics on the large numbers of exposed secrets that we discovered via both data collection approaches (Section V-A). Then, we show that most discovered secrets are likely sensitive through a rigorous manual experiment (Section V-B). Next, we compare single- and multiple-owner secrets to further confirm the aforementioned section (Section V-C). We also demonstrate that finding one secret can be leveraged to discover other secrets with high probability (Section V-D). We show many secrets are very infrequently removed from GitHub and persist indefinitely (Section V-E). Finally, we focus specifically on RSA keys to exemplify how an attacker could abuse exposed keys (Section V-F).

A. Secret Collection

In this section, we provide high-level statistics on the set of secrets we discovered. We detail the number of files at each step in our collection methodology and culminate in the total number of unique secrets that we discover. Here, we refer to a "unique" secret as a secret that appears at least once in the dataset; note that a unique secret can occur multiple times.

GitHub Search API The GitHub Search API collection began on October 31, 2017 and finished on April 20, 2018. During this period of nearly 6 months, we captured 4,394,476 candidate files representing 681,784 repos (Phase 1a of Figure 1), from which our distinct secret regular expression scan (Phase 2 of Figure 1) identified 307,461 files from 109,278 repos containing at least one candidate string, giving a file hit rate of approximately 7%.

Overall, we identified 403,258 total and 134,887 unique candidate strings that matched our regular expressions. In addition, our search collection collected a median of 4,258 and 1,793 unique candidate secrets per day, with a range from 2,516 to 7,159 total.

As discussed, some of the strings that match the regular expression could be invalid secrets. We applied our filtering heuristics to determine the number of valid secrets from candidate strings (Phase 3 of Figure 1). In total, we found that 133,934 of the unique candidate strings were valid, giving an overall accuracy of 99.29% for the distinct signature regular expressions used in Phase 2.

GitHub BigQuery We performed our query on a single GitHub weekly BigQuery snapshot on April 4, 2018. We were able to scan the contents of 2,312,763,353 files in 3,374,973 repos (Phase 1b of Figure 1). We identified at least one regular expression match in 100,179 of these files representing 52,117 repos (Phase 2 of Figure 1), giving a file hit rate of approximately 0.005% across all open-source GitHub repositories in BigQuery. Within the files with matches, we identified 172,295 total strings and 73,799 unique strings, of which 73,079, or 98.93%, were valid (Phase 3 of Figure 1).

Dataset Overlap Some of our secrets may appear in both datasets since a file we see via the Search API could be contained within the BigQuery snapshot, or a secret may simply be duplicated in different files. After joining both collections, we determined that 7,044 secrets, or 3.49% of the total, were seen in both datasets. This indicates that our approaches are largely complementary.

Breakdown by Secret Table II breaks down the total and unique numbers of secrets by distinct secret. The most commonly leaked were Google API keys. RSA private key leakage was also common, although leakage of other keys, such as PGP and EC, was orders of magnitude lower. Many of our

TABLE II: The majority of secrets in the combined dataset are used by a single-owner

Secret	# Total	# Unique	% Single-Owner
Google API Key	212,892	85,311	95.10%
RSA Private Key	158,011	37,781	90.42%
Google OAuth ID	106,909	47,814	96.67%
General Private Key	30,286	12,576	88.99%
Amazon AWS Access Key ID	26,395	4,648	91.57%
Twitter Access Token	20,760	7,935	94.83%
EC Private Key	7,838	1,584	74.67%
Facebook Access Token	6,367	1,715	97.35%
PGP Private Key	2,091	684	82.58%
MailGun API Key	1,868	742	94.25%
MailChimp API Key	871	484	92.51%
Stripe Standard API Key	542	213	91.87%
Twilio API Key	320	50	90.00%
Square Access Token	121	61	96.67%
Square OAuth Secret	28	19	94.74%
Amazon MWS Auth Token	28	13	100.00%
Braintree Access Token	24	8	87.50%
Picatic API Key	5	4	100.00%
TOTAL	**575,456**	**201,642**	**93.58%**

API keys had relatively small incidents of leakage, likely due to lower popularity of those platforms in the type of projects on GitHub. Most importantly, we were able to identify multiple secrets for every API we targeted.

B. Manual Review

Throughout this paper, we use statistical approaches and heuristics to estimate the prevalence of secrets on GitHub. To validate these results, we carried out a rigorous manual review on a sample of the dataset. We collected a random sample of 240 total candidate secrets, evenly split between Amazon AWS and RSA keys. Two of three raters (all paper co-authors) examined the file and repo containing the secret on GitHub's website. After considering the context of the secret, the raters evaluated each secret as sensitive, non-sensitive, indeterminate, or not a secret. Once every secret was coded, we evaluated the interrater reliability of the two raters. We found a total of 88.8% of the judgments were in agreement with a Cohen's Kappa of 0.753, lending confidence in the result. All disagreements were mediated by the third rater, who independently rated each disagreeing case without knowledge of the prior codings, and then were settled by group consensus. In the results that follow, we exclude secrets that could not be determined sensitive or non-sensitive (5 total) or that were not valid secrets (4 total)[2].

We used these findings to estimate the overall sensitivity of our entire data. We considered the sensitivity of AWS keys representative of all API keys and the sensitivity of RSA keys representative of all asymmetric keys. We then scaled the percentages determined by the manual review experiment against the base rate of each sub-type in our dataset. We estimated that 89.10% of all discovered secrets are sensitive. *If we consider API and asymmetric key secrets separately, we estimated that 93.74% of API secrets and 76.24% of asymmetric keys are sensitive. This indicates that most of the discovered secrets are sensitive and many users are at risk.*

[2]We drew our random selection against non-filtered "candidate" secrets, so the number of invalid secrets in this sample is not representative of the effectiveness of our overall analysis pipeline.

C. Single- and Multiple-Owner Secrets

The results in Table II show a level of duplication of secrets within our collection as the number of unique secrets is less than the number of total secrets. Since we previously defined a secret as a credential whose privacy must be maintained for security, we evaluated this duplication to determine whether it indicated our results were skewed towards non-sensitive secrets. Intuitively, a secret should be kept private to the single individual who "owns" it. While it would be a valid use case to see duplication due to an individual using the same sensitive secret in multiple files or repos, it would be unlikely to see multiple users do so.

To verify this intuition, we further analyzed the results of the Manual Review experiment from Section V-B. First, we defined a secret with one owner as a "single-owner secret," and a secret with multiple owners as a "multiple-owner secret." As we were unable to identify the committer of a secret, and because our data sources did not easily provide contributor information[3], we considered the repo owner as the entity who owned the secret. Of the 240 secrets examined, we had also evenly split the secrets between single- and multiple-owner secrets, allowing us to examine whether there was a difference in sensitivity between single- and multiple-owner secrets for AWS and RSA keys. At a high-level, 91.67% of single-owner AWS keys were sensitive compared to 66.67% multiple-owner AWS keys, and respectively 75% versus 38.33% for RSA keys. For AWS keys, we found a statistically significant difference with a medium effect size ($\chi^2 = 15.2, p < 10^{-4}, r > 0.56$), and for RSA keys, we found a statistically significant difference with a large effect size ($\chi^2 = 35.7, p < 10^{-5}, r > 0.56$). These findings confirmed our assertion that single-owner secrets are more likely to be sensitive.

With our intuition confirmed, we classified every secret in our dataset as single- or multiple-owner to evaluate the impact of duplication. Table II shows the results of this classification on the combined Search and BigQuery datasets. We show that an overwhelming majority (93.58%) of unique secrets are found in repos owned by a single owner, indicating that these are more likely to be sensitive secrets[4]. Further, we computed the Pearson correlation coefficient between the relative rates of single- and multiple-owner secrets between the Search and BigQuery datasets. We found that the two datasets had a correlation of $r = 0.944$ and a p-value of 1.4×10^{-9}, indicating that they have a similar level of exposure and distribution of sensitive secrets, irrespective of their size and perspective.

Critically, because almost all detected secrets had their privacy maintained, we show that the observed duplication does not suggest our results were skewed by non-sensitive secrets. In fact, deeper investigation showed one major source of duplication was a single developer using their secrets multiple times; in the Search dataset, we found that the average single-owner secret was used in 1.52 different files, with the most duplicated appearing in 5,502 files. A second source of duplication was from a very small number of secrets used

[3]BigQuery does not provide this information. It is possible to obtain it via the GitHub API, but not at a large scale due to rate limiting.

[4]Technical limitations prevented us from retrieving repo owner information for about 7,500 secrets from BigQuery, which where excluded from this analysis.

by many developers. This was particularly evident with RSA private keys, where nearly 50% of all occurrences were caused by the most common unique 0.1% keys, which were multiple-owner secrets and likely test keys. Fortunately, since this source duplication was restricted to a very small subset of keys, duplication would have minimal impact on analysis done on unique valid secrets. *Consequently, we will only consider the unique valid secrets in future sections of this paper*.

D. Parallel Leakage

Some of our secrets require additional pieces of information to be used, such as Google OAuth IDs which require the OAuth Secret for privileged actions. We previously defined these secrets as "multi-factor" secrets in Section III-A and identified them in Table I. While these parallel secrets may seem to improve security by reducing the impact of leakage, in this section we show that the missing information is often leaked in parallel to the main secret, making this protection mostly inconsequential. The difficulty in detecting the parallel secrets is that they may not have a sufficiently distinct structure to be included within our distinct signatures. However, they can still be matched by a crafted regular expression and located with high confidence given prior knowledge of secret leakage. We examined every file containing a distinct multi-factor secret and then scanned for the parallel secrets[5] in the 5 lines before and after a secret. This context size was chosen based on prior work that scanned Google Play applications [62].

Figure 2 shows the results of this experiment in terms of the percent of files containing one of our secrets that has a parallel secret. *Every multi-factor secret in the Search dataset has at least an 80% likelihood of leaking another parallel secret*. For example, even though Google OAuth IDs require another secret, our ability to write regular expressions to identify them with high fidelity allows us to discover one of the other secrets in nearly 90% of cases. BigQuery shows lower rates of parallel leakage, perhaps due to the data source containing more mature files, but still has a worrying amount of leakage. Thus, we argue that the fact that these multi-factor secrets have varying levels of compromisability and secrecy is not a large hurdle.

Additionally, this parallel leakage was not restricted to single types of secrets; many files containing one secret also contained another secret. We identified 729 files that leaked secrets for two or more API platforms within the same file.

E. Secret Lifetime

Once a secret is exposed by a user, a user may attempt to retroactively remove the secret via a subsequent commit. To quantify the prevalence of this, we began monitoring all secrets collected via the Search API after they were discovered starting on April 4th, 2018. For the first 24 hours from discovery, we queried GitHub hourly to determine if the repo containing the file, the file itself, and the detected secrets still existed on the head of the default branch. After the initial 24 hours, we performed the same check at a reduced daily frequency. The results of the initial 24 hour short-term monitoring is shown in Figure 3a, and the daily long-term monitoring is shown in Figure 3b.

[5]The parallel targets we scanned for, and their regular expressions, can be found in Table VI in the Appendix

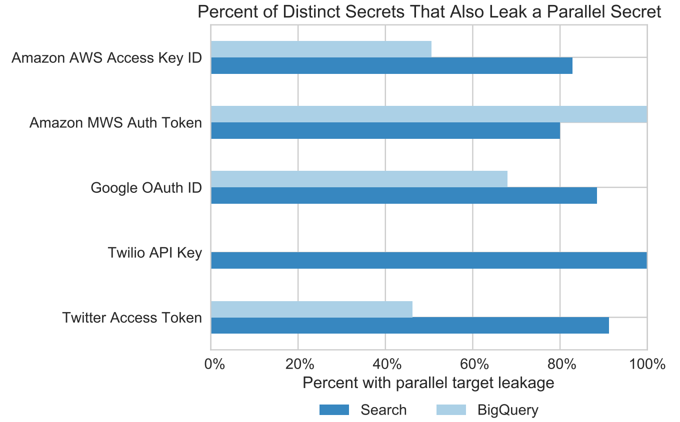


Fig. 2: All of the multi-factor distinct secrets have a high rate of leakage of other pieces of secret information

We observe several trends. First, the largest drop in secret presence occurred in the first hour after discovery, where about 6% of all detected secrets were removed. Second, secrets that existed for more than a day tended to stay on GitHub long-term—at the end of the first day, over 12% of secrets were gone, while only 19% were gone after 16 days. Third, the rate at which secrets and files were removed dramatically outpaces the rate at which repos were removed; this indicates that users were not deleting their repos, but were simply creating new commits that removed the file or secret. Unfortunately, due to the nature of the Git software, the secrets are likely still accessible [50] (see Section VIII for more on this issue).

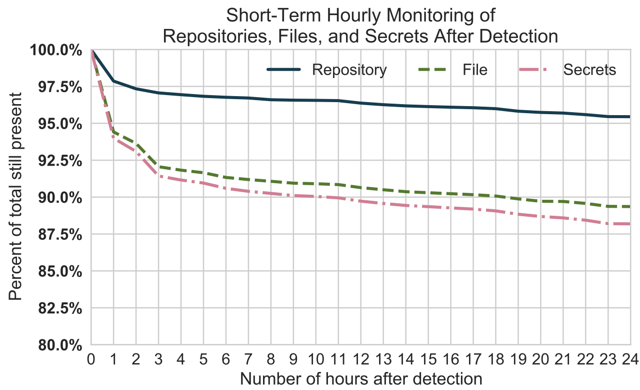
These conclusions suggest that many of the secrets discovered were committed in error and that they were sensitive. 19% of the secrets were removed at some point in roughly 2 weeks, and most of those were done in the first 24 hours. This also means 81% of the secrets we discover were *not* removed. It is likely that the developers for this 81% either do not know the secrets are being committed or are underestimating the risk of compromise. In absolute terms, 19% of our results amounts of tens of thousands of secrets and serves as a lower bound on the number of discovered secrets that were sensitive, adding confidence to our overall result.

Further, we examined whether users that removed their secrets while keeping their repos performed any process to rewrite history to remove the commit, as suggested by GitHub [50]. For every such instance, we queried the GitHub Commits API for information on the commit we discovered; if the commit had been rewritten, it would no longer be accessible. We found that none of the monitored repos had their history rewritten, meaning the secrets were trivially accessible via Git history.

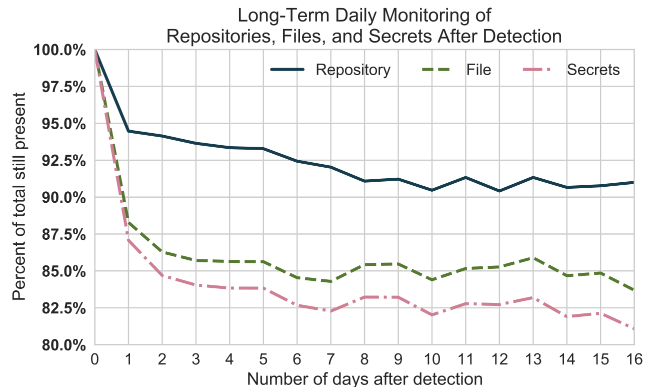
F. RSA Key Leakage

Table II shows a large portion of secrets in our dataset were RSA keys, which is expected as they are used for a large number of different applications. We performed various experiments to investigate how many of these RSA keys pose a significant risk if discovered.

Number of Valid Keys RSA keys contain a defined and parseable structure. Thus, we can determine how many of these



(a) Many secrets are removed in the first few hours after being committed, but the majority remain



(b) Secrets that still exist on GitHub for a day after commit tend to stay on GitHub indefinitely

Fig. 3: Short and Long Term Monitoring of Secrets

keys were valid using the Paramiko library [47]. Out of the 25,437 secrets discovered via the Search API, we found that 25,370 keys, or 99.74%, were valid. From the BigQuery dataset, of the 15,262 keys, 98.31% or 15,004 keys were valid.

Number of Encrypted Keys The Public Key Cryptography Standard (PKCS) allows for private keys to be encrypted [39]. While leaking a key is never a good idea, an attacker will have a much more difficult time compromising a leaked key if the key is encrypted. Again, we used the Paramiko [47] library, which can determine when a key is encrypted, on our keys to count how many were encrypted. *From this experiment, we found that none of the leaked keys in the Search and BigQuery datasets were encrypted, meaning an attacker could easily use every single one.*

OpenVPN Config Analysis Another application of RSA keys is usage in OpenVPN configuration files, in which keys may be embedded for client authentication to the VPN server. As an additional layer of protection, OpenVPN recommends clients specify the `auth-user-pass` option in the config file. This option requires a user to also enter a valid password to be connected to the VPN, which makes use of a stolen key more difficult. In order to determine whether an attacker could gain unauthorized access to VPN servers, we analyzed how many OpenVPN configs containing RSA keys existed in our dataset by looking for files with the `.ovpn` extension and investigated whether they could be used without further effort.

In the Search dataset, we identified 1,890 total OpenVPN config files in our dataset. Critically, 13.18% of these did not use the `auth-user-pass` option, meaning the user could easily be compromised by an attacker. In the BigQuery dataset, we identified 5,390 total OpenVPN config files, of which 1.08% were vulnerable. There is a discrepancy between the two datasets, likely because licensed repos are more mature and contain more example files, but both datasets still revealed a substantial number in absolute terms.

VI. Methodology Evaluation

In this section, we evaluate the key aspects of our methodology. First in Section VI-A, we consider our data collection methods. Second in Section VI-B, we evaluate the efficacy of our secret validation filters.

A. Search API Evaluation

The Search API collection gave us near real-time insight into files committed to GitHub. We performed two experiments to validate this method. First, we found how long it took for a known random string to appear in search results after being committed. Second, we measured the percent of files we could see using a single API key with the Search API out of all files committed within a time period.

Time to Discovery If the time for GitHub to index and return search results is long, users have a large period of time to remove their secret before we could detect it. To determine this time to discovery, we set up an experiment in which a known string was pushed to a known repository. Immediately after pushing, we started a timer and began querying the API continuously until the string appeared, at which point the timer was stopped. This time measures the time to discovery. We ran this experiment once a minute over a 24 hour period to compensate for variations with time of day.

We found that the median time to discovery was 20 seconds, with times ranging from half a second to over 4 minutes, and no discernible impact from time-of-day. Importantly, this experiment demonstrates that our Search API approach is able to discover secrets almost immediately, achieving near real-time discovery of secrets. If a user does realize their secrets were mistakenly pushed, this leaves little time for them to correct the problem before their secrets would be discovered.

Coverage Since we only used a single GitHub key for Search API scanning, we were restricted to searching at 30 minute intervals to avoid rate limiting issues. Running at this rate created the possibility of missing results that were pushed and removed between the scanning times. To evaluate how much coverage we could achieve at our interval, we set up the following experiment. For each query (running individually to get better scanning granularity), we did an initial pull of results on GitHub (the $start_set$). For the next 30 minutes, we constantly pulled all results for the query from GitHub non-stop. This constant pull represented the theoretical maximum of results that we could potentially have pulled with no rate limiting (the max_set). Finally, we pulled a final set of results after the interval (the end_set). Then, our total coverage is

given by:

$$coverage = 1 - \frac{|max_set - (start_set \cup end_set)|}{|max_set|} \quad (2)$$

This experiment was repeated over a period of 3 days covering both a weekend day and weekdays.

We found that the overall coverage across all our queries was 98.92%. There was minimal fluctuation on the weekday as compared to the weekend, with each achieving 98.85% and 99.02% coverage, respectively. *This result shows that a single user operating legitimately within the API rate limits imposed by GitHub is able to achieve near perfect coverage of all files being committed on GitHub for our sensitive search queries.* Of course, a motivated attacker could obtain multiple API keys and achieve full coverage.

B. Regular Expression and Valid Secret Filter Evaluation

In this section, we discuss three experiments we performed to validate our regular expressions and filters. We previously defined a "valid" secret as a string that is a true instance of the distinct secret for which it matches. The strings matching our distinct secret regular expressions may not necessarily be "valid" secrets, and therefore we designed and implemented a series of filters on these strings to validate them using various techniques. After collecting a large number of candidate secrets, we were able to validate the performance of this filter.

Most Common Patterns and Words Recall that two of the key stages for the validity filters were eliminating strings containing patterns or English (and common GitHub) words. To gain insight into which patterns were causing secrets to be invalidated, we counted the individual patterns and words detected by the filters on our dataset. The only word which appeared more than 3 times in both datasets was "example", which appeared 9 times in Search and 20 times in BigQuery.

Patterns appeared more frequently. The most common pattern was "XXXX", which appeared 27 times in Search and 46 times in BigQuery. There were over 90 patterns that appeared more than 2 times. To count the relative occurrence, each pattern was grouped into one of three categories: ascending sequence, descending sequence, or repeated sequence. Ascending sequences were most common and comprised 52.91% of all patterns, while repeated sequences made up 31.24% and descending sequences made up 15.86% of the patterns.

The results of this experiment suggest that most of the keys being invalidated by the pattern filter are example keys or patterned gibberish sequences. Keys containing the word "example" are most likely example keys. Patterns of letters that perfectly fit one of the secret regular expressions are unlikely to happen by chance, which suggests that users are typing sequences on their keyboard to act as an example key.

Filter Validation Determining whether a given string is a valid secret is a difficult task to do with perfect accuracy as most secrets do not contain a verifiable structure. This challenge also makes automatic validation of our filters itself difficult. RSA private keys, however, do have a parseable and verifiable structure. These keys were also one of the largest sets of secrets we had. We validated our filters by checking that the RSA private keys we collected were valid. We used the Paramiko library [47] to try and parse each RSA key. This library throws a specific exception if a key is not parseable, allowing us to distinguish between errors due to parsing and other issues (e.g. encryption errors). All keys that threw non-parsing errors were excluded as we would not be able to verify these keys. After running the library, which provided ground truth, we ran our filters and analyzed its predicted performance.

In the Search dataset, we considered a total of 25,437 RSA keys. We find that our filters correctly confirmed 24,935 (98.03%) of them to be valid (24,918) or invalid (17). Furthermore, our filter only failed to invalidate 50 keys that were not parseable by the library. In the BigQuery dataset, we considered a total of 15,252 RSA keys, we find that our filter confirmed 97.97% of them to be valid or invalid, and 60 keys were not parseable by the library.

The results of this experiment provide mixed insights. On one hand, it shows that the filter was able to validate a large number of secrets with minimal inaccuracy. On the other hand, the experiment also reveals that the number of valid keys dramatically outweighs the number of invalid keys. In total, only 67 of the 25,437 were invalid. We believe that the targeted secrets have such distinct structures that truly invalid keys are rare and a minimal concern for an attacker, who would be able to simply test the keys themselves against the target service.

Regular Expression Accuracy In this section, we measure the performance of the distinct secret regular expressions. For each regular expression, we calculate precision as number of valid matches (validated by all filters) divided by the total number of matches of the regular expression. Unfortunately, because ground truth on all secret leakages on GitHub is simply not available, we are unable to compute recall.

The main concern was that a loose regular expression would match a large number of invalid strings and create many false positives. However, this was not the case with our distinct secret regular expressions. Our regular expressions achieved 99.29% precision in the Search dataset and 98.93% precision in the BigQuery dataset. The precision of each regular expression is shown in Figure 4. Only 4 of the distinct secret regular expressions gave more than a 2% rejection rate in both datasets, and these all had relatively small sample sizes. This shows that our regular expressions themselves are sufficient to determine valid keys for most of the secrets.

VII. Potential Root Cause Analysis

It is now clear that secret leakage on GitHub puts a large number of keys at risk. In this section, we demonstrate various experiments that try to address the question of why secret leakage is so extensive. Without asking developers directly, we are unable to determine the root cause for each individual leakage, but we are able to leverage our dataset to analyze a number of *potential contributing factors*. Ultimately, we find that most secret leakage is likely caused by committed cryptographic key files and API keys embedded in code.

A. Repository and Contributor Statistics

In order to evaluate whether the metadata of repos containing secrets and information about their owners could reveal interesting insights, we sourced such data from the GitHub API.

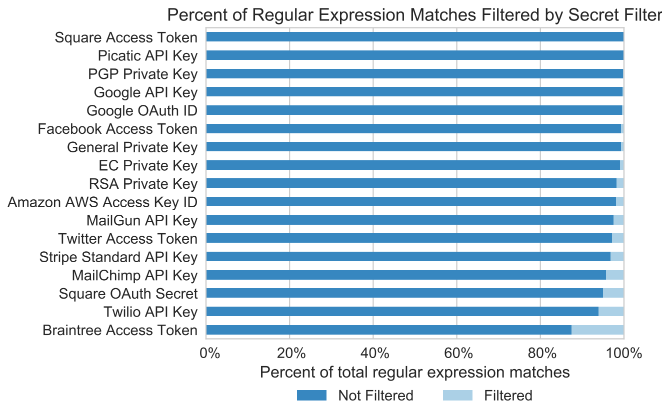


Fig. 4: Most of the regular expressions had high precision with minimal rejection by the filter

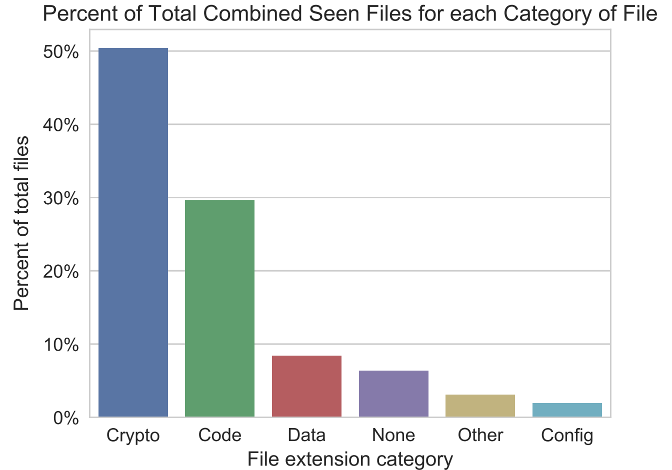


Fig. 5: Most detected secrets were found in cryptographic or source code files

For each repo, we collected the number of forks, watchers, and contributors. We chose these features as they indicate the level of activity, and it may be the case that more active repos are less (or more) likely to leak secrets. For each repo contributor, we queried the user's number of public repos, contributions in the repo, and contributions in the past year across GitHub. We chose these features because they act as a proxy for developer experience with GitHub, and we question whether less experienced developers are more likely to leak secrets or if it occurs irrespective of experience. For a comparison, we also collected a control group by randomly sampling approximately 100,000 GitHub repos and collecting the same information.

With both sets of metadata, we conducted a Mann-Whitney U-test to determine if there were meaningful differences on the variables. Our control dataset featured 99,878 repos and 360,074 developers, while our leakage dataset featured 95,456 repos and 180,101 developers. Because we examine large datasets, slight differences in the datasets have a strong probability of passing statistical significance tests. We chose a significance threshold of $\alpha = 0.01$, and applied a conservative Bonferroni correction to $\alpha = \frac{0.01}{10} = 0.001$ to account for all of our planned hypothesis tests.

We found no meaningful differences on the variables in the repo metadata that leak secrets compared to a randomly selected control dataset. All tests were statistically significant ($p < 10^{-5}$). We examined the effect size using the rank-biserial correlation method. We found that all effects are quite small ($r < 0.21$), indicating that there is likely not a large difference between the leakage sets and control sets based on these criteria. With the contributor metadata, we again found no meaningful differences on the variables. All tests were statistically significant ($p < 10^{-5}$) with a small effect size ($r < 0.27$), indicating no large difference between the two sets. *These results show that neither repo activity nor developer experience are strongly correlated with leakage.*

We also examined the distribution of forked repos in our data. We found that repos that leak secrets are far less likely to be forked, with only 0.11% of leaking repos being forked compared to 47.5% of the control. We used a χ^2 test and found that this difference is statistically significant with a strong effect ($p < 10^{-9}, r > 0.64$). This implies that secrets rarely propagate by forks, and that most leaks are in an original repo.

B. Top File Types

Since certain usage patterns may be more prone to secret leakage, a method of approximating this is to evaluate the most common types of files containing secrets. As there are many different file extensions, we grouped each extension into one of several defined categories. The "crypto" category contained files commonly used for storing keys and certificates (e.g., `.key`), "code" contained source code files (e.g., `.py`), "data" contained files used to serialize data (e.g., `.csv`), and "config" contained files used primarily for configuration (e.g., `.conf`). All other files with extensions were grouped into "other", while files without extensions were grouped into "none". For a detailed breakdown of the extensions included in each group, see Appendix Section A.

Figure 5 shows the relative share of files for each category in the combined dataset. Unsurprisingly, "crypto" files make up the largest percent of the dataset because private keys are the largest group of compromised secrets. "Code" groups also make up a very large percentage; this indicates that many secrets, mainly API secrets, are being embedded directly within source code. While we cannot say from this analysis alone why cryptographic keys are leaked, it is clear that the poor practice of embedding secrets in code is a major root cause.

C. Personal RSA Key Leakage

Many developers store personal files in GitHub repositories, so another research question is whether overall leakage could largely be attributed to this. Common examples of this are `dotfiles` repos, which people use to backup configuration files and folders [19]. One common "dotfolder" is `.ssh`, a directory which often contains SSH keys, commonly in a file named `id_rsa`. To approximate the prevalence of secret leakage through this source, we gathered metrics on how many RSA keys appeared within a repo containing `dotfiles` in the name, within a `.ssh` folder, or within an `id_rsa` file.

In the Search dataset, we found 1676 (6.61%) of all RSA keys appeared within an `id_rsa` file, 653 (2.57%) within

a `.ssh` folder, and 353 (1.39%) within a `dotfiles` repo. In the BigQuery dataset, we found 651 (4.26%) of all RSA keys appeared within a `id_rsa` file, 110 (.007%) within a `.ssh` folder, and 39 (.002%) within a `dotfiles` repo. The lower prevalence of this leakage in BigQuery is likely due to a developer being unlikely to license their personal files. These keys are also more likely to be personal keys, especially SSH keys, which could allow an attacker to compromise a user. In total, the low representation of `dotfiles` indicates this is not the main cause of secret leakage, but it is a non-trivial factor.

D. TruffleHog Analysis

Secret management is a difficult task and tools exist to assist developers. One such tool is TruffleHog [59], which is used in a local Git repository to check every commit for secrets. TruffleHog detects secrets by finding strings with an entropy above a pre-defined threshold or by matching one of a set of regular expressions. While this is similar to our work, TruffleHog has a more limited set of regular expressions and does no validation on their results to avoid false positives. Since users may rely on tools like TruffleHog, we analyze its performance by running every secret we detected through its algorithm.

Our results show that TruffleHog is largely ineffective at detecting secrets, as its algorithm only detected 25.236% of the secrets in our Search dataset and 29.39% in the BigQuery dataset. Every one of these secrets were detected by entropy, while only a fraction of them were double detected by regular expressions. This discrepancy is worrying as the TruffleHog developers are deprecating entropy detection in favor of regular expressions detection [59]. If this change had been in place, only 19.263% of secrets from Search and 22.29% of secrets from BigQuery would have been detected. Because this problem is caused by a set of weak regular expressions, this tool could be improved by the use of our techniques. While TruffleHog does detect some secrets, we do not believe it provides strong enough protection. Developers using tools such as this may believe they are safe, but may unwittingly be pushing secrets. Unfortunately, because TruffleHog runs locally, we were unable to measure the rate of usage.

E. Google Client Secret Files

As a convenience and in an attempt to aid security, Google provides developers with a `client_secret_*.json` file, where the asterisk would be replaced with a unique identifier. Google recommends that you download this file for using your OAuth credentials instead of embedding the values in code directly, stressing clearly that the file should be stored out of the source tree [30]. To evaluate whether developers took heed of this warning, we analyzed every file containing Google OAuth IDs that matched the `client_secret_*.json` pattern. Ideally, these files should not exist on GitHub; however, we identified 2,155 of these files in the Search dataset and 388 in the BigQuery dataset. Further, this accounted for 5.027% of all files containing Google OAuth IDs in the search dataset, and 2.246% in the BigQuery dataset. The leakage of this file is particularly worrying as it contains other secret information, such as the OAuth Secret, in an easily parseable format. While this file mismanagement is not the primary source of leakage, it is certainly a substantial factor.

VIII. Case Studies

In this section, we discuss several interesting case studies discovered in the course of our research.

Secrets in `.gitignore` Files The `.gitignore` file is intended to allow developers to specify certain files that should not be committed, and it can be used (among other things) to prevent files containing secrets from being leaked. While this is a good strategy, many developers do not use this option and some do not fully understand it. To better investigate its usage, we used the Search API to collect `.gitignore` files over a 3 week period in the same manner as our normal collection process. Our assumption had been that these files would not contain secrets as they should only contain path references to files. Yet, we identified 58 additional secrets from this process. While this is a small number compared to the full dataset, this finding indicates that some developers commit secrets out of fundamental misunderstandings of features like `.gitignore`.

YouTube Copyright Infringement Our data collection methodology uncovered an interesting case study in which a GitHub user appeared to be conducting copyright infringement. A single user was hosting repositories containing source code for different websites that pull videos from YouTube and rehost them. We found a total of 564 Google API keys in these repositories, along with indications that they were being used to bypass rate limits. Because the number of keys is so high, we suspect (but cannot confirm) that these keys may have been obtained fraudulently. We could not locate the keys elsewhere in our limited view of GitHub, but it is possible that the keys came from elsewhere on GitHub, other public leaks, or accounts sold on the black market. Further, this example demonstrates the potential misuse of leaked secrets by a malicious actor.

High-Value Targets Our data shows that even high-value targets run by experienced developers can leak secrets. In one case, we found what we believe to be AWS credentials for a major website relied upon by millions of college applicants in the United States, possibly leaked by a contractor. We also found AWS credentials for the website of a major government agency in a Western European country. In that case, we were able to verify the validity of the account, and even the specific developer who committed the secrets. This developer claims in their online presence to have nearly 10 years of development experience. These examples anecdotally support our findings in Section VII-A that developer inexperience is not a strong predictor of leaks.

Rewriting History Does Not Protect Secrets It is obvious that adversaries who monitor commits in real time can discover leaked secrets, even if they are naively removed. However, we discovered that even if commit histories are rewritten, secrets can still be recovered. In investigating the previous European case study, we discovered that we could recover the full contents of deleted commits from GitHub with only the commit's SHA-1 ID. Using our own repos, we experimentally confirmed that this held true for both of GitHub's recommended methods for removing sensitive information: `git filter-branch` or the `bfg` tool [50]. The difficulty in this approach is in acquiring the commit hash, as it is hidden from GitHub's UI and Commits API. However, we found that these hidden commit hashes could be recovered with trivial effort via the Events API [20]. Moreover, historical data from this API is

available through the GHTorrent Project [31]. Taken together, this indicates that the consequences of even rapidly detected secret disclosure is severe and difficult to mitigate short of deleting a repository or reissuing credentials.

IX. Mitigations

We have shown that an attacker with minimal resources could compromise many GitHub users by stealing leaked keys. In this section, we discuss how current mitigations fall short and what new mitigations might be successful. Through our results, we addressed three potential mitigations to our collection methodology and demonstrated their ineffectiveness. First, secret leakage could be stopped prior to commit using tools like TruffleHog [59]. However, in Section VII-D, we found that TruffleHog is only approximately 25% effective. Second, many API platforms require multiple secret values to be used, possibly inhibiting a potential attacker. We showed in Section V-D that complementary secrets are committed in the same file with high probability, nullifying the advantages of this technique in practice. Finally, GitHub imposes strict rate limits for Search API requests to inhibit large-scale mining. Unfortunately, Section VI-A demonstrated this rate limiting can be easily bypassed with a single key.

Fortunately, there is room to improve these mitigations. For example, TruffleHog could detect all of the secrets that we detect if it implements our detection techniques. Second, the fact that many secrets have distinct patterns that simplify accurate detection means that our techniques could be used by services to monitor and instantly alert developers and services of compromise. In fact, there is evidence that AWS may already do this [3], [9]. Finally, while GitHub might consider trying to increase their rate limiting, we argue that this would still be trivial to bypass with multiple keys and would disproportionately affect legitimate users. Instead, GitHub could extend their security alerts program [34] to scan for secrets and notify developers at commit time. GitHub recently introduced a beta version of Token Scanning [16], [58], which scans repos for tokens and contacts the provider with the token and metadata. The provider can then revoke the token mitigating the impact of it's disclosure. This feature can be improved with the findings from this paper, increasing the providers included.

All of these, however, are mitigations, taking action late in the secret's lifetime after it has already been exposed. Ultimately, we believe that more research is needed on the question of secret and key management for *software*. Extensive work has been done in this area for *users* (e.g. passwords and alternative authentication). Two possible approaches to address this earlier in the process are extending Git to handle secrets natively, or changing the architecture of libraries to automatically and securely manage secrets for developers.

X. Limitations

In this section we briefly detail limitations of our experiments. First, we do not have ground truth knowledge that the secrets we discover are exploitable. Though we make every effort to exclude them, some secrets may be non-sensitive, revoked, stale, or simply invalid. Without actually testing such secrets (which we do not do for reasons discussed in Section IV), it is not possible to have certainty that a secret is exploitable.

Second, we focus only on secrets that we felt could be discovered with high probability of validity and sensitivity. There are certainly many important services that we do not detect secrets for. Similarly, while GitHub is the largest public code hosting service, there are many other services where secrets may be leaked, including services like BitBucket or Pastebin. *This means that our findings are a lower bound on the risks of secret leakage through public repositories.*

Finally, for some of the APIs we study we find few leaked keys, as shown in Table II. While we surveyed many public APIs, the relative usage of each API in a project that would be hosted on GitHub will naturally differ. We nevertheless discover a large number of keys overall, including keys for every service we chose to investigate.

XI. Conclusion

GitHub has become the most popular platform for collaboratively editing software, yet this collaboration often conflicts with the need for software to use secret information. This conflict creates the potential for public secret leakage. In this paper, we characterize the prevalence of such leakage. By leveraging two complementary detection approaches, we discover hundreds of thousands of API and cryptographic keys leaked at a rate of thousands per day. This work not only demonstrates the scale of this problem, but also highlights the potential causes of leakage and discusses the effectiveness of existing mitigations. In so doing, we show that secret leakage via public repositories places developers at risk.

References

[1] (2018, Mar.) About GitHub. [Online]. Available: https://github.com/about

[2] (2018) Alexa Top Software Sites. [Online]. Available: https://www.alexa.com/topsites/category/Top/Computers/Software

[3] (2018) AWS Labs git-secrets. [Online]. Available: https://github.com/awslabs/git-secrets

[4] (2014, Mar.) AWS Urges Devs To Scrub Secret Keys From GitHub. [Online]. Available: https://developers.slashdot.org/story/14/03/24/0111203/aws-urges-devs-to-scrub-secret-keys-from-github

[5] M. Balduzzi, J. Zaddach, D. Balzarotti, E. Kirda, and S. Loureiro, "A Security Analysis of Amazon's Elastic Compute Cloud Service," *SAC*, 2012.

[6] C. M. Bishop, *Pattern Recognition and Machine Learning*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2006.

[7] T. F. Bissyande, D. Lo, L. Jiang, L. Reveillere, J. Klein, and Y. L. Traon, "Got issues? Who cares about it? A large scale investigation of issue trackers from GitHub," *International Symposium on Software Reliability Engineering*, 2013.

[8] (2015, Jan.) Bots Scanning GitHub To Steal Amazon EC2 Keys. [Online]. Available: https://it.slashdot.org/story/15/01/02/2342228/bots-scanning-github-to-steal-amazon-ec2-keys

[9] D. Bourke. (2017, Oct.) Breach Detection at Scale. [Online]. Available: https://developer.atlassian.com/blog/2017/10/project-spacecrab-breach-detection/

[10] J. Cabot, J. L. C. Izquierdo, V. Cosentino, and B. Rolandi, "Exploring the use of labels to categorize issues in Open-Source Software projects," in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, March 2015, pp. 550–554.

[11] S. Chacon and B. Straub, *Pro Git*. Apress, 2014.

[12] A. D. Diego, "Automatic extraction of API Keys from Android applications," Ph.D. dissertation, UNIVERSITÀ DEGLI STUDI DI ROMA "TOR VERGATA", 2017.

[13] R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen, "Boa: A Language and Infrastructure for Analyzing Ultra-Large-Scale Software Repositories," *International Conference on Software Engineering*, 2013.

[14] S. Farooqi, F. Zaffar, N. Leontiadis, and Z. Shafiq, “Measuring and Mitigating OAuth Access Token Abuse by Collusion Networks,” in *Proceedings of the 2017 Internet Measurement Conference*, ser. IMC ’17. New York, NY, USA: ACM, 2017, pp. 355–368.

[15] D. Fett, R. Küsters, and G. Schmitz, “A Comprehensive Formal Security Analysis of OAuth 2.0,” in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS ’16. New York, NY, USA: ACM, 2016, pp. 1204–1215. [Online]. Available: http://doi.acm.org/10.1145/2976749.2978385

[16] GitHub. (2018, Oct.) About Token Scanning. [Online]. Available: https://help.github.com/articles/about-token-scanning/

[17] (2018) GitHub API Rate Limiting Documentation. [Online]. Available: https://developer.github.com/v3/#rate-limiting

[18] (2018) GitHub Content API Documentation. [Online]. Available: https://developer.github.com/v3/repos/contents/#get-contents

[19] (2018) GitHub dotfiles. [Online]. Available: https://dotfiles.github.io/

[20] (2018) GitHub Events API Documentation. [Online]. Available: https://developer.github.com/v3/activity/events/

[21] (2013, Jan.) Github Kills Search After Hundreds of Private Keys Exposed. [Online]. Available: https://it.slashdot.org/story/13/01/25/132203/github-kills-search-after-hundreds-of-private-keys-exposed

[22] (2018) GitHub Search API Documentation. [Online]. Available: https://developer.github.com/v3/search/#search-code

[23] (2018) GitHub Search API Rate Limiting Documentation. [Online]. Available: https://developer.github.com/v3/search/#rate-limit

[24] (2018) GitHub Searching Code. [Online]. Available: https://help.github.com/articles/searching-code/

[25] D. Goodin. (2013, Jan.) PSA: Don’t upload your important passwords to GitHub. [Online]. Available: https://arstechnica.com/information-technology/2013/01/psa-dont-upload-your-important-passwords-to-github/

[26] ——. (2018, Mar.) Thousands of servers found leaking 750MB worth of passwords and keys. [Online]. Available: https://arstechnica.com/information-technology/2018/03/thousands-of-servers-found-leaking-750-mb-worth-of-passwords-and-keys/

[27] (2018, Apr.) Google BigQuery GitHub Data. [Online]. Available: https://console.cloud.google.com/marketplace/details/github/github-repos

[28] (2018) Google BigQuery Public Datasets. [Online]. Available: https://cloud.google.com/bigquery/public-data/

[29] (2018) Google Hacking Database. [Online]. Available: https://www.exploit-db.com/google-hacking-database/

[30] (2018, Jan.) Google Using OAuth 2.0 for Installed Applications. [Online]. Available: https://developers.google.com/api-client-library/python/auth/installed-app

[31] G. Gousios, “The GHTorrent Dataset and Tool Suite,” *Mining Software Repositories*, 2013.

[32] (2018) Grawler. [Online]. Available: https://github.com/jregele/grawler

[33] (2017, Feb.) HackerNews GitHub Commit Search for ”Remove Password”. [Online]. Available: https://news.ycombinator.com/item?id=13650818

[34] M. Han. (2017, Nov.) Introducing security alerts on GitHub. [Online]. Available: https://blog.github.com/2017-11-16-introducing-security-alerts-on-github/

[35] R. Heiland, S. Koranda, S. Marru, M. Pierce, and V. Welch, “Authentication and Authorization Considerations for a Multi-tenant Service,” in *Proceedings of the 1st Workshop on The Science of Cyberinfrastructure: Research, Experience, Applications and Models*. ACM, 2015, pp. 29–35.

[36] F. Hoffa. (2016, Jun.) GitHub on BigQuery: Analyze all the open source code. [Online]. Available: https://cloudplatform.googleblog.com/2016/06/GitHub-on-BigQuery-analyze-all-the-open-source-code.html

[37] J. L. C. Izquierdo, V. Cosentino, B. Rolandi, A. Bergel, and J. Cabot, “GiLA: GitHub label analyzer,” in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, March 2015, pp. 479–483.

[38] S. Josefsson and S. Leonard, “Textual Encodings of PKIX, PKCS, and CMS Structures,” *RFC*, vol. 7468, pp. 1–20, 2015.

[39] B. Kaliski, “Public-Key Cryptography Standards (PKCS)# 8: Private-Key Information Syntax Specification Version 1.2,” 2008.

[40] A. Kashcha. (2018) Common Words. [Online]. Available: https://github.com/anvaka/common-words

[41] M. Kumar. (2013, Jan.) Hundreds of SSH Private Keys exposed via GitHub Search. [Online]. Available: https://thehackernews.com/2013/01/hundreds-of-ssh-private-keys-exposed.html

[42] J. Linn, “Privacy Enhancement for Internet Electronic Mail: Part I: Message Encryption and Authentication Procedures,” pp. 1–42, 1993. [Online]. Available: https://tools.ietf.org/html/rfc1421

[43] T. Lodderstedt, M. McGloin, and P. Hunt, “OAuth 2.0 threat model and security considerations,” 2013.

[44] S. Mansfield-Devine, “Google Hacking 101,” *Network Security*, 2009.

[45] D. Mathews. (2018) List of the Most Common English Words. [Online]. Available: https://github.com/dolph/dictionary

[46] R. Mogull. (2014, Jan.) My $500 Cloud Security Screwup. [Online]. Available: https://securosis.com/blog/my-500-cloud-security-screwup

[47] (2018) Paramiko. [Online]. Available: http://www.paramiko.org/

[48] B. Pedro. (2017, Oct.) How to securely store API keys. [Online]. Available: https://dev.to/bpedro/how-to-securely-store-api-keys-ab6

[49] (2018) Public API List. [Online]. Available: https://github.com/toddmotto/public-apis

[50] (2018) Removing Sensitive Data from a Repository. [Online]. Available: https://help.github.com/articles/removing-sensitive-data-from-a-repository/

[51] C. E. Shannon, “Prediction and entropy of printed English,” *Bell Labs Technical Journal*, vol. 30, no. 1, pp. 50–64, 1951.

[52] R. Shu, X. Gu, and W. Enck, “A Study of Security Vulnerabilities on Docker Hub,” *CODASPY*, 2017.

[53] V. S. Sinha, D. Saha, P. Dhoolia, R. Padhye, and S. Mani, “Detecting and Mitigating Secret-Key Leaks in Source Code Repositories,” *Mining Software Repositories*, 2015.

[54] M. Soto, F. Thung, C. P. Wong, C. L. Goues, and D. Lo, “A Deeper Look into Bug Fixes: Patterns, Replacements, Deletions, and Additions,” in *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*, May 2016, pp. 512–515.

[55] (2017) State of the Octoverse. [Online]. Available: https://octoverse.github.com/

[56] S.-T. Sun and K. Beznosov, “The devil is in the (implementation) details: an empirical analysis of OAuth SSO systems,” in *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 2012, pp. 378–390.

[57] J. Terrell, A. Kofink, J. Middleton, C. Rainear, E. Murphy-Hill, C. Parnin, and J. Stallings, “Gender differences and bias in open source: pull request acceptance of women versus men,” *PeerJ Computer Science*, 2017.

[58] P. Toomey. (2018, Oct.) Behind the scenes of GitHub Token Scanning. [Online]. Available: https://blog.github.com/2018-10-17-behind-the-scenes-of-github-token-scanning/

[59] TruffleHog. [Online]. Available: https://github.com/dxa4481/truffleHog

[60] D. Ueltschi. Shannon entropy. [Online]. Available: http://www.ueltschi.org/teaching/chapShannon.pdf

[61] E. v. d. Veen, G. Gousios, and A. Zaidman, “Automatically Prioritizing Pull Requests,” in *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, May 2015, pp. 357–361.

[62] N. Viennot, E. Garcia, and J. Nieh, “A Measurement Study of Google Play,” *ACM SIGMETRICS*, 2014.

[63] J. Wagner. (2015, Jan.) Why Exposed API Keys and Sensitive Data are Growing Cause for Concern. [Online]. Available: https://www.programmableweb.com/news/why-exposed-api-keys-and-sensitive-data-are-growing-cause-concern/analysis/2015/01/05

[64] J. Wei, X. Zhang, G. Ammons, V. Bala, and P. Ning, “Managing Security of Virtual Machine Images in a Cloud Environment,” *CCSW*, 2009.

[65] J. Wright. (2014, Dec.) Why Deleting Sensitive Information from Github Doesn’t Save You. [Online]. Available: https://jordan-wright.com/blog/2014/12/30/why-deleting-sensitive-information-from-github-doesnt-save-you/

[66] Y. Yu, H. Wang, G. Yin, and C. X. Ling, “Reviewer Recommender of Pull-Requests in GitHub,” in *2014 IEEE International Conference on Software Maintenance and Evolution*, Sept 2014, pp. 609–612.

TABLE III: Robust regular expressions can be written to target API credentials for platforms with distinct key structures

Domain	Platform/API	Key Type	Target Regular Expression
Social Media	Twitter	Access Token	`[1-9][0-9]+-[0-9a-zA-Z]{40}`
	Facebook	Access Token	`EAACEdEose0cBA[0-9A-Za-z]+`
	Google YouTube	API Key	`AIza[0-9A-Za-z\-_]{35}`
		OAuth ID	`[0-9]+-[0-9A-Za-z_]{32}\.apps\.googleusercontent\.com`
	Picatic	API Key	`sk_live_[0-9a-z]{32}`
Finance	Stripe	Standard API Key	`sk_live_[0-9a-zA-Z]{24}`
		Restricted API Key	`rk_live_[0-9a-zA-Z]{24}`
	Square	Access Token	`sq0atp-[0-9A-Za-z\-_]{22}`
		OAuth Secret	`sq0csp-[0-9A-Za-z\-_]{43}`
	PayPal Braintree	Access Token	`access_token\$production\$[0-9a-z]{16}\$[0-9a-f]{32}`
	Amazon MWS	Auth Token	`amzn\.mws\.[0-9a-f]{8}-[0-9a-f]{4}-[0-9a-f]{4}-[0-9a-f]{4}-[0-9a-f]{12}`
Communications	Google Gmail	(see YouTube)	(see YouTube)
	Twilio	API Key	`SK[0-9a-fA-F]{32}`
	MailGun	API Key	`key-[0-9a-zA-Z]{32}`
	MailChimp	API Key	`[0-9a-f]{32}-us[0-9]{1,2}`
Storage	Google Drive	(see YouTube)	(see YouTube)
IaaS	Amazon AWS	Access Key ID	`AKIA[0-9A-Z]{16}`
	Google Cloud Platform	(see YouTube)	(see YouTube)

TABLE IV: Asymmetric private keys have a distinct structure mainly due to their PEM header

Asymmetric Key Type	Target Regular Expression

RSA Private Key	`-----BEGIN RSA PRIVATE KEY-----`
	`[\r\n]+(?:\w+:.+)*[\s]*`
	`(?:[0-9a-zA-Z+\/=]{64,76}[\r\n]+)+`
	`[0-9a-zA-Z+\/=]+[\r\n]+`
	`-----END RSA PRIVATE KEY----`

EC Private Key	`-----BEGIN EC PRIVATE KEY-----`
	`[\r\n]+(?:\w+:.+)*[\s]*`
	`(?:[0-9a-zA-Z+\/=]{64,76}[\r\n]+)+`
	`[0-9a-zA-Z+\/=]+[\r\n]+`
	`-----END EC PRIVATE KEY-----`

PGP Private Key	`-----BEGIN PGP PRIVATE KEY BLOCK-----`
	`[\r\n]+(?:\w+:.+)*[\s]*`
	`(?:[0-9a-zA-Z+\/=]{64,76}[\r\n]+)+`
	`[0-9a-zA-Z+\/=]+[\r\n]+=`
	`[0-9a-zA-Z+\/=]{4}[\r\n]+`
	`-----END PGP PRIVATE KEY BLOCK-----`

General Private Key	`-----BEGIN PRIVATE KEY-----`
	`[\r\n]+(?:\w+:.+)*[\s]*`
	`(?:[0-9a-zA-Z+\/=]{64,76}[\r\n]+)+`
	`[0-9a-zA-Z+\/=]+[\r\n]+`
	`-----END PRIVATE KEY-----`

APPENDIX

A. *File Extension Categories*

The experiment in Section VII-B grouped file extensions into various categories. Those categories and the extensions they contained were: **crypto** (crt, gpg, key, p12, pem, pkey, ppk, priv, rsa), **code** (aspx, c, cpp, cs, cshtml, ejs, erb, go, h, html, ipynb, js, jsp, jsx, php, phtml, py, rb, sh, swift, ts, twig, vue, xhtml), **data** (csv, dat, json, log, md, txt, xml, yaml), and **config** (cfg, conf, config, ini, ovpn, plist, properties).

TABLE V: As GitHub does not allow regular expression searches, these targeted queries identify candidate files which are then scanned offline for secrets

Type	Search Query	Targeted Secret

General	`access_token`
	`access_secret`
	`api_key`
	`client_secret`
	`consumer_secret`
	`customer_secret`
	`user_secret`
	`secret_key`
		API Secret

Specific	`-----BEGIN RSA PRIVATE KEY-----`
	`-----BEGIN EC PRIVATE KEY-----`
	`-----BEGIN PRIVATE KEY-----`
	`-----BEGIN PGP PRIVATE KEY BLOCK-----`
		Private Key

	`AKIA`	AWS Access Key ID
	`EAA, EAACEd, EAACEdEose0cBA`	Facebook Access Token
	`AIza`	Google API Key
	`.apps.googleusercontent.com`	Google OAuth ID
	`sq0atp`	Square Access Token
	`sq0csp`	Square OAuth Secret
	`key-`	MailGun API Key
	`sk_live_`	Picatic/Stripe API Key
	`rk_live_`	Stripe Restricited API Key

TABLE VI: Some of our distinct secrets may be leaked with additional information

Distinct Secret	Parallel Target(s)	Parallel Target Regular Expression
Amazon AWS Access Key ID	Client Secret	`[0-9a-zA-Z/+=]{40}`
Amazon MWS Auth Token	AWS Client ID	`AKIA[0-9A-Z]{16}`
	AWS Secret Key	`[0-9a-zA-Z/+=]{40}`
Google OAuth ID	OAuth Secret	`[0-9a-zA-Z\-_]{24}`
	OAuth Auth Code	`4/[0-9A-Za-z\-_]+`
	OAuth Refresh Token	`1/[0-9A-Za-z\-_]{43}|` `1/[0-9A-Za-z\-_]{64}`
	OAuth Access Token	`ya29\.[0-9A-Za-z\-_]+`
	API Key	`AIza[0-9A-Za-z\-_]{35}`
Twilio API Key	API Secret	`[0-9a-zA-Z]{32}`
Twitter Access Token	Access Token Secret	`[0-9a-zA-Z]{45}`